# Projeto 2 - Algoritmos de ordenação

Lucas Dúckur Nunes Andreolli - N° USP: 15471518
Rafael Perez Carmanhani - N° USP: 15485420

## 1 Descrição do problema

Entende-se por ordenação a organização de uma sequência de elementos de modo que eles estabeleçam alguma relação de ordem. Desse modo, dado um conjunto de dados (registros) em que cada dado é representado por uma chave, busca-se organizá-los com o objetivo de se manter uma ordem entre as chaves. No caso, a ordenação é feita a partir de números inteiros (os registros são as variáveis inteiras e as chaves os valores desses inteiros) e a relação de ordem é a ordem convencional dos números reais. Assim, as implementações apresentadas dos algoritmos de ordenação estudados levam em consideração uma ordenação em ordem crescente: todo número na sequência deve ser menor ou igual ao número a sua direita.

A análise dos algoritmos foi feita de acordo com numerosos testes empíricos que buscam avaliar três aspectos: tempo de execução; quantidade de comparações de chaves e quantidade de movimentações de registros. Os tempos de execução dos algoritmos foram medidos por meio da função *clock()*, definida na biblioteca *time.h*; as quantidades de comparações de chaves, bem como de movimentações de registros, foram obtidas por meio de contadores presentes nas implementações dos algoritmos. Vale ressaltar que as comparações contadas são apenas as comparações realizadas entre as chaves dos elementos a serem ordenados, desconsiderando aquelas necessárias, por exemplo, para se executar um laço de repetição. Ademais, entende-se por uma movimentação de registro toda atribuição feita a algum elemento do vetor; desse modo, uma operação de troca ("*swap*") é contabilizada como duas movimentações.

Os testes feitos compreendem uma variedade de tamanhos de entrada e de formatações da entrada. Foram realizados testes com entradas de 100, 1000, 10000 e 100000 inteiros, sendo que, para cada um desses tamanhos, foram realizados testes com entradas ordenadas, inversamente ordenadas e aleatórias.

## 2 Comparação entre as quantidade de comparações de chaves

### 2.1 Entradas ordenadas

Os resultados apresentados a seguir são referentes à quantidade de comparações de chaves que cada algoritmo realizou considerando que a sequência dos inteiros de entrada já estava ordenada. Os resultados brutos são sumarizados na *Tabela 1.*

Para melhor visualização dos resultados obtidos, foi feito um gráfico que compara a ordem de magnitude da quantidade de comparações dos algoritmos a depender do tamanho da entrada. Uma vez que o *Radix Sort* é um algoritmo de ordenação que não se baseia em comparações, ele não realiza comparações entre chaves e, portanto, é omitido do gráfico. A comparação entre as ordens de magnitude encontra-se no *Gráfico 1.*

*Tabela 1: Quantidade de comparações por algoritmo e por tamanho de entrada (ordenada)*

| Algoritmo | n = 100 | n = 1000 | n = 10000 | n = 100000 |
|---|---|---|---|---|
| Bubble Sort | 99 | 999 | 9999 | 99999 |
| Selection Sort | 4950 | 499500 | 49995000 | 4999950000 |
| Insertion Sort | 99 | 999 | 9999 | 99999 |
| Shell Sort | 503 | 8006 | 120005 | 1500006 |
| Quick Sort | 1002 | 13005 | 165435 | 2000012 |
| Merge Sort | 672 | 9976 | 133616 | 1668928 |
| Heap Sort | 1081 | 17538 | 244460 | 3112517 |
| Radix Sort | 0 | 0 | 0 | 0 |
| Contagem dos menores | 4950 | 499500 | 49995000 | 4999950000 |

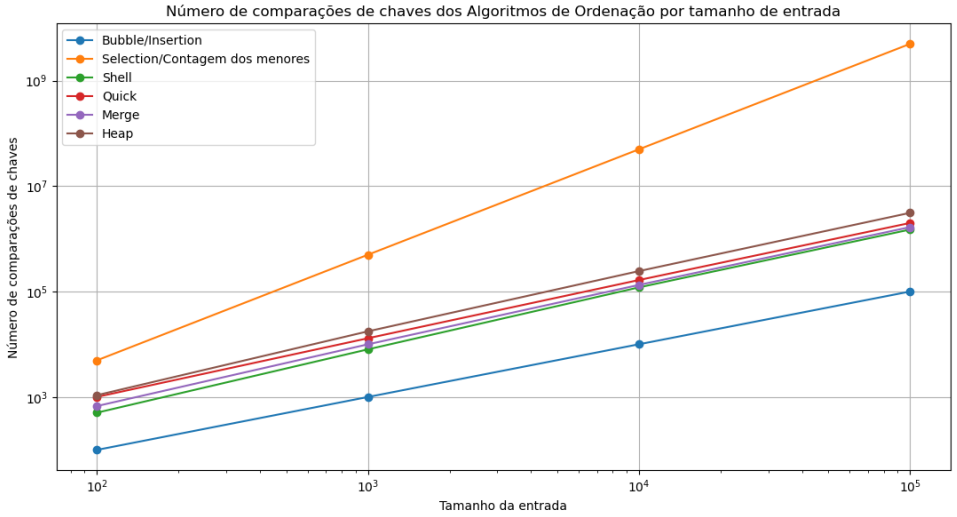


*Gráfico 1: ordens de magnitude das quantidades de comparações de chaves dos algoritmos (entradas ordenadas)*

## 2.2 Entradas inversamente ordenadas

Os resultados apresentados a seguir são referentes à quantidade de comparações de chaves que cada algoritmo realizou considerando que a sequência dos inteiros de entrada estava inversamente ordenada (no caso, isto significa estar ordenado em ordem decrescente). Os resultados brutos são sumarizados na *Tabela 2*.

