UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS RUSSAS

GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

DISCIPLINA PROJETO E ANÁLISE DE ALGORITMOS

PROFESSOR DR. MAYRTON DIAS DE QUEIROZ

Relatório

Análise experimental

Johnath Silva da Costa - 476651

Russas – CE

2025

SUMÁRIO

1. Introdução

Os testes foram implementados na linguagem Python 3.11, escolhida por sua simplicidade de escrita, funções prontas e clareza na estruturação dos algoritmos. Os experimentos foram realizados em um computador pessoal com as seguintes especificações: Sistema operacional: Windows 11 (64 bits) , Processador: AMD Ryzen 5 5600x (6 núcleos / 12 threads), Memória RAM: 16 GB DDR4 3200mhz.

A metodologia utilizada consistiu na utilização de um vetor base de 10 elementos, que foi replicado para formar listas com tamanhos variando de n = 10 até n = 200, com incremento de 10 em 10, como solicitado. Para cada tamanho, os algoritmos foram executados em três diferentes configurações de entrada: Vetor original (em ordem aleatória) vetor ordenado em ordem crescente, vetor ordenado em ordem decrescente. Posteriormente, foram comparados os sete algoritmos de ordenação ao mesmo tempo, nos três diferentes cenários. É importante destacar que os testes foram executados apenas uma vez para cada caso, e em um ambiente de execução real (um computador pessoal com sistema multitarefa), sujeito a oscilações naturais de desempenho do processador, uso de memória e processos paralelos em execução.

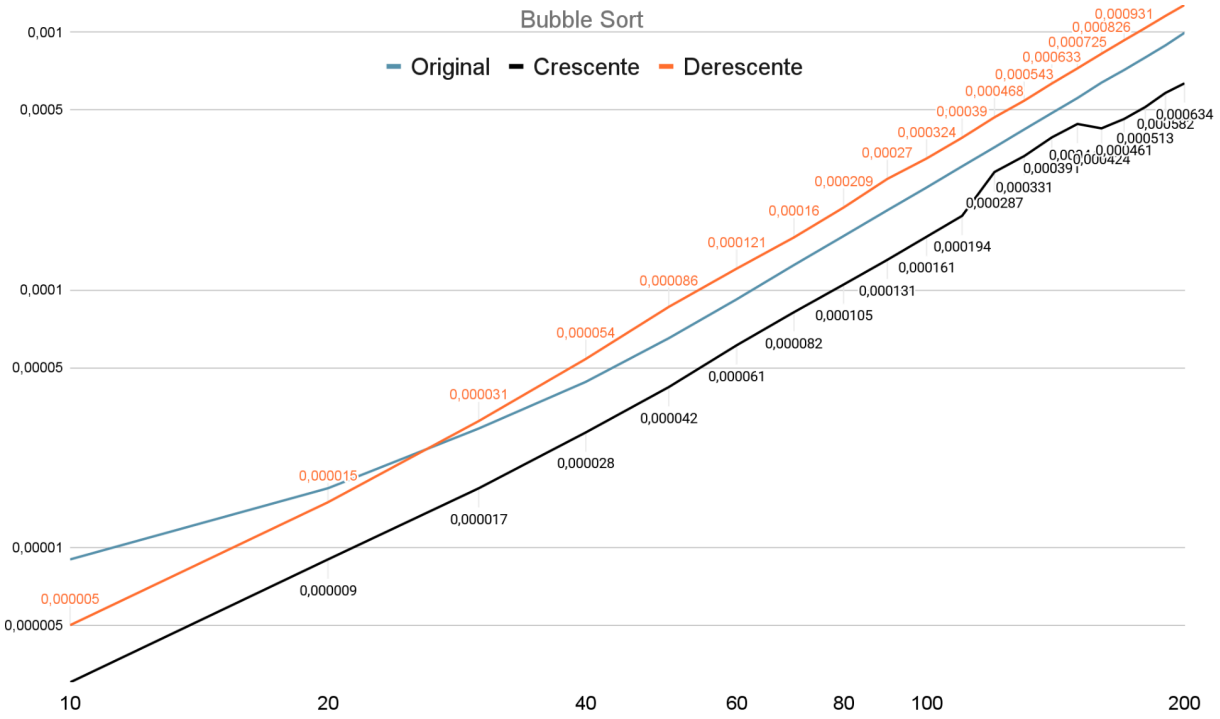
O tempo de execução de cada algoritmo foi medido com a função *time.time()* da biblioteca padrão do Python, capturando o tempo real em segundos. A repetição de testes e a variação dos dados de entrada permitiram observar o comportamento de cada algoritmo frente a diferentes cenários, possibilitando comparações práticas entre os casos médio, melhor e pior.

2. Análise Bubble Sort

Nesta análise, o algoritmo Bubble Sort foi aplicado sobre vetor:

[10, -74, -4, -38, -90, 16, 28, 65, -55, 19].

Gráfico 1: Desempenho - Bubble Sort



Os resultados do Bubble Sort demonstraram com clareza seu comportamento esperado de crescimento O(n²), com tempos aumentando de forma estável à medida que 'n' cresce. No caso dos vetores com dados em ordem aleatória(original), o tempo passou de 0.000009s para 0.000996s entre n = 10 e n = 200.

A ordenação de vetores já ordenados em ordem crescente resultou nos menores tempos de execução, com ganho considerável de desempenho, ainda que o algoritmo percorra todos os elementos mesmo sem realizar trocas. Isso indica a ineficiência do algoritmo, pela ausência de otimização por verificação de trocas.

Por outro lado, os vetores ordenados em ordem decrescente mostraram o pior desempenho, com n = 200, o tempo atingiu 0.001277s, representando o cenário de maior número de trocas possíveis.

A diferença crescente entre os três cenários se acentua proporcionalmente ao tamanho n, confirmando o impacto da entrada sobre a eficiência do algoritmo. Embora ainda execute

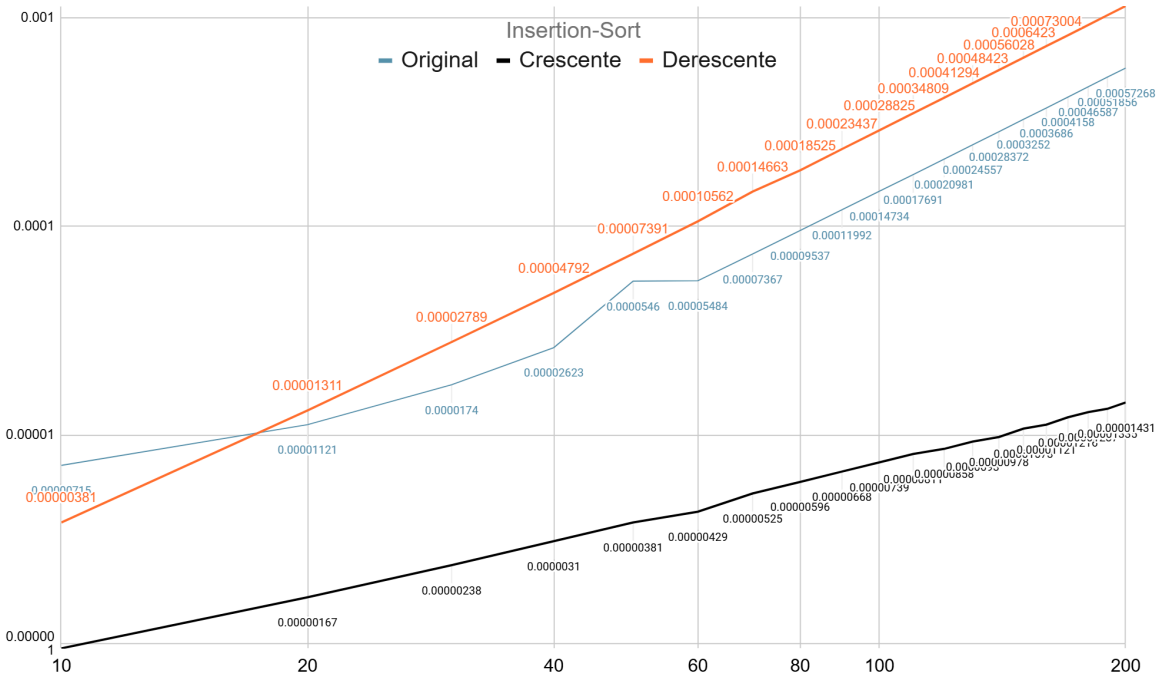
rapidamente para vetores pequenos, o Bubble Sort se mostra inviável para vetores maiores ou aplicações com restrições de desempenho.

3. Análise Insertion Sort

O vetor de entrada do algoritmo Insertion Sort utilizado foi:

['J', 'O', 'H', 'N', 'A', 'T', 'H', 'S', 'I', 'L'] .

Gráfico 2: Desempenho - Insertion Sort



O desempenho do Insertion Sort ao longo dos testes apresentou uma tendência de crescimento com o aumento do tamanho do vetor ‘n’, refletindo o comportamento natural desse algoritmo.

Nos vetores com dados em ordem aleatória, o tempo de execução aumentou gradualmente, partindo de 0.000007s para n = 10 até alcançar cerca de 0.00057s para n = 200. Esse crescimento é coerente com a complexidade média do Insertion Sort $O(n^2)$, especialmente perceptível a partir de n = 50, onde o custo de deslocamentos internos começa a impactar mais visivelmente.

No cenário ordenado em ordem crescente, que representa o melhor caso do algoritmo, os tempos permaneceram extremamente baixos em todos os tamanhos, 0.00000095s a 0.000014s, demonstrando que, nesse caso, o algoritmo realiza apenas uma comparação por elemento e nenhum deslocamento. Isso é característico do comportamento linear (O(n)) do Insertion Sort em vetores já ordenados. No entanto, dado o baixo tempo absoluto medido (na

ordem de microssegundos), qualquer variação no sistema pode causar pequenas distorções no valor real registrado.