Novamente, a visualização dos dados é facilitada com um gráfico da magnitude deles. As quantidades de comparações do algoritmo *Radix Sort* são omitidas de modo similar às entradas anteriores. A comparação das magnitudes das quantidades de comparações de chaves encontra-se no *Gráfico 2*.

*Tabela 2: Quantidade de comparações por algoritmo e por tamanho de entrada (inversamente ordenada)*

| Algoritmo | n = 100 | n = 1000 | n = 10000 | n = 100000 |
|---|---|---|---|---|
| Bubble Sort | 4950 | 499500 | 49995000 | 4999950000 |
| Selection Sort | 4950 | 499500 | 49995000 | 4999950000 |
| Insertion Sort | 4950 | 499500 | 49995000 | 4999950000 |
| Shell Sort | 668 | 11716 | 172578 | 2244585 |
| Quick Sort | 1010 | 13016 | 165452 | 2000030 |
| Merge Sort | 672 | 9976 | 133616 | 1668928 |
| Heap Sort | 944 | 15965 | 226682 | 2926640 |
| Radix Sort | 0 | 0 | 0 | 0 |
| Contagem dos menores | 4950 | 499500 | 49995000 | 4999950000 |

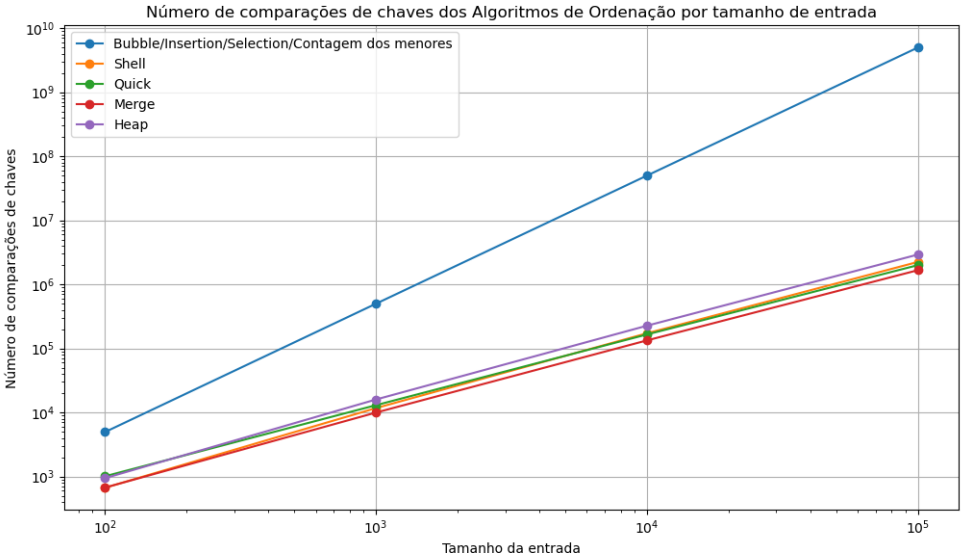


*Gráfico 2: ordens de magnitude das quantidades de comparações de chaves dos algoritmos (entradas inversamente ordenadas)*

## 2.3 Entradas aleatórias

Os resultados apresentados a seguir são referentes à quantidade de comparações de chaves que cada algoritmo realizou considerando que a sequência dos inteiros de entrada é aleatória. Para que as medições não fossem dependentes das entradas usadas, foram realizados cinco testes para cada algoritmo e para cada tamanho de entrada, de modo que os resultados obtidos são fruto da média aritmética desses testes. Os resultados brutos são sumarizados na *Tabela 3*.

*Tabela 3: Quantidade de comparações por algoritmo e por tamanho de entrada (aleatória)*

| Algoritmo | n = 100 | n = 1000 | n = 10000 | n = 100000 |
|---|---|---|---|---|
| Bubble Sort | 4922 | 498761 | 49985294 | 4999815658 |
| Selection Sort | 4950 | 499500 | 49995000 | 4999950000 |
| Insertion Sort | 2595 | 255231 | 25168315 | 2500872913 |
| Shell Sort | 873 | 15205 | 264812 | 4370997 |
| Quick Sort | 1383 | 18049 | 225017 | 2634837 |
| Merge Sort | 672 | 9976 | 133616 | 1668928 |
| Heap Sort | 1029 | 16863 | 235343 | 3019826 |
| Radix Sort | 0 | 0 | 0 | 0 |
| Contagem dos menores | 4950 | 499500 | 49995000 | 4999950000 |

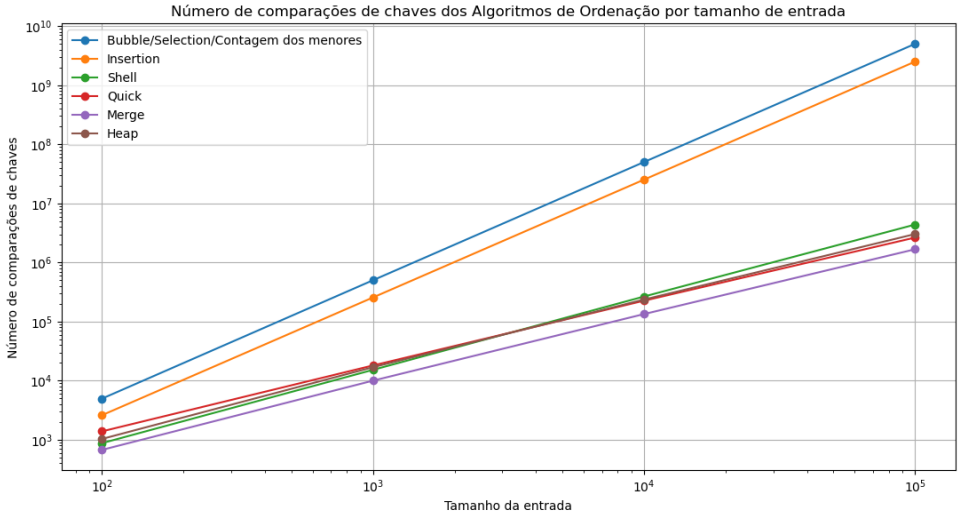


*Gráfico 3: ordens de magnitude das quantidades de comparações de chaves dos algoritmos (entradas aleatórias)*

## 2.4 Análise dos resultados

O algoritmo que exigiu a menor quantidade de comparações é, obviamente, o *Radix Sort*, que não é um algoritmo baseado em comparações e, por conseguinte, não exige nenhuma.

Os algoritmos que exigiram o maior número de comparações foram o *Selection Sort* e o de *Contagem dos menores*, visto que, independente da formatação da entrada, a quantidade $C(n)$ de comparações que os algoritmos exigem é igual a $C(n) = \frac{n \cdot (n-1)}{2}$, em que $n$ é o tamanho da entrada.

Os algoritmos *Bubble Sort* e *Insertion Sort* obtiveram os melhores desempenhos quando as entradas já estavam ordenadas (nesses cenários, a quantidade de comparações é $C(n) = n - 1$), pois este é o melhor caso dos algoritmos. Contudo, com entradas inversamente ordenadas, pior caso deles, eles exigiram uma quantidade de comparações idêntica ao *Selection Sort* e o *Contagem dos menores*.

O *Shell Sort*, o *Quick Sort*, o *Merge Sort* e o *Heap Sort* exigiram números parecidos de comparações, sendo que o *Merge Sort* exigiu o menor número. Dentre eles, o *Heap Sort* obteve o pior desempenho (no aspecto analisado) devido à complexidade do seu anel interno, que realiza diversas comparações.

# 3 Comparação entre as quantidades de trocas de registros

## 3.1 Entradas ordenadas

Os resultados apresentados a seguir são referentes à quantidade de movimentações de registros que cada algoritmo necessitou considerando que a sequência dos inteiros de entrada já estava ordenada. Os resultados brutos são sumarizados na *Tabela 4*.

As ordens de magnitude dos dados obtidos encontram-se no *Gráfico 4*. Uma vez que, quando as entradas estão ordenadas, os algoritmos *Bubble Sort* e *Selection Sort* não exigem movimentações de registros, eles são omitidos do gráfico.

*Tabela 4: Quantidade de trocas de registros por algoritmo e por tamanho de entrada (ordenada)*

| Algoritmo | n = 100 | n = 1000 | n = 10000 | n = 100000 |
|---|---|---|---|---|
| Bubble Sort | 0 | 0 | 0 | 0 |
| Selection Sort | 0 | 0 | 0 | 0 |
| Insertion Sort | 99 | 999 | 9999 | 99999 |
| Shell Sort | 503 | 8006 | 120005 | 1500006 |
| Quick Sort | 126 | 1022 | 11808 | 131070 |
| Merge Sort | 672 | 9976 | 133616 | 1668928 |
| Heap Sort | 1280 | 19416 | 263912 | 3301708 |
| Radix Sort | 1200 | 12000 | 120000 | 1200000 |
| Contagem dos menores | 100 | 1000 | 10000 | 100000 |