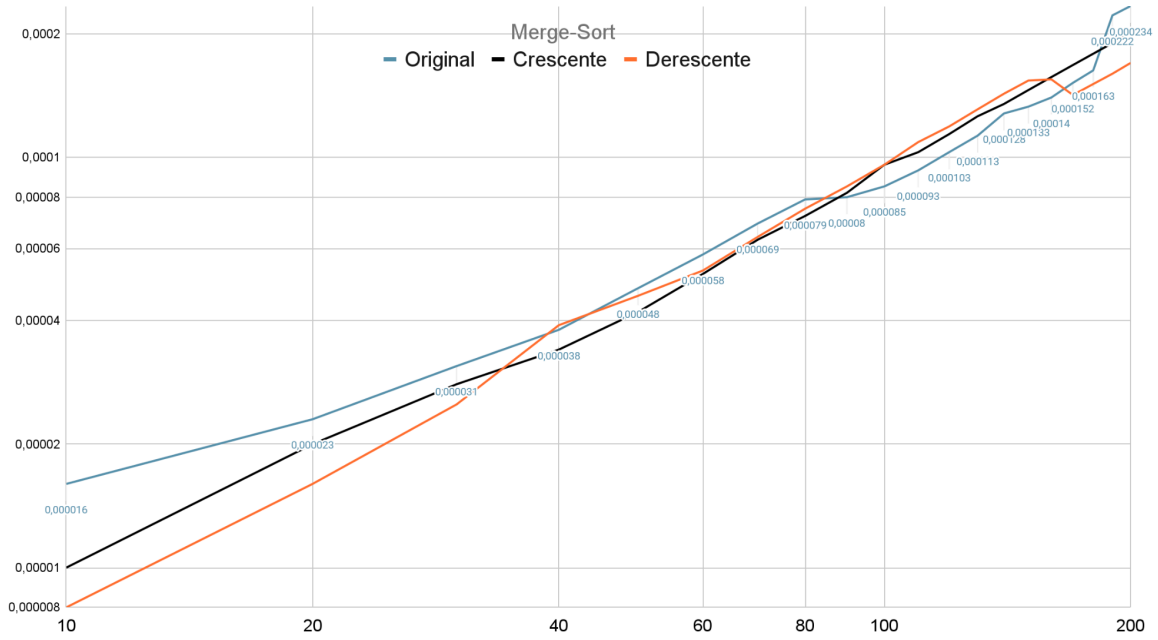
Já no cenário ordenado em ordem decrescente, que representa o pior caso, os tempos cresceram significativamente, alcançando 0.00113s em n = 200. Esse crescimento é consistente com a situação em que todos os elementos precisam ser deslocados em cada iteração , essa foi a manifestação mais direta da complexidade do algoritmo.

Em resumo, mesmo com as possíveis oscilações de tempo devido ao ambiente de execução e à natureza desses testes, os dados refletem bem o comportamento teórico do Insertion Sort, é muito eficiente em vetores pequenos ou quase ordenados, o desempenho decresce rapidamente em vetores invertidos e sensível à ordem dos dados.

4. Análise Merge Sort

Nesta análise, avaliamos o desempenho do algoritmo Merge Sort. O vetor original utilizado foi: **[50, 60, 61, 46, 85, 18, 52, 84, 85, 68].**

Gráfico 3: Desempenho - Merge Sort



O comportamento do Merge Sort nos testes realizados confirmou sua característica mais forte: a estabilidade de desempenho, independente da ordem dos dados de entrada. Isso se alinha perfeitamente à sua complexidade teórica garantida de O(n log n), tanto no melhor quanto no pior caso.

Nos vetores com dados em ordem aleatória, os tempos cresceram de forma bastante regular, com um aumento gradual e previsível. O tempo foi de 0.000016s com n = 10, chegando a 0.000234s com n = 200. Essa consistência reforça a eficiência do algoritmo mesmo para vetores de tamanho considerável, se comparado aos algoritmos como os anteriores, Bubble Sort e Insertion Sort.

No caso dos vetores ordenados em ordem crescente, o tempo de execução se manteve muito próximo ao observado no cenário aleatório, iniciando com 0.000010s e crescendo até 0.000202s. Isso demonstra que o Merge Sort não se beneficia diretamente de dados previamente organizados, já que sempre divide e funde os dados em etapas fixas, independentemente da ordem.

Já para os vetores ordenados em ordem decrescente, o comportamento também se manteve regular, com uma leve variação nos tempos (0.000008s a 0.000170s). Não houve aumento anormal nem picos de processamento, evidenciando que, mesmo no pior cenário possível para algoritmos com complexidade $O(n^2)$, o Merge Sort mantém sua complexidade controlada e eficiente.

Lembrando que, como foi citado na introdução, os testes foram realizados apenas uma vez por entrada, e em ambiente de uso real (com multitarefa, variação de temperatura, cache, etc.), pequenas oscilações nos tempos podem ter ocorrido, especialmente nos valores mais baixos (microsegundos), onde qualquer interrupção mínima do sistema pode impactar o resultado.

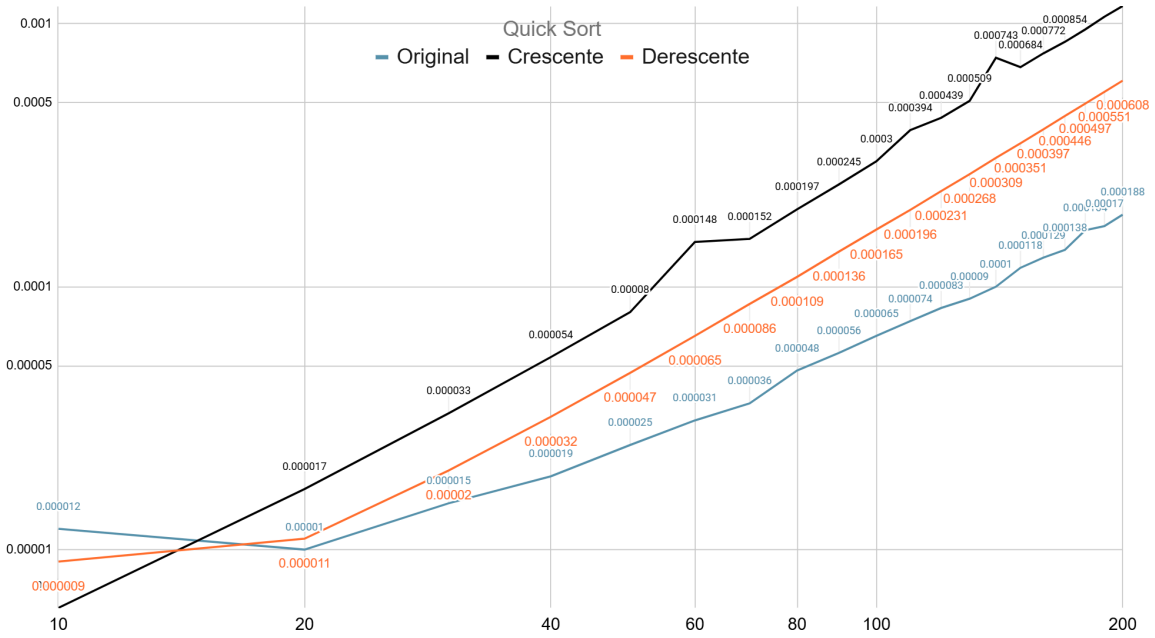
Mesmo com essas limitações, os dados obtidos validam a posição do Merge Sort como um dos algoritmos de ordenação mais confiáveis e estáveis, oferecendo excelente desempenho tanto para dados embaralhados quanto organizados.

5. Análise Quick Sort

Para o algoritmo QuickSort, o vetor original utilizado foi:

[62, 87, 22, 36, 52, 48, 96, 95, 37, 98].

Gráfico 4: Desempenho - Quick Sort



A implementação do Quicksort neste experimento utilizou o último elemento como pivô fixo, o que tem impacto direto na eficiência do algoritmo dependendo da ordem dos dados de entrada.

Nos vetores com dados em ordem aleatória, o algoritmo apresentou excelente desempenho e tempos de execução crescentes de forma suave e eficiente, de 0.000012s a 0.000188s. Esse comportamento reflete o caso médio do QuickSort, cuja complexidade esperada é O(n log n), e que se mostra bastante competitivo frente a outras abordagens de ordenação.

No cenário de vetores ordenados em ordem crescente, o uso do último elemento como pivô fixo provocou um efeito significativo no desempenho do algoritmo. À medida que o ‘n’ cresceu, os tempos aumentam muito mais rápido, saindo de 0.000006s para 0.001169s em n = 200. Isso ocorreu, porque quando o vetor já está ordenado, o último elemento é sempre o maior, levando à formação de partições altamente desequilibradas. Essa condição levou o Quick Sort a operar em seu pior caso, cuja complexidade se aproxima de $O(n^2)$. Esse comportamento evidencia uma limitação prática importante: a escolha estática e não aleatória do pivô comprometeu a performance do algoritmo em vetores já ordenados.

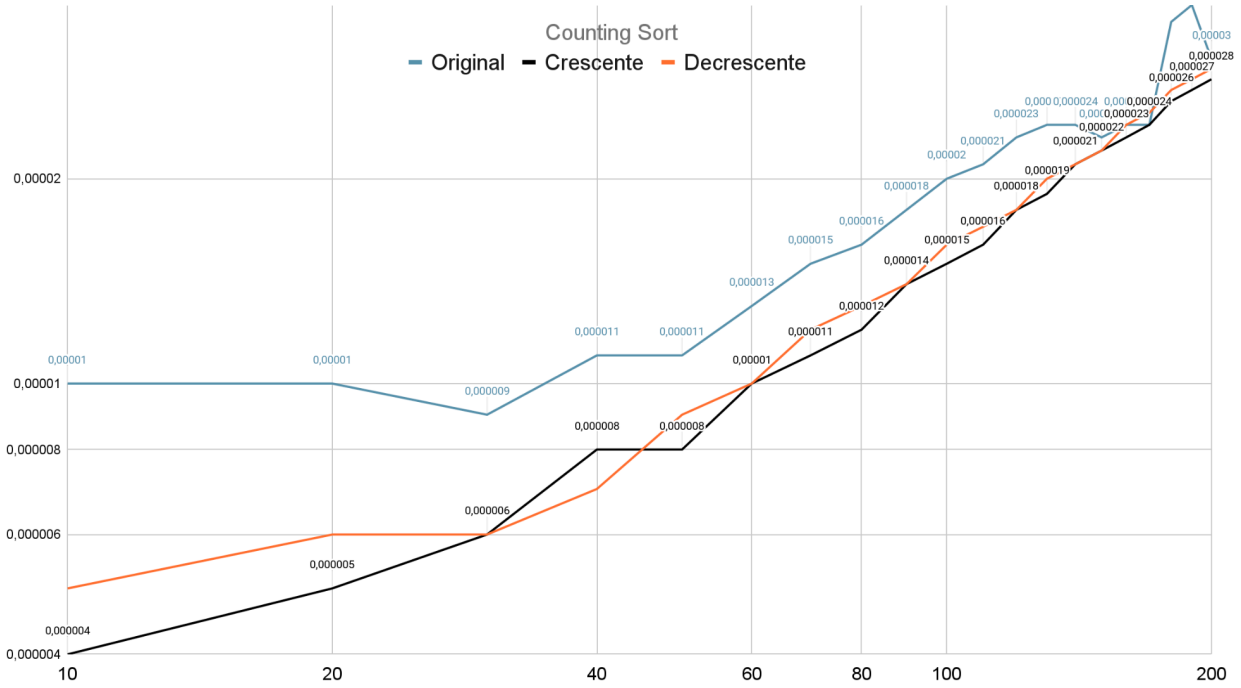
Os Vetores ordenados em ordem decrescente, surpreendentemente, pelo menos neste teste, o algoritmo teve desempenho melhor do que no caso crescente, com crescimento de tempo mais moderado: de 0.000009s para 0.000608s em n = 200. Isso indica que, apesar de também gerar partições desequilibradas, a estrutura desse vetor decrescente ofereceu divisões menos degradantes do que o cenário crescente com pivô fixo.