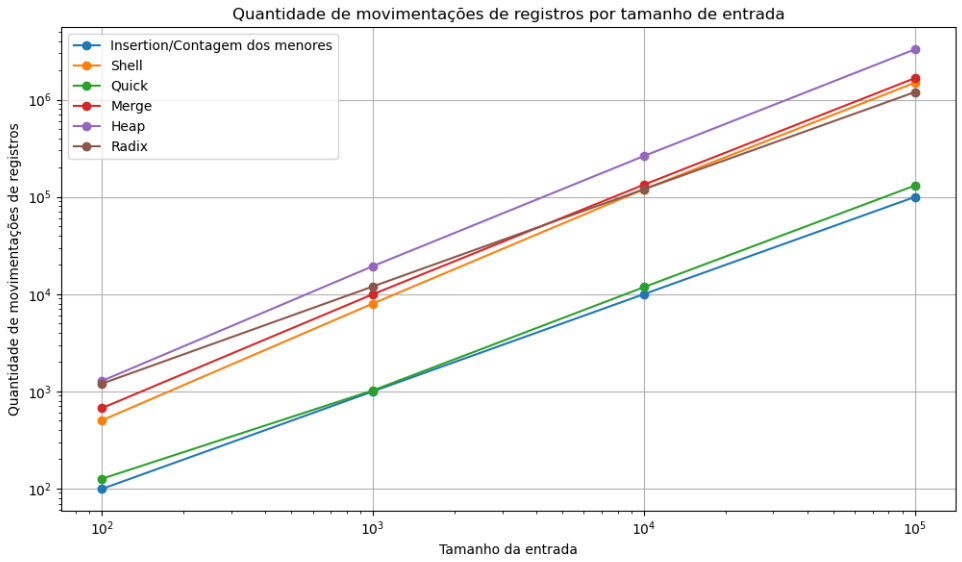


*Gráfico 4: ordens de magnitude das quantidades de trocas de registros dos algoritmos (entradas ordenadas)*

## 3.2 Entradas inversamente ordenadas

Os resultados apresentados a seguir são referentes à quantidade de movimentações de registros que cada algoritmo realizou considerando que a sequência dos inteiros de entrada estava inversamente ordenada. Os resultados brutos são sumarizados na *Tabela 5*.

A comparação entre as magnitudes dos dados é apresentada no *Gráfico 5*.

*Tabela 5: Quantidade de movimentações de registros por algoritmo e por tamanho de entrada (inversamente ordenada)*

| Algoritmo | n = 100 | n = 1000 | n = 10000 | n = 100000 |
|---|---|---|---|---|
| Bubble Sort | 9900 | 999000 | 99990000 | 9999900000 |
| Selection Sort | 100 | 1000 | 10000 | 100000 |
| Insertion Sort | 5049 | 500499 | 50004999 | 5000049999 |
| Shell Sort | 763 | 12706 | 182565 | 2344566 |
| Quick Sort | 224 | 2020 | 21808 | 231068 |
| Merge Sort | 672 | 9976 | 133616 | 1668928 |
| Heap Sort | 1032 | 16632 | 233392 | 2994868 |
| Radix Sort | 1200 | 12000 | 120000 | 1200000 |
| Contagem dos menores | 100 | 1000 | 10000 | 100000 |

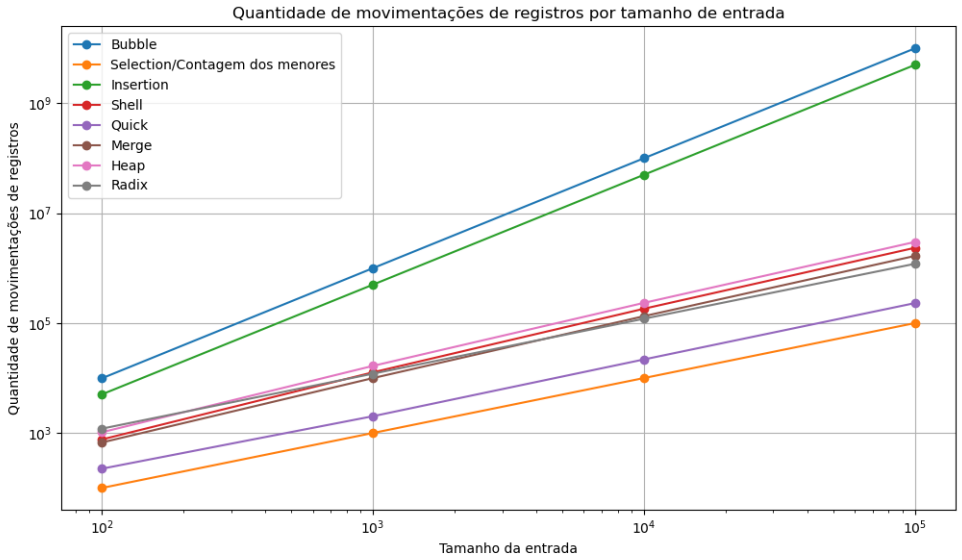


*Gráfico 5: ordens de magnitude das quantidades de movimentações de registros dos algoritmos (entradas ordenadas)*

## 3.3 Entradas aleatórias

Os resultados apresentados a seguir são referentes ao tempo de execução de cada algoritmo considerando que a sequência dos inteiros de entrada é aleatória. Mais uma vez, foram realizados cinco testes para cada algoritmo e para cada tamanho de entrada, de modo que os resultados obtidos são fruto da média aritmética desses testes. Os resultados brutos são sumarizados na *Tabela 6*. O gráfico que compara a ordem de magnitude dos resultados obtidos é o (*Gráfico 6*).

*Tabela 6: Quantidade de movimentações de registros por algoritmo e por tamanho de entrada (aleatória)*

| Algoritmo | n = 100 | n = 1000 | n = 10000 | n = 100000 |
|---|---|---|---|---|
| Bubble Sort | 5001 | 508478 | 50316648 | 5001545856 |
| Selection Sort | 189 | 1981 | 19981 | 199975 |
| Insertion Sort | 2600 | 255238 | 25168323 | 2500872927 |
| Shell Sort | 926 | 15732 | 269907 | 4421501 |
| Quick Sort | 396 | 5611 | 71359 | 873588 |
| Merge Sort | 672 | 9976 | 133616 | 1668928 |
| Heap Sort | 1163 | 18162 | 248328 | 3150291 |
| Radix Sort | 1200 | 12000 | 120000 | 1200000 |
| Contagem dos menores | 100 | 1000 | 10000 | 100000 |

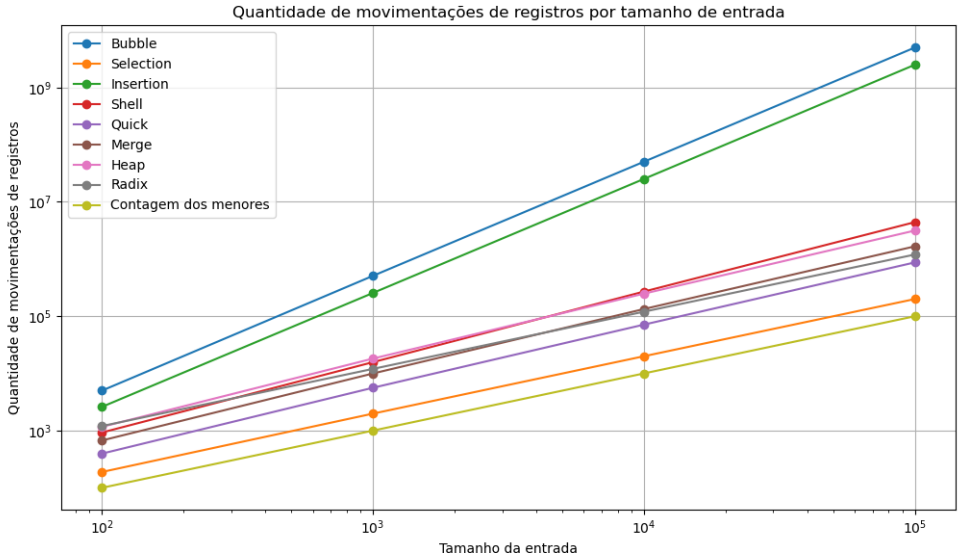


*Gráfico 6: ordens de magnitude das quantidades de movimentações de registros dos algoritmos (entradas aleatórias)*

## 3.4 Análise dos resultados

Em primeiro lugar, os algoritmos *Bubble Sort* e *Selection Sort* obtiveram performance ideal no que tange o critério de quantidade de movimentações de registros quando as entradas já estavam ordenadas, visto que, nesses cenários, eles não exigem movimentações. Em compensação, o *Bubble Sort* obteve as piores performances com entradas aleatórias ou inversamente ordenadas, o que mostra que, quando as entradas não estão ordenadas, ele realiza diversos movimentos desnecessários. De modo similar, o *Insertion Sort* demonstrou excelente desempenho (relativo ao critério em questão) com vetores de entrada ordenados, mas foi bastante penalizado com outras formatações de entrada, superando apenas o *Bubble Sort* em listas de inteiros inversamente ordenadas ou aleatórias.

Para entradas não-ordenadas, o algoritmo de *Contagem dos menores* movimentou menos os registros na medida em que, em toda movimentação, o algoritmo posiciona um elemento da lista em sua posição correta. Dessa forma, apesar do bom desempenho do *Selection Sort* em todos os casos, ele é superado pelo *Contagem dos menores* nos casos não ordenados.

Os algoritmos *Shell Sort*, *Quick Sort*, *Merge Sort*, *Radix Sort* e *Heap Sort* exigiram números de movimentações de magnitudes parecidas, sendo que o *Quick Sort* foi o mais eficiente nesse quesito. Dentre eles, o *Heap Sort* foi o que obteve a pior perfomance, haja vista que exigiu, por exemplo, praticamente o dobro de movimentações de registros se comparado ao *Merge Sort*, o que evidencia novamente a complexidade de seu anel interno.

# 4 Comparação entre os tempos de execução

## 4.1 Entradas ordenadas

Os resultados apresentados a seguir são referentes ao tempo de execução (em segundos) que cada algoritmo necessitou considerando que a sequência dos inteiros de entrada já estava ordenada. Os resultados brutos são sumarizados na *Tabela 7* e possuem precisão de seis casas decimais. Para melhor visualização dos resultados obtidos, foi feito um gráfico que compara a ordem de magnitude dos tempos de execução dos algoritmos a depender do tamanho da entrada (*Gráfico 7*).