Esse comportamento reforça o impacto da estratégia de escolha do pivô sobre a eficiência do Quick Sort. Em outras implementações, pode-se utilizar técnicas como pivô aleatório ou mediana de três, justamente para evitar esse tipo de queda de desempenho em situações comuns como vetores já ordenados. Portanto, variações menores nos valores podem ser atribuídas a fatores externos, como processos em segundo plano e picos de uso de CPU. Apesar disso, os resultados evidenciam com clareza os pontos fortes e fracos do Quick Sort, altamente eficiente no caso médio, sensível à escolha do pivô e sujeito a queda de desempenho em vetores já ordenados.

6. Análise Counting Sort

No algoritmo Counting Sort, o vetor original utilizado foi: **[1, 1, 2, 4, 4, 0, 3, 3, 6, 5]**.

Gráfico 5: Desempenho - Counting Sort



Já o Counting Sort demonstrou um comportamento altamente eficiente e previsível, o algoritmo é especialmente vantajoso em cenários com números inteiros positivos e com faixa de valores limitada, como é o caso do vetor utilizado

O desempenho do Counting Sort em vetores aleatórios foi bastante estável, mantendo tempos muito baixos e praticamente constantes até n = 200, com valores entre 0.000009s e 0.000036s. Isso demonstra sua escalabilidade e eficiência quando operando em dados com características ideais, independentemente da ordem dos elementos.

Tanto em vetores já ordenados quanto em ordem inversa, o algoritmo apresentou tempos levemente menores em relação aos dados aleatórios, variando entre 0.000004s e 0.000028s em n = 200. No entanto, essa diferença é pouco significativa, já que o Counting Sort não depende da ordem dos dados de entrada, ele simplesmente conta a frequência de ocorrências e redistribui os valores, mantendo seu desempenho praticamente constante.

A leve vantagem nos vetores ordenados pode ser explicada por fatores externos, como variações de hardware ou pequenas diferenças de alocação de memória, uma vez que os tempos estão todos na faixa de microssegundos.

A estabilidade dos resultados confirma a vantagem do Counting Sort em sua aplicação quando o vetor contém valores inteiros não negativos e um intervalo pequeno. O tempo de execução não sofreu impactos perceptíveis da ordem dos dados, e os valores extremamente baixos mostram sua superioridade em cenários bem definidos.

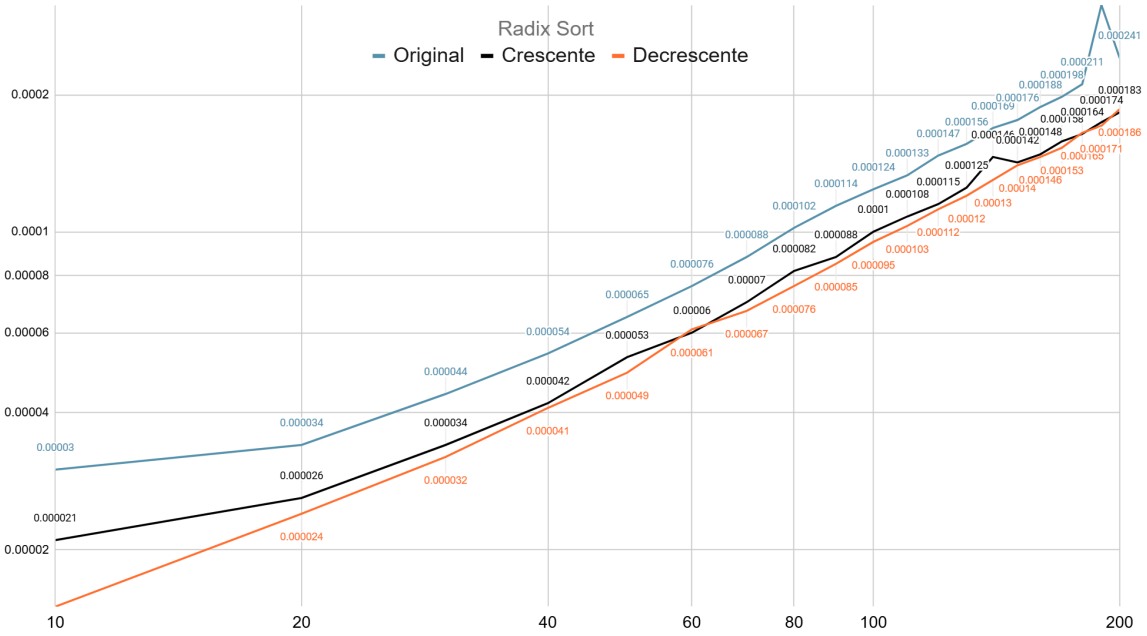
Mesmo com essas possíveis oscilações naturais do sistema, os dados coletados são consistentes e demonstram que o Counting Sort é uma das melhores opções em termos de desempenho absoluto, quando usado no contexto adequado.

7. Análise Radix Sort

Para analisar o desempenho do algoritmo Radix Sort. O vetor original utilizado foi:

[27287, 74058, 30845, 64876, 97470, 59348, 25483, 90564, 39644, 78264].

Gráfico 6: Desempenho - Radix Sort



O Radix Sort apresentou ao longo dos testes, um desempenho notavelmente estável e eficiente. Diferente dos algoritmos baseados em comparações diretas (como Quick ou Merge), o Radix Sort utiliza um processo baseado em dígitos e ordenamentos estáveis (Counting Sort) em múltiplas passagens, o que o torna adequado especialmente para números inteiros não negativos e com número de dígitos controlado, como foi o caso do vetor utilizado.

Em vetores aleatórios, com os dados embaralhados, o Radix Sort mostrou um crescimento suave e quase linear, com tempo de execução saindo de 0.000030s para 0.000240s em n = 200, com uma pequena oscilação pontual em n = 190. Esses resultados confirmam sua complexidade.

Nos casos ordenados, tanto crescente quanto decrescente, o desempenho do Radix Sort se manteve consistente e praticamente inalterado, com diferenças mínimas entre os dois casos e em relação ao vetor aleatório. Por exemplo, n = 200, crescente: 0.000183s e n = 200, decrescente: 0.000186s, isso confirma que o algoritmo não é afetado pela ordem dos elementos de entrada, pois seu funcionamento não se baseia em comparações entre elementos consecutivos, mas sim na análise de cada dígito individual.

A performance do Radix Sort foi a mais estável entre todos os algoritmos avaliados até o momento, com tempos baixos e previsíveis em todos os cenários. Embora não seja aplicável a todos os tipos de dados (como strings, floats, negativos), seu uso é muito recomendado em listas grandes de inteiros com limite de dígitos, onde pode superar até mesmo algoritmos O(n log n).

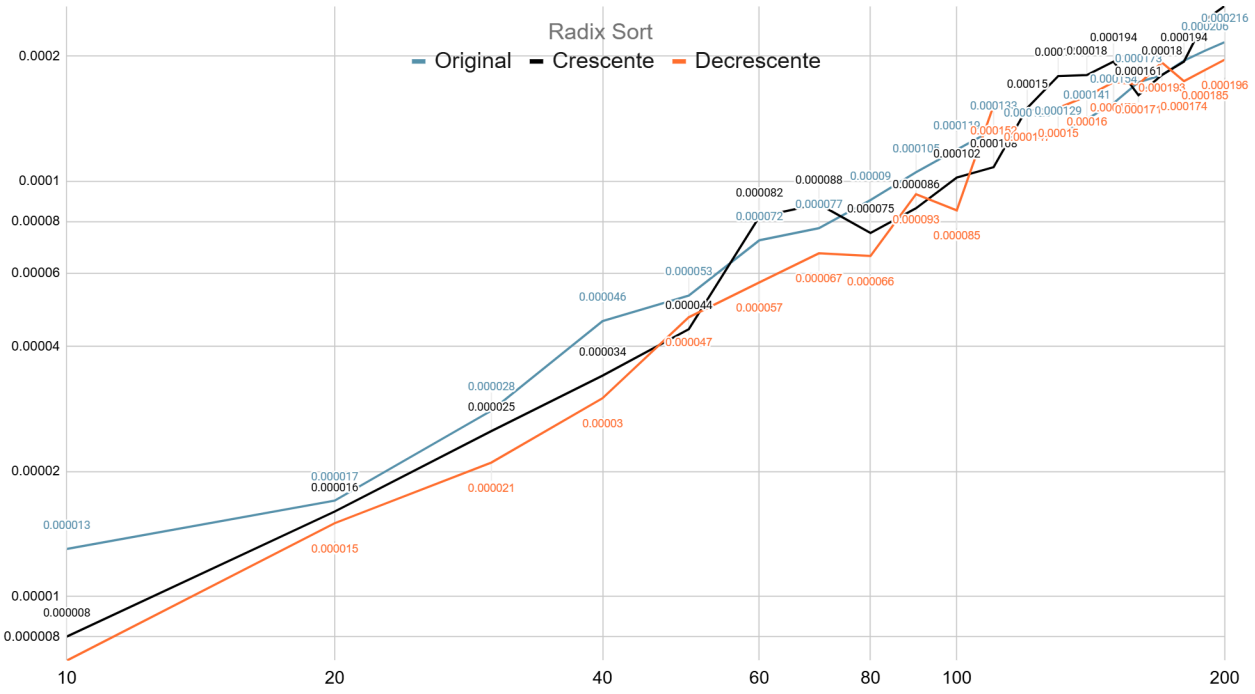
Como nos outros testes, os tempos registrados foram obtidos a partir de uma única execução por entrada, o que pode ter causado pequenas oscilações nos tempos finais, sobretudo nas medições mais curtas. Ainda assim, a regularidade dos resultados confirma a eficácia do Radix Sort na prática.

8. Análise Heap Sort

O vetor base utilizado pelo Heap sort foi:

[10, -74, -4, -38, -90, 16, 28, 65, -55, 19].