*Tabela 7: Tempo de execução (em segundos) por algoritmo e por tamanho de entrada (ordenada)*

| Algoritmo | n = 100 | n = 1000 | n = 10000 | n = 100000 |
|---|---|---|---|---|
| Bubble Sort | 0.000006 | 0.000015 | 0.000057 | 0.000548 |
| Selection Sort | 0.000062 | 0.003097 | 0.248455 | 23.650144 |
| Insertion Sort | 0.000005 | 0.000010 | 0.000077 | 0.000758 |
| Shell Sort | 0.000010 | 0.000070 | 0.000986 | 0.012227 |
| Quick Sort | 0.000015 | 0.000062 | 0.000768 | 0.009262 |
| Merge Sort | 0.002976 | 0.008736 | 0.041015 | 0.399908 |
| Heap Sort | 0.000028 | 0.000337 | 0.004300 | 0.052867 |
| Radix Sort | 0.000839 | 0.007715 | 0.076311 | 0.772723 |
| Contagem dos menores | 0.000054 | 0.004274 | 0.287309 | 28.771584 |

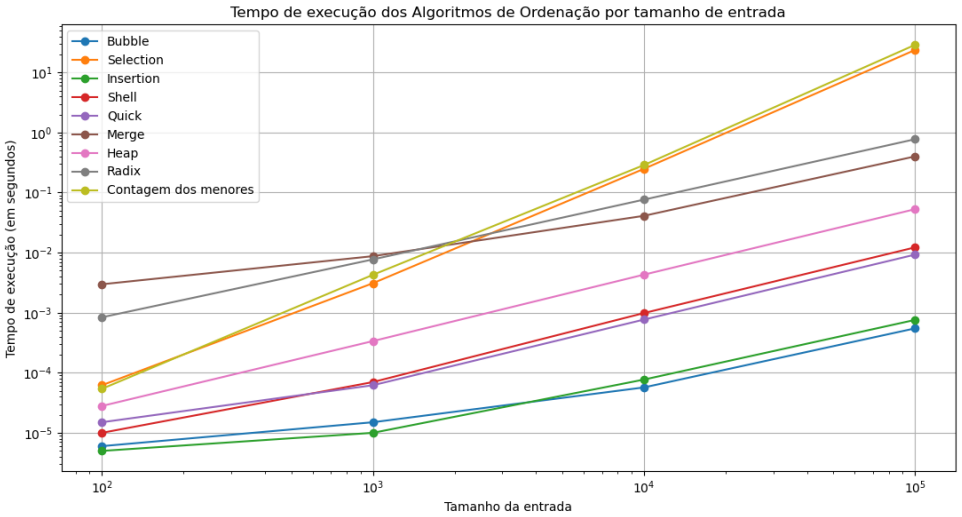


*Gráfico 7: ordens de magnitude dos tempos de execução dos algoritmos (entradas ordenadas)*

## 4.2 Entradas inversamente ordenadas

Os resultados apresentados a seguir são referentes ao tempo de execução (em segundos) que cada algoritmo necessitou considerando que a sequência dos inteiros de entrada estava inversamente ordenada. Os resultados brutos são sumarizados na *Tabela 8* e possuem uma precisão de seis casas decimais. Para melhor visualização dos resultados obtidos, foi feito um gráfico que compara a ordem de magnitude dos tempos de execução dos algoritmos a depender do tamanho da entrada (*Gráfico 8*).

*Tabela 8: Tempo de execução (em segundos) por algoritmo e por tamanho de entrada (inversamente ordenada)*

| Algoritmo | n = 100 | n = 1000 | n = 10000 | n = 100000 |
|---|---|---|---|---|
| Bubble Sort | 0.000097 | 0.005565 | 0.548586 | 51.503974 |
| Selection Sort | 0.000054 | 0.002547 | 0.249906 | 25.001232 |
| Insertion Sort | 0.000107 | 0.003311 | 0.327883 | 32.930939 |
| Shell Sort | 0.000017 | 0.000106 | 0.001534 | 0.018666 |
| Quick Sort | 0.000016 | 0.000064 | 0.000781 | 0.009381 |
| Merge Sort | 0.002961 | 0.008866 | 0.041081 | 0.399076 |
| Heap Sort | 0.000024 | 0.000314 | 0.004062 | 0.049695 |
| Radix Sort | 0.000858 | 0.007884 | 0.079174 | 0.792761 |
| Contagem dos menores | 0.000060 | 0.002750 | 0.269188 | 27.016966 |

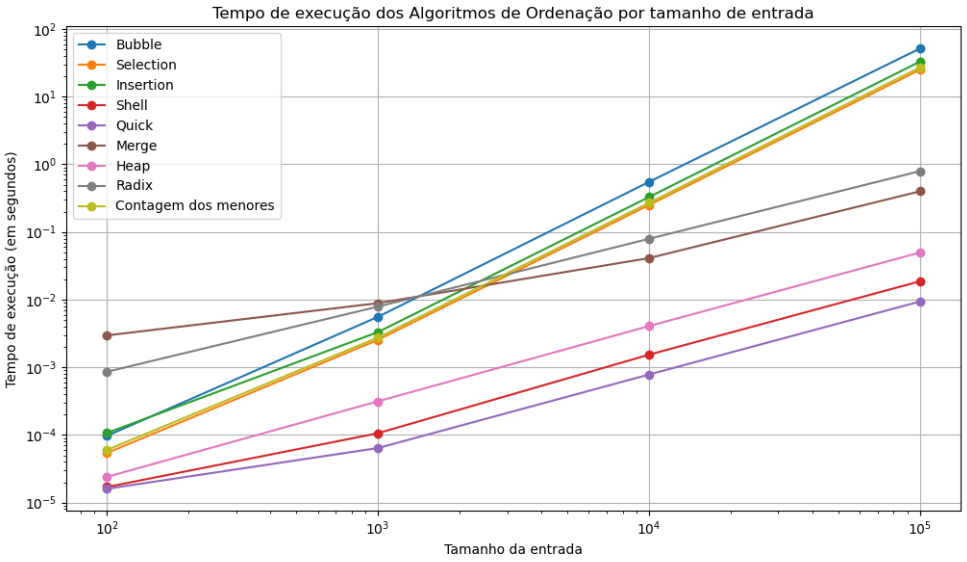


*Gráfico 8: ordens de magnitude dos tempos de execução dos algoritmos (entradas inversamente ordenadas)*

## 4.3 Entradas aleatórias

Os resultados apresentados a seguir são referentes ao tempo de execução de cada algoritmo considerando que a sequência dos inteiros de entrada é aleatória. Para que as medições não fossem dependentes das entradas usadas, foram realizados cinco testes para cada algoritmo e para cada tamanho de entrada, de modo que os resultados obtidos são fruto da média aritmética desses testes. Os resultados brutos são sumarizados na *Tabela 9*. Para melhor visualização dos resultados obtidos, foi feito um gráfico que compara a ordem de magnitude dos tempos de execução dos algoritmos a depender do tamanho da entrada (*Gráfico 9*).

*Tabela 9: Tempo de execução (em segundos) por algoritmo e por tamanho de entrada (aleatória)*

| Algoritmo | n = 100 | n = 1000 | n = 10000 | n = 100000 |
|---|---|---|---|---|
| Bubble Sort | 0.000108 | 0.006813 | 0.589378 | 69.185119 |
| Selection Sort | 0.000062 | 0.003300 | 0.237820 | 23.679675 |
| Insertion Sort | 0.000036 | 0.00227 | 0.165778 | 16.623128 |
| Shell Sort | 0.000036 | 0.000349 | 0.004002 | 0.058014 |
| Quick Sort | 0.000039 | 0.000304 | 0.002580 | 0.030102 |
| Merge Sort | 0.003074 | 0.011923 | 0.043541 | 0.427472 |
| Heap Sort | 0.000032 | 0.000554 | 0.004988 | 0.064197 |
| Radix Sort | 0.000980 | 0.012508 | 0.080850 | 0.843510 |
| Contagem dos menores | 0.000076 | 0.008329 | 0.565497 | 56.535670 |

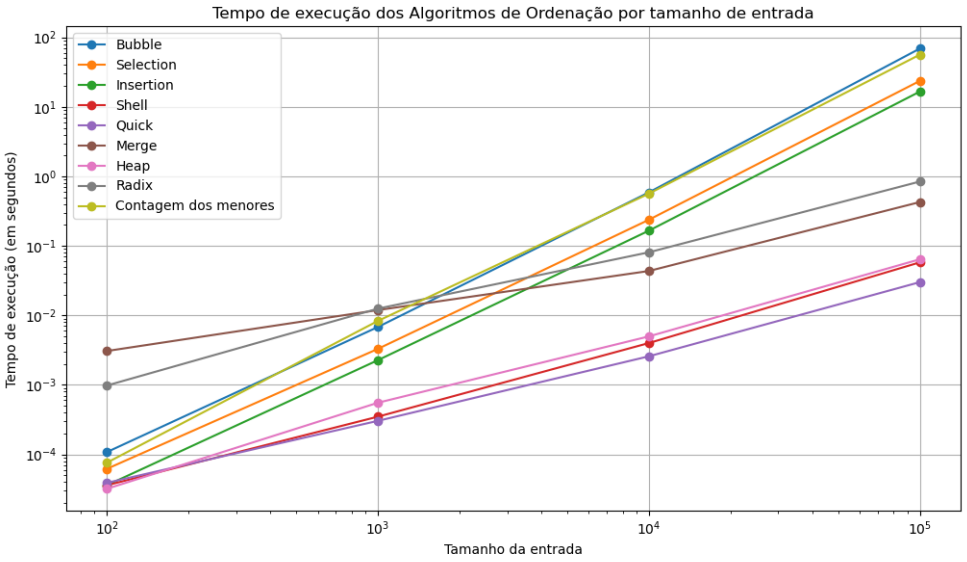


*Gráfico 9: ordens de magnitude dos tempos de execução dos algoritmos (entradas aleatórias)*