Gráfico 7 Desempenho - Heap Sort



O Heap Sort é um algoritmo de ordenação que usa uma estrutura de árvore para organizar os dados, oferece uma complexidade de tempo O(n log n) estável, independentemente da ordem dos dados de entrada. Ele não é estável no sentido de algoritmo, mas tem a vantagem de não depender da ordem dos dados e de não utilizar recursão.

Nos testes realizados, com execução única, o Heap Sort apresentou desempenho consistente e próximo de outros algoritmos, mantendo tempos estáveis e próximos em todos os cenários. Nos vetores com dados em ordem aleatória, o Heap Sort entregou resultados muito bons, começou com 0.000013s em n = 10 e chegou a 0.000216s em n = 200. O crescimento foi gradual e compatível com a complexidade O(n log n), demonstrando boa escalabilidade mesmo com aumento gradual do vetor. As pequenas oscilações nos tempos (como a leve queda em n = 130) são aceitáveis e até previsíveis, como aconteceu em outros testes e podem estar relacionadas ao gerenciamento de memória ou variações do processador durante a execução, como citado anteriormente.

Em vetores ordenados em ordem crescente, como esperado, a ordenação inicial em ordem crescente não impactou negativamente o desempenho do Heap Sort. Os tempos foram praticamente equivalentes aos do vetor aleatório, iniciando em 0.000008s e alcançando 0.000264s com n = 200. Essa estabilidade reforça uma das principais vantagens do algoritmo, ele não se beneficia nem sofre com a organização prévia dos dados. Por ser um algoritmo que constrói e reestrutura um heap completo ao longo da ordenação, a ordem dos dados não interfere na quantidade de operações necessárias

Em Vetores ordenados em ordem decrescente, o Heap Sort seguiu a mesma tendência, baixo impacto da organização dos dados na performance. Os tempos variaram de 0.000007s a 0.000196s, com pequenas oscilações que são naturais em execuções únicas e valores extremamente baixos de tempo (microsegundos).

A leve superioridade dos tempos médios, quando comparados com os vetores em ordem crescente ou aleatória, pode ser atribuída ao fato de que vetores decrescentes formam heaps com estrutura levemente mais eficientes em termos de reorganização durante a construção, ainda que isso não seja uma regra garantida.

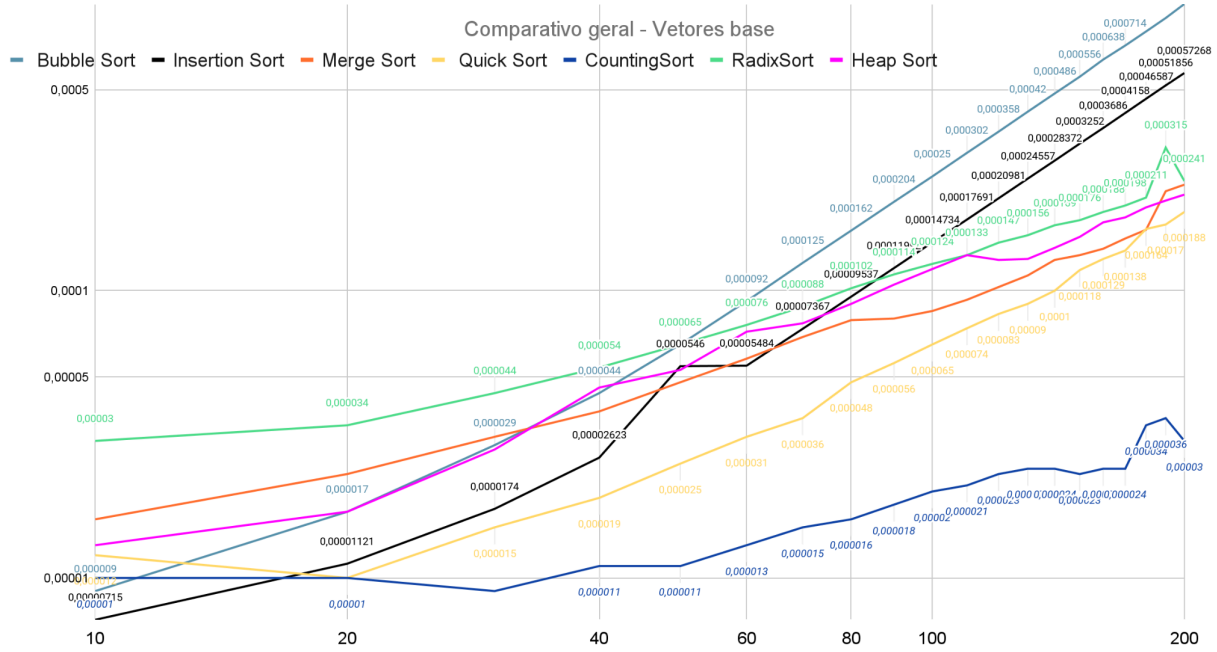
O Heap Sort é uma alternativa robusta a algoritmos como Quick Sort (que sofre em casos específicos) e Insertion/Bubble (que são quadráticos). Apesar dos tempos não serem os mais baixos entre todos os algoritmos testados, o Heap Sort oferece boa performance e comportamento estável, sendo altamente indicado para aplicações que exigem previsibilidade e evitam piores casos com grande degradação.

9. Análise comparativa geral dos algoritmos de ordenação.

9.1 Vetores base

Dessa vez, o experimento comparativo envolveu os sete algoritmos juntos utilizados neste trabalho, Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort e Heap Sort, utilizando vetores base com 10 elementos, replicados até n = 200 com incrementos de 10 em 10, cada algoritmo foi testado com a entrada original (aleatória), e os tempos de execução foram medidos individualmente, seguindo a mesma lógica anterior, mas agora, comparando o desempenho entre os algoritmos.

Gráfico 8: Desempenho geral - Vetores originais



Em termos gerais, os algoritmos com complexidade quadrática, como Bubble Sort e Insertion Sort, apresentaram os maiores tempos médios ao longo da execução, com crescimento rápido à medida que o tamanho do vetor aumentava. O Bubble Sort, como já observado, foi o mais lento em praticamente todos os casos, enquanto o Insertion Sort demonstrou desempenho um pouco superior, especialmente em tamanhos menores.

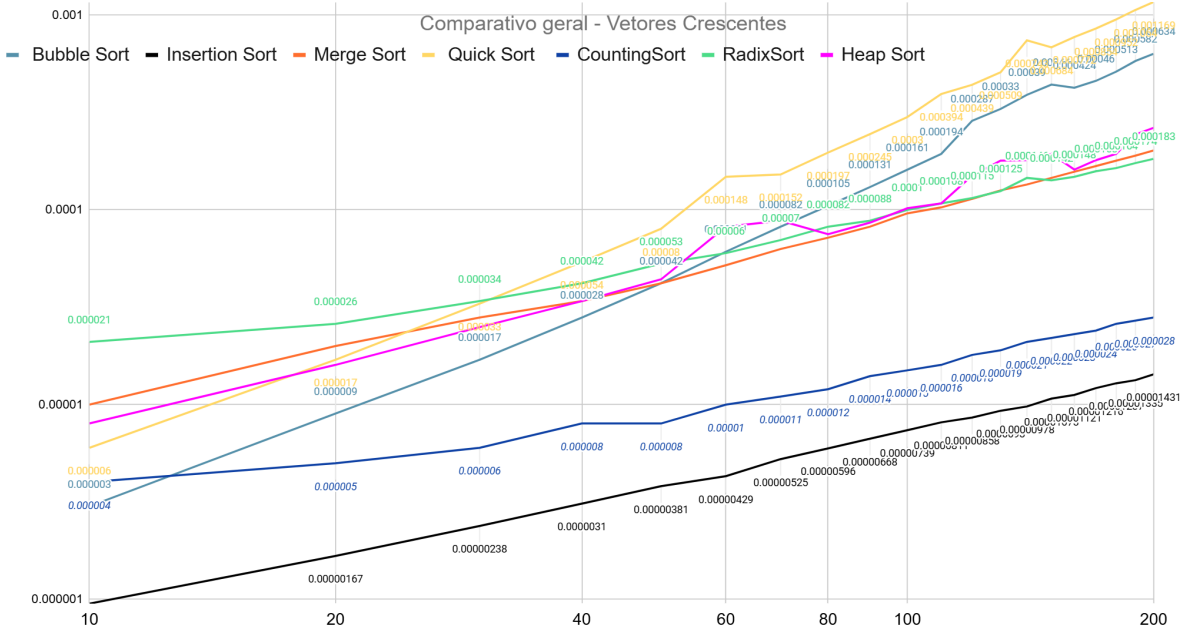
Os algoritmos com complexidade O(n log n), Merge Sort, Quick Sort e Heap Sort, foram consistentemente superiores aos anteriores. O Merge Sort se destacou pela regularidade e estabilidade de tempo, enquanto o Quick Sort, mesmo com pivô fixo, teve desempenho eficiente na entrada aleatória. O Heap Sort apresentou comportamento intermediário, tempos levemente acima do Merge, mas ainda muito melhores que os algoritmos quadráticos.

Já os algoritmos não baseados em comparação mostraram os melhores resultados absolutos, com destaque para o Counting Sort, que teve os menores tempos de execução em quase todas as instâncias. Mas, vale lembrar que isso se deve ao seu vetor base conter valores pequenos e repetidos, cenário ideal para ele. Já o Radix Sort, teve um vetor com valores significativamente maiores, mesmo assim, se manteve eficiente, demonstrando eficiência mesmo em casos menos favoráveis.

9.2 Vetores em ordem crescente.

Seguindo a estrutura já adotada, foi realizada a comparação entre os sete algoritmos de ordenação, agora com vetores previamente organizados em ordem crescente. A metodologia permaneceu a mesma da análise anterior, mostrando um comportamento de cada abordagem frente a dados já ordenados, cenário que representa o melhor caso para alguns algoritmos e o pior caso para outros.

Gráfico 9: Desempenho geral - Vetores Crescentes



Como esperado, o Insertion Sort foi o grande destaque nesse cenário, apresentando os menores tempos absolutos entre todos os algoritmos. Seu desempenho manteve-se extremamente baixo mesmo com o aumento do tamanho dos vetores, encerrando em apenas 0.00001431 segundos para n = 200, o que reforça seu excelente comportamento quando os dados já estão ordenados, já que esse é o cenário ideal pra ele, onde há um sistema em que os vetores já estão previamente ordenados, e é necessário inserir dados nele.