### 4.4 Análise dos resultados

Os dados obtidos demonstram que os algoritmos *Bubble Sort* e *Insertion Sort* são muito eficientes (com relação ao tempo de execução) quando as entradas estão ordenadas, pois, nesses casos, eles realizam poucas comparações (como exibido na *Seção 2*) e poucas trocas de registros (*Seção 3*). No entanto, esses algoritmos não obtiveram performances ótimas com outros tipos de entradas, haja vista que eles realizam muitas comparações e muitas movimentações de registros.

Os algoritmos *Selection Sort* e *Contagem dos menores* apresentaram eficiências baixas em todos os casos teste, porque, apesar de não moverem tanto os registros, realizam muito mais comparações que os outros (*Seção 2*).

Dentre os algoritmos de complexidade sub-quadrática (caso médio), os mais lentos foram o *Merge Sort* e o *Radix Sort*. Apesar de não realizarem muitas comparações ou trocas de registros (*Seção 2* e *Seção 3*), o *Merge Sort* realiza diversas outras operações ao longo de sua execução, tais como alocação de memória, cópia de vetores e, o mais importante, o processo de intercalação, enquanto que o *Radix Sort* lida com dez listas encadeadas realizando inserções, remoções e alocações, de modo que todos esses processos acabam por exigir mais tempo para serem executados.

Os algoritmos restantes - *Heap Sort*, *Shell Sort* e *Quick Sort* - mostraram-se bastante eficientes, sendo que o o mais veloz foi o *Quick Sort*, o qual geralmente realiza menos comparações e menos trocas de registros (*Seção 2* e *Seção 3*). A escolha do pivot como sendo a mediana entre o primeiro, o último e o elemento do meio da lista obteve êxito: mesmo nos casos em que a entrada estava ordenada ou inversamente ordenada, ele foi um dos três algoritmos mais rápidos, evidenciando que o pior caso foi evitado.

*Observação:* o tempo de execução do algoritmo *Bubble Sort* foi maior nos casos aleatórios do que nos casos inversamente ordenados (vide *Tabela 8* e *Tabela 9*), embora o número de comparações e de movimentações de registros seja muito maior nos últimos casos (pior caso do algoritmo). Acredita-se que isso ocorra devido ao fenômeno de *"Branch Prediction"* ("Previsão de ramificação", em tradução livre), que acontece quando o processador do computador tenta prever os resultados de desvios condicionais. Na medida em que, a cada iteração do laço de repetição interno, a comparação entre elementos consecutivos sempre resulta que os elementos estão fora de ordem quando as entradas estão inversamente ordenadas, essa técnica computacional sempre acerta as previsões e, assim, o algoritmo é executado mais rapidamente.

## 5 Análise Final

Todos os resultados das comparações realizadas permitem concluir que, quando a lista de entrada já está ordenada, os algoritmos *Bubble Sort* e *Insertion Sort* apresentaram os melhores desempenhos: além de exigirem menos comparações (apenas o *Radix Sort* é superior nesse quesito; vide *Seção 2*) e menos movimentações de registros (*Seção 3*), eles demonstraram menores tempos de execução (*Seção 4*). Isso ocorre devido ao fato desse cenário ser justamente o melhor caso dos algoritmos, em que eles apresentam complexidade linear ($O(n)$).

No entanto, os algoritmos de complexidade quadrática ($O(n^2)$) - *Bubble Sort*; *Selection Sort*; *Insertion Sort* e *Contagem dos menores* - foram os menos eficientes com entradas não-ordenadas, porque ou exigiam muitas comparações (*Selection Sort* e *Contagem dos menores*), ou exigiam muitas movimentações de registros (*Insertion Sort*) ou ambos (*Bubble Sort*). Desse modo, os tempos de execução desses algoritmos foram, no geral, muito maiores do que os dos algoritmos de complexidade sub-quadrática (*Tabela 8* e *Tabela 9*).

Dentre os algoritmos de complexidade sub-quadrática (inclui-se aqui o *Shell Sort*, cujo caso médio é igual a ($O(n \log n)$ na implementação utilizada, em que os *gaps* são reduzidos pela metade a cada iteração), o *Radix Sort* e o *Merge Sort* foram os mais lentos embora não exigissem muitas comparações ou trocas de registros, como explicado na *Seção 4*; o *Shell Sort* e o *Quick Sort* foram, no geral, mais eficientes do que o *Heap Sort*, cujo anel interno é complexo, tornando-o mais lento que os outros dois algoritmos.

Por fim, o *Quick Sort* foi o mais eficiente em todas as comparações (com exceção de quando as entradas já estavam ordenadas, nas quais os desempenhos do *Bubble Sort* e do *Insertion Sort* são superiores), o que evidencia sua eficiência e o sucesso da estratégia de escolha do pivot.