O Bubble Sort, embora não tão eficiente quanto o Insertion, também teve melhora considerável em relação aos vetores desordenados. Seu tempo cresceu de forma mais suave, mas ainda assim não o suficiente para superar algoritmos mais eficientes. Curiosamente, após n = 120, seus tempos apresentaram picos inesperados, o que pode estar ligado a oscilações no sistema durante as medições, uma vez que os vetores já estavam em ordem.

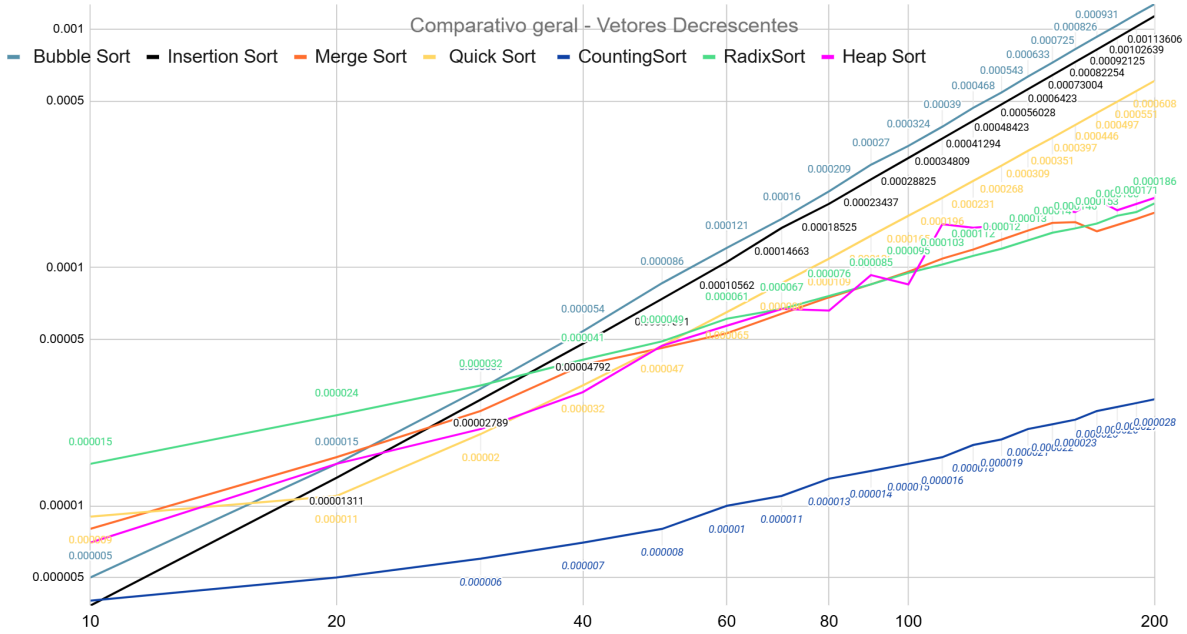
Os algoritmos com complexidade O(n log n), como Merge Sort, Heap Sort e até mesmo o Quick Sort, mantiveram comportamento relativamente estável, mas sem grandes ganhos de desempenho. O Merge e o Heap apresentaram crescimento previsível, enquanto o Quick Sort demonstrou degradação clara de desempenho, chegando a 1.169 ms em n = 200, o pior entre os três. Esse comportamento se explica pelo uso de pivô fixo (último elemento), que gera particionamentos desequilibrados nesse tipo de entrada, aproximando o algoritmo de seu pior caso $O(n^2)$.

O Counting Sort e o Radix Sort, por sua vez, continuaram entregando tempos baixos e consistentes, sem serem afetados pela ordem dos dados, como já observado anteriormente. O Counting Sort teve os menores tempos entre os algoritmos lineares, seguido de perto pelo Radix, que também manteve uma boa regularidade, mesmo operando sobre números grandes.

9.3 Vetores em ordem decrescente.

Na última análise, os algoritmos foram aplicados sobre vetores organizados em ordem decrescente, o que representa um cenário desfavorável para diversos métodos, especialmente os que dependem de comparações sequenciais para detectar ordem. A abordagem permaneceu a mesma das etapas anteriores, mas com os algoritmos ordenados em ordem decrescente.

Gráfico 10: Desempenho geral - Vetores Decrescentes



Neste contexto, os algoritmos Bubble Sort e Insertion Sort apresentaram os piores desempenhos entre todos, confirmando suas limitações em cenários de pior caso. O Insertion Sort, que havia se destacado nos vetores crescentes, sofreu forte impacto aqui, seu tempo saltou para mais de 1.1 ms em n = 200, crescendo rapidamente à medida que o tamanho do vetor aumentava. O mesmo ocorreu com o Bubble Sort, cujos tempos ultrapassaram 1.27 ms em n = 200, sendo o algoritmo mais lento neste cenário.

Por outro lado, os algoritmos de complexidade O(n log n) mantiveram seu padrão de desempenho mais estável. O Merge Sort novamente demonstrou boa consistência, com crescimento gradual e previsível, independentemente da ordem dos dados. O Heap Sort, ainda que com pequenas oscilações nos tempos, também apresentou desempenho confiável, encerrando em 0.000196 segundos em n = 200. Já o Quick Sort, mesmo com o uso de pivô fixo, se comportou bem melhor do que no caso crescente, o que sugere que a estrutura decrescente do vetor favoreceu a divisão mais equilibrada na partição, seus tempos ficaram dentro do esperado, sem degradação extrema.

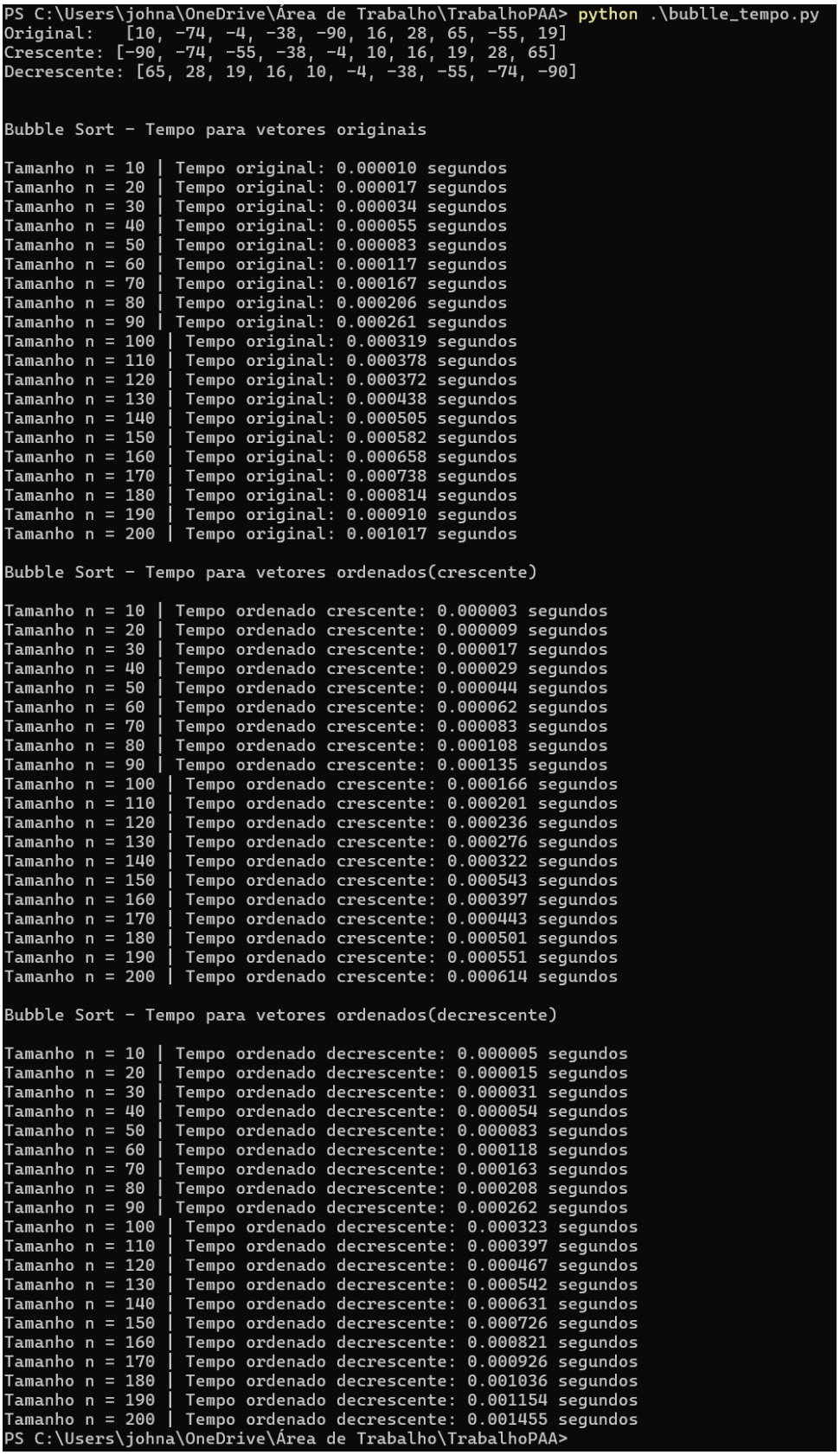
Entre os algoritmos não baseados em comparação, Counting Sort novamente se destacou, mantendo tempos consistentemente baixos, assim como o Radix Sort, que mesmo lidando com números inteiros grandes, apresentou desempenho regular. Ambos os algoritmos foram pouco impactados pela ordem dos dados, o que é coerente com suas naturezas, já que eles não dependem de comparações diretas entre elementos.

10. Conclusão

A análise dos sete algoritmos evidenciou diferenças significativas de desempenho conforme o tipo de entrada. Algoritmos quadráticos como o Bubble Sort foram os menos eficientes em todos os cenários, enquanto o Insertion Sort teve excelente desempenho com dados já ordenados. Entre os algoritmos de comparação com complexidade O(n log n), o Merge Sort demonstrou maior estabilidade, seguido pelo Heap Sort. O Quick Sort teve bom desempenho geral, mas sua sensibilidade à escolha do pivô comprometeu sua eficiência em entradas ordenadas. Já os algoritmos não comparativos, Counting Sort e Radix Sort, foram os mais rápidos, com destaque para o Counting, beneficiado pela natureza simples dos dados. De forma geral, os testes reforçam que a escolha do algoritmo ideal depende tanto da complexidade quanto das características da entrada, sendo fundamental considerar o contexto real de uso.

Anexos

Figura 1 : Resultado Bubble Sort

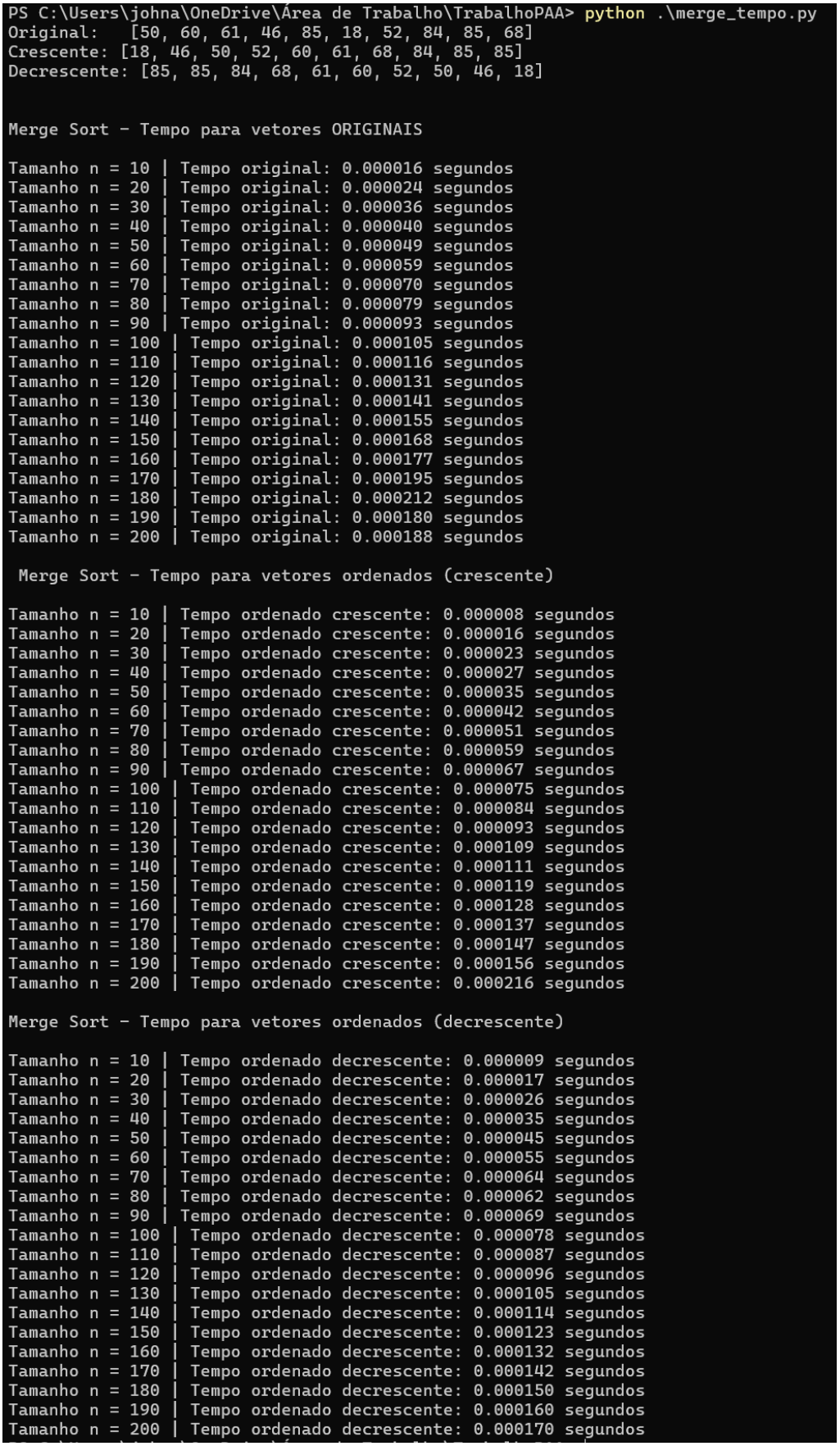


```
PS C:\Users\johna\OneDrive\Área de Trabalho\TrabalhoPAA> python .\bublle_tempo.py
Original:   [10, -74, -4, -38, -90, 16, 28, 65, -55, 19]
Crescente: [-90, -74, -55, -38, -4, 10, 16, 19, 28, 65]
Decrescente: [65, 28, 19, 16, 10, -4, -38, -55, -74, -90]


Bubble Sort - Tempo para vetores originais

Tamanho n = 10 | Tempo original: 0.000010 segundos
Tamanho n = 20 | Tempo original: 0.000017 segundos
Tamanho n = 30 | Tempo original: 0.000034 segundos
Tamanho n = 40 | Tempo original: 0.000055 segundos
Tamanho n = 50 | Tempo original: 0.000083 segundos
Tamanho n = 60 | Tempo original: 0.000117 segundos
Tamanho n = 70 | Tempo original: 0.000167 segundos
Tamanho n = 80 | Tempo original: 0.000206 segundos
Tamanho n = 90 | Tempo original: 0.000261 segundos
Tamanho n = 100 | Tempo original: 0.000319 segundos
Tamanho n = 110 | Tempo original: 0.000378 segundos
Tamanho n = 120 | Tempo original: 0.000372 segundos
Tamanho n = 130 | Tempo original: 0.000438 segundos
Tamanho n = 140 | Tempo original: 0.000505 segundos
Tamanho n = 150 | Tempo original: 0.000582 segundos
Tamanho n = 160 | Tempo original: 0.000658 segundos
Tamanho n = 170 | Tempo original: 0.000738 segundos
Tamanho n = 180 | Tempo original: 0.000814 segundos
Tamanho n = 190 | Tempo original: 0.000910 segundos
Tamanho n = 200 | Tempo original: 0.001017 segundos

Bubble Sort - Tempo para vetores ordenados(crescente)

Tamanho n = 10 | Tempo ordenado crescente: 0.000003 segundos
Tamanho n = 20 | Tempo ordenado crescente: 0.000009 segundos
Tamanho n = 30 | Tempo ordenado crescente: 0.000017 segundos
Tamanho n = 40 | Tempo ordenado crescente: 0.000029 segundos
Tamanho n = 50 | Tempo ordenado crescente: 0.000044 segundos
Tamanho n = 60 | Tempo ordenado crescente: 0.000062 segundos
Tamanho n = 70 | Tempo ordenado crescente: 0.000083 segundos
Tamanho n = 80 | Tempo ordenado crescente: 0.000108 segundos
Tamanho n = 90 | Tempo ordenado crescente: 0.000135 segundos
Tamanho n = 100 | Tempo ordenado crescente: 0.000166 segundos
Tamanho n = 110 | Tempo ordenado crescente: 0.000201 segundos
Tamanho n = 120 | Tempo ordenado crescente: 0.000236 segundos
Tamanho n = 130 | Tempo ordenado crescente: 0.000276 segundos
Tamanho n = 140 | Tempo ordenado crescente: 0.000322 segundos
Tamanho n = 150 | Tempo ordenado crescente: 0.000543 segundos
Tamanho n = 160 | Tempo ordenado crescente: 0.000397 segundos
Tamanho n = 170 | Tempo ordenado crescente: 0.000443 segundos
Tamanho n = 180 | Tempo ordenado crescente: 0.000501 segundos
Tamanho n = 190 | Tempo ordenado crescente: 0.000551 segundos
Tamanho n = 200 | Tempo ordenado crescente: 0.000614 segundos

Bubble Sort - Tempo para vetores ordenados(decrescente)

Tamanho n = 10 | Tempo ordenado decrescente: 0.000005 segundos
Tamanho n = 20 | Tempo ordenado decrescente: 0.000015 segundos
Tamanho n = 30 | Tempo ordenado decrescente: 0.000031 segundos
Tamanho n = 40 | Tempo ordenado decrescente: 0.000054 segundos
Tamanho n = 50 | Tempo ordenado decrescente: 0.000083 segundos
Tamanho n = 60 | Tempo ordenado decrescente: 0.000118 segundos
Tamanho n = 70 | Tempo ordenado decrescente: 0.000163 segundos
Tamanho n = 80 | Tempo ordenado decrescente: 0.000208 segundos
Tamanho n = 90 | Tempo ordenado decrescente: 0.000262 segundos
Tamanho n = 100 | Tempo ordenado decrescente: 0.000323 segundos
Tamanho n = 110 | Tempo ordenado decrescente: 0.000397 segundos
Tamanho n = 120 | Tempo ordenado decrescente: 0.000467 segundos
Tamanho n = 130 | Tempo ordenado decrescente: 0.000542 segundos
Tamanho n = 140 | Tempo ordenado decrescente: 0.000631 segundos
Tamanho n = 150 | Tempo ordenado decrescente: 0.000726 segundos
Tamanho n = 160 | Tempo ordenado decrescente: 0.000821 segundos
Tamanho n = 170 | Tempo ordenado decrescente: 0.000926 segundos
Tamanho n = 180 | Tempo ordenado decrescente: 0.001036 segundos
Tamanho n = 190 | Tempo ordenado decrescente: 0.001154 segundos
Tamanho n = 200 | Tempo ordenado decrescente: 0.001455 segundos
PS C:\Users\johna\OneDrive\Área de Trabalho\TrabalhoPAA>
```

Figura 2 : Resultado Insertion Sort

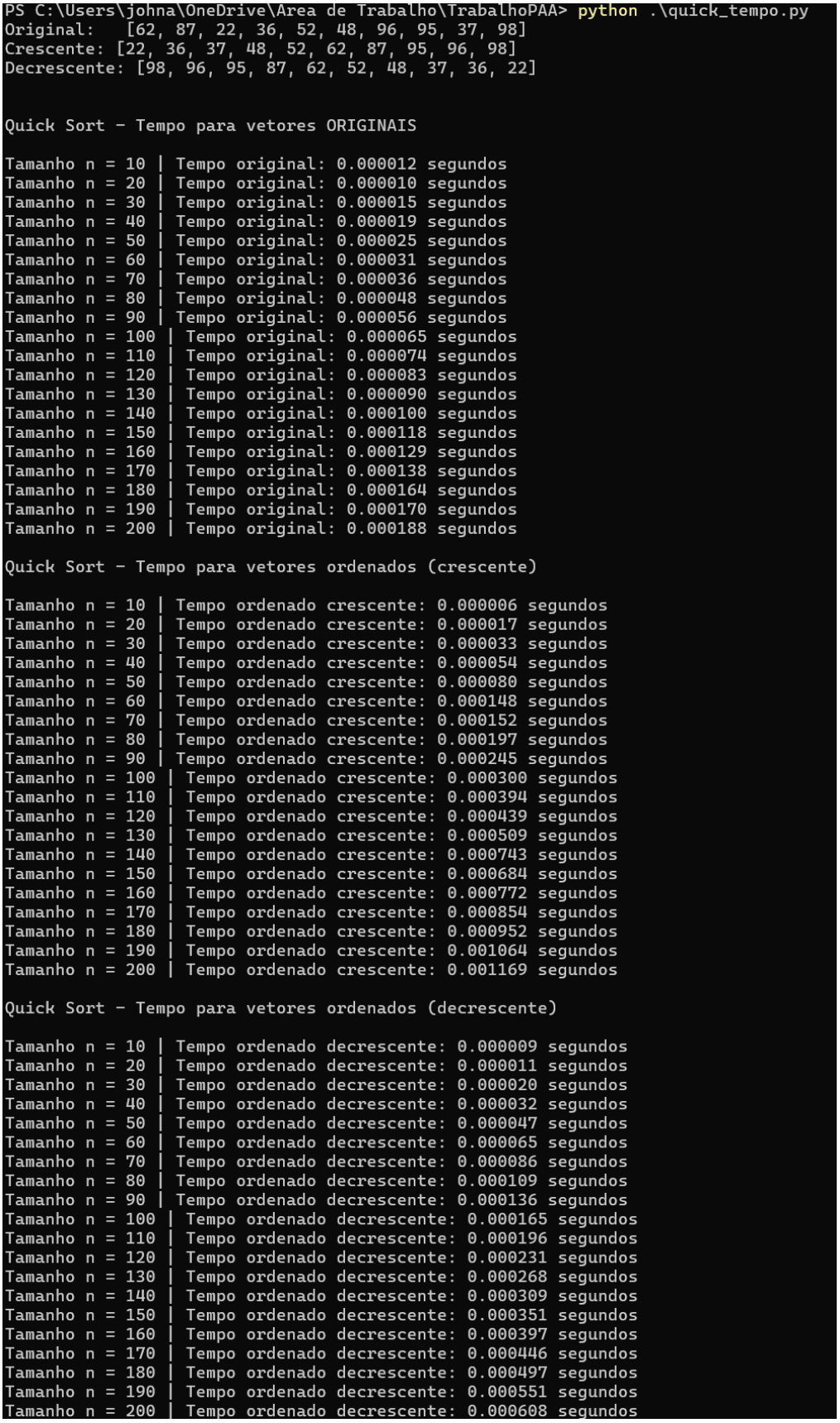
```
PS C:\Users\johna\OneDrive\Área de Trabalho\TrabalhoPAA> python .\insertionSort_tempo.p
y
Original:   ['J', 'O', 'H', 'N', 'A', 'T', 'H', 'S', 'I', 'L']
Crescente: ['A', 'H', 'H', 'I', 'J', 'L', 'N', 'O', 'S', 'T']
Decrescente: ['T', 'S', 'O', 'N', 'L', 'J', 'I', 'H', 'H', 'A']


Insertion Sort - Tempo para vetores ORIGINAIS (caracteres aleatórios)

Tamanho n = 10 | Tempo original: 0.00000572 segundos
Tamanho n = 20 | Tempo original: 0.00000882 segundos
Tamanho n = 30 | Tempo original: 0.00001550 segundos
Tamanho n = 40 | Tempo original: 0.00002575 segundos
Tamanho n = 50 | Tempo original: 0.00003910 segundos
Tamanho n = 60 | Tempo original: 0.00005531 segundos
Tamanho n = 70 | Tempo original: 0.00007486 segundos
Tamanho n = 80 | Tempo original: 0.00009680 segundos
Tamanho n = 90 | Tempo original: 0.00012159 segundos
Tamanho n = 100 | Tempo original: 0.00014949 segundos
Tamanho n = 110 | Tempo original: 0.00017953 segundos
Tamanho n = 120 | Tempo original: 0.00021291 segundos
Tamanho n = 130 | Tempo original: 0.00024891 segundos
Tamanho n = 140 | Tempo original: 0.00028825 segundos
Tamanho n = 150 | Tempo original: 0.00032854 segundos
Tamanho n = 160 | Tempo original: 0.00037289 segundos
Tamanho n = 170 | Tempo original: 0.00042009 segundos
Tamanho n = 180 | Tempo original: 0.00047016 segundos
Tamanho n = 190 | Tempo original: 0.00052309 segundos
Tamanho n = 200 | Tempo original: 0.00057840 segundos

Insertion Sort - Tempo para vetores ordenado (crescente)

Tamanho n = 10 | Tempo ordenado crescente: 0.00000095 segundos
Tamanho n = 20 | Tempo ordenado crescente: 0.00000191 segundos
Tamanho n = 30 | Tempo ordenado crescente: 0.00000262 segundos
Tamanho n = 40 | Tempo ordenado crescente: 0.00000334 segundos
Tamanho n = 50 | Tempo ordenado crescente: 0.00000644 segundos
Tamanho n = 60 | Tempo ordenado crescente: 0.00000477 segundos
Tamanho n = 70 | Tempo ordenado crescente: 0.00000525 segundos
Tamanho n = 80 | Tempo ordenado crescente: 0.00000596 segundos
Tamanho n = 90 | Tempo ordenado crescente: 0.00000668 segundos
Tamanho n = 100 | Tempo ordenado crescente: 0.00000739 segundos
Tamanho n = 110 | Tempo ordenado crescente: 0.00000811 segundos
Tamanho n = 120 | Tempo ordenado crescente: 0.00000882 segundos
Tamanho n = 130 | Tempo ordenado crescente: 0.00000954 segundos
Tamanho n = 140 | Tempo ordenado crescente: 0.00001001 segundos
Tamanho n = 150 | Tempo ordenado crescente: 0.00001097 segundos
Tamanho n = 160 | Tempo ordenado crescente: 0.00001144 segundos
Tamanho n = 170 | Tempo ordenado crescente: 0.00001216 segundos
Tamanho n = 180 | Tempo ordenado crescente: 0.00001287 segundos
Tamanho n = 190 | Tempo ordenado crescente: 0.00001359 segundos
Tamanho n = 200 | Tempo ordenado crescente: 0.00001431 segundos

Insertion Sort - Tempo para vetores ordenado (decrescente)

Tamanho n = 10 | Tempo ordenado decrescente: 0.00000405 segundos
Tamanho n = 20 | Tempo ordenado decrescente: 0.00001335 segundos
Tamanho n = 30 | Tempo ordenado decrescente: 0.00002813 segundos
Tamanho n = 40 | Tempo ordenado decrescente: 0.00004864 segundos
Tamanho n = 50 | Tempo ordenado decrescente: 0.00007510 segundos
Tamanho n = 60 | Tempo ordenado decrescente: 0.00010705 segundos
Tamanho n = 70 | Tempo ordenado decrescente: 0.00014448 segundos
Tamanho n = 80 | Tempo ordenado decrescente: 0.00018764 segundos
Tamanho n = 90 | Tempo ordenado decrescente: 0.00023651 segundos
Tamanho n = 100 | Tempo ordenado decrescente: 0.00029182 segundos
Tamanho n = 110 | Tempo ordenado decrescente: 0.00035524 segundos
Tamanho n = 120 | Tempo ordenado decrescente: 0.00041747 segundos
Tamanho n = 130 | Tempo ordenado decrescente: 0.00048947 segundos
Tamanho n = 140 | Tempo ordenado decrescente: 0.00056577 segundos
Tamanho n = 150 | Tempo ordenado decrescente: 0.00065160 segundos
Tamanho n = 160 | Tempo ordenado decrescente: 0.00073934 segundos
Tamanho n = 170 | Tempo ordenado decrescente: 0.00083089 segundos
Tamanho n = 180 | Tempo ordenado decrescente: 0.00093102 segundos
Tamanho n = 190 | Tempo ordenado decrescente: 0.00103617 segundos
Tamanho n = 200 | Tempo ordenado decrescente: 0.00114679 segundos
```

Figura 3 : Resultado Merge Sort



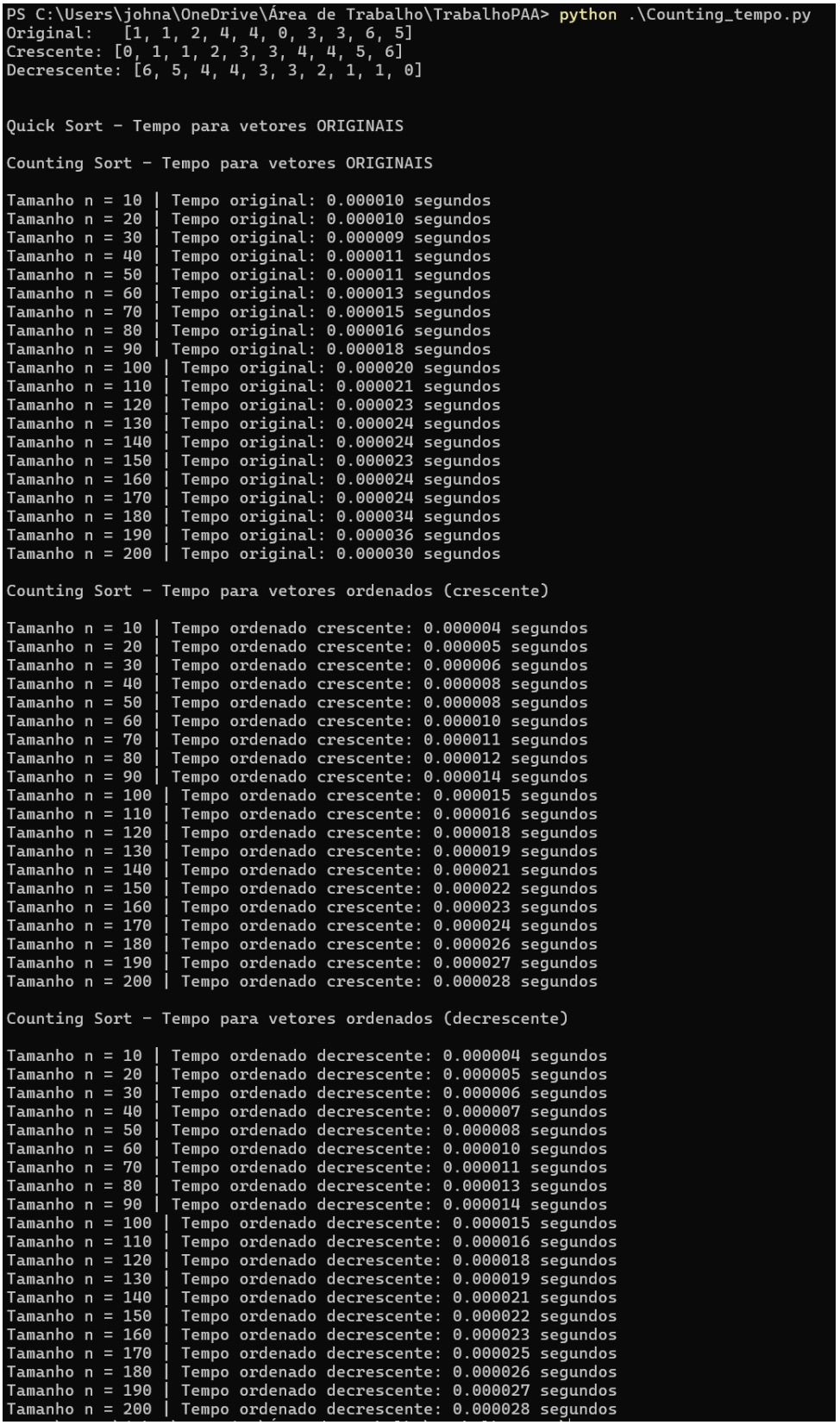
```
PS C:\Users\johna\OneDrive\Área de Trabalho\TrabalhoPAA> python .\merge_tempo.py
Original:   [50, 60, 61, 46, 85, 18, 52, 84, 85, 68]
Crescente: [18, 46, 50, 52, 60, 61, 68, 84, 85, 85]
Decrescente: [85, 85, 84, 68, 61, 60, 52, 50, 46, 18]


Merge Sort - Tempo para vetores ORIGINAIS

Tamanho n = 10 | Tempo original: 0.000016 segundos
Tamanho n = 20 | Tempo original: 0.000024 segundos
Tamanho n = 30 | Tempo original: 0.000036 segundos
Tamanho n = 40 | Tempo original: 0.000040 segundos
Tamanho n = 50 | Tempo original: 0.000049 segundos
Tamanho n = 60 | Tempo original: 0.000059 segundos
Tamanho n = 70 | Tempo original: 0.000070 segundos
Tamanho n = 80 | Tempo original: 0.000079 segundos
Tamanho n = 90 | Tempo original: 0.000093 segundos
Tamanho n = 100 | Tempo original: 0.000105 segundos
Tamanho n = 110 | Tempo original: 0.000116 segundos
Tamanho n = 120 | Tempo original: 0.000131 segundos
Tamanho n = 130 | Tempo original: 0.000141 segundos
Tamanho n = 140 | Tempo original: 0.000155 segundos
Tamanho n = 150 | Tempo original: 0.000168 segundos
Tamanho n = 160 | Tempo original: 0.000177 segundos
Tamanho n = 170 | Tempo original: 0.000195 segundos
Tamanho n = 180 | Tempo original: 0.000212 segundos
Tamanho n = 190 | Tempo original: 0.000180 segundos
Tamanho n = 200 | Tempo original: 0.000188 segundos

 Merge Sort - Tempo para vetores ordenados (crescente)

Tamanho n = 10 | Tempo ordenado crescente: 0.000008 segundos
Tamanho n = 20 | Tempo ordenado crescente: 0.000016 segundos
Tamanho n = 30 | Tempo ordenado crescente: 0.000023 segundos
Tamanho n = 40 | Tempo ordenado crescente: 0.000027 segundos
Tamanho n = 50 | Tempo ordenado crescente: 0.000035 segundos
Tamanho n = 60 | Tempo ordenado crescente: 0.000042 segundos
Tamanho n = 70 | Tempo ordenado crescente: 0.000051 segundos
Tamanho n = 80 | Tempo ordenado crescente: 0.000059 segundos
Tamanho n = 90 | Tempo ordenado crescente: 0.000067 segundos
Tamanho n = 100 | Tempo ordenado crescente: 0.000075 segundos
Tamanho n = 110 | Tempo ordenado crescente: 0.000084 segundos
Tamanho n = 120 | Tempo ordenado crescente: 0.000093 segundos
Tamanho n = 130 | Tempo ordenado crescente: 0.000109 segundos
Tamanho n = 140 | Tempo ordenado crescente: 0.000111 segundos
Tamanho n = 150 | Tempo ordenado crescente: 0.000119 segundos
Tamanho n = 160 | Tempo ordenado crescente: 0.000128 segundos
Tamanho n = 170 | Tempo ordenado crescente: 0.000137 segundos
Tamanho n = 180 | Tempo ordenado crescente: 0.000147 segundos
Tamanho n = 190 | Tempo ordenado crescente: 0.000156 segundos
Tamanho n = 200 | Tempo ordenado crescente: 0.000216 segundos

Merge Sort - Tempo para vetores ordenados (decrescente)

Tamanho n = 10 | Tempo ordenado decrescente: 0.000009 segundos
Tamanho n = 20 | Tempo ordenado decrescente: 0.000017 segundos
Tamanho n = 30 | Tempo ordenado decrescente: 0.000026 segundos
Tamanho n = 40 | Tempo ordenado decrescente: 0.000035 segundos
Tamanho n = 50 | Tempo ordenado decrescente: 0.000045 segundos
Tamanho n = 60 | Tempo ordenado decrescente: 0.000055 segundos
Tamanho n = 70 | Tempo ordenado decrescente: 0.000064 segundos
Tamanho n = 80 | Tempo ordenado decrescente: 0.000062 segundos
Tamanho n = 90 | Tempo ordenado decrescente: 0.000069 segundos
Tamanho n = 100 | Tempo ordenado decrescente: 0.000078 segundos
Tamanho n = 110 | Tempo ordenado decrescente: 0.000087 segundos
Tamanho n = 120 | Tempo ordenado decrescente: 0.000096 segundos
Tamanho n = 130 | Tempo ordenado decrescente: 0.000105 segundos
Tamanho n = 140 | Tempo ordenado decrescente: 0.000114 segundos
Tamanho n = 150 | Tempo ordenado decrescente: 0.000123 segundos
Tamanho n = 160 | Tempo ordenado decrescente: 0.000132 segundos
Tamanho n = 170 | Tempo ordenado decrescente: 0.000142 segundos
Tamanho n = 180 | Tempo ordenado decrescente: 0.000150 segundos
Tamanho n = 190 | Tempo ordenado decrescente: 0.000160 segundos
Tamanho n = 200 | Tempo ordenado decrescente: 0.000170 segundos
```

Figura 4 : Resultado Quick Sort



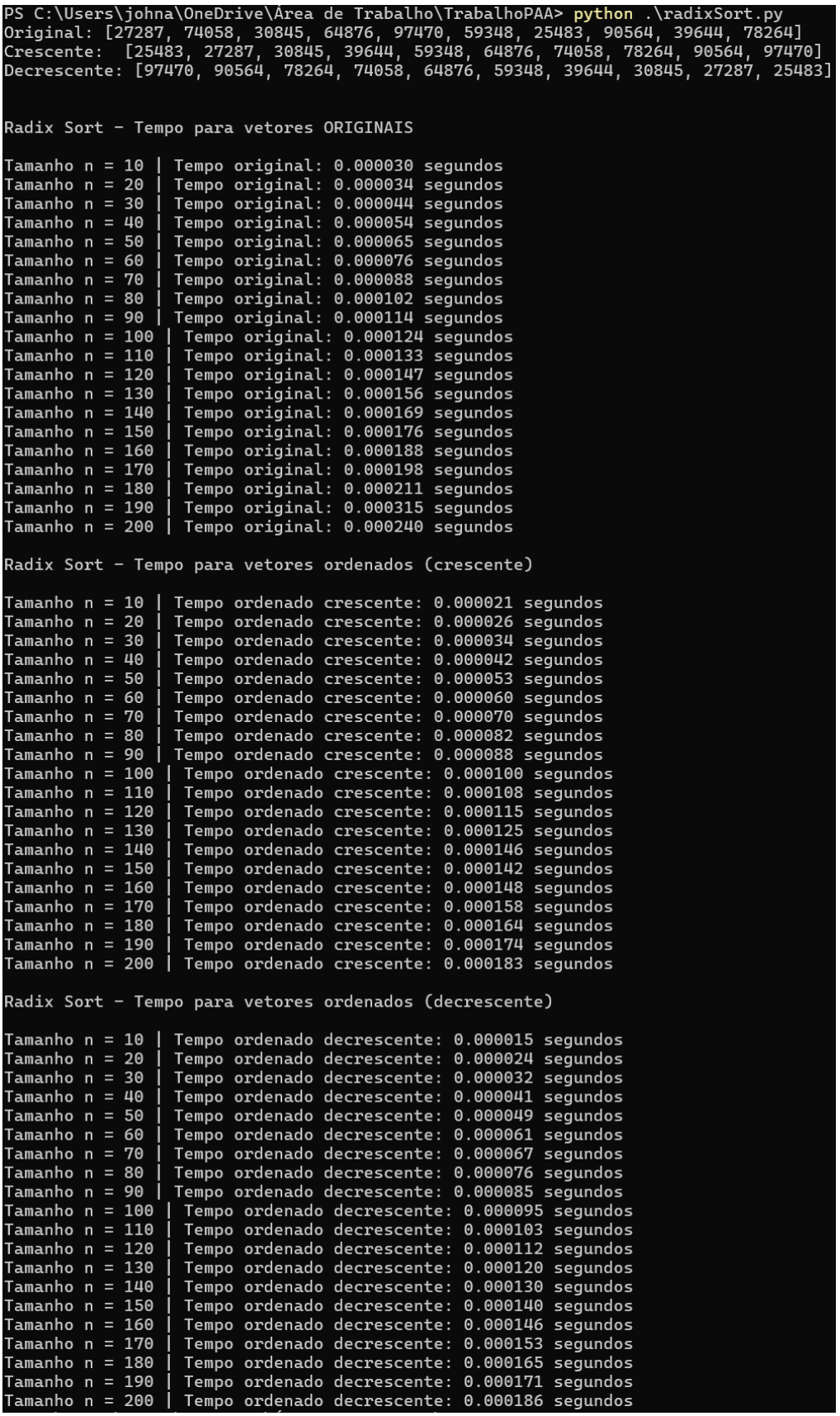
```
PS C:\Users\johna\OneDrive\Area de Trabalho\TrabalhoPAA> python .\quick_tempo.py
Original:   [62, 87, 22, 36, 52, 48, 96, 95, 37, 98]
Crescente: [22, 36, 37, 48, 52, 62, 87, 95, 96, 98]
Decrescente: [98, 96, 95, 87, 62, 52, 48, 37, 36, 22]


Quick Sort - Tempo para vetores ORIGINAIS

Tamanho n = 10 | Tempo original: 0.000012 segundos
Tamanho n = 20 | Tempo original: 0.000010 segundos
Tamanho n = 30 | Tempo original: 0.000015 segundos
Tamanho n = 40 | Tempo original: 0.000019 segundos
Tamanho n = 50 | Tempo original: 0.000025 segundos
Tamanho n = 60 | Tempo original: 0.000031 segundos
Tamanho n = 70 | Tempo original: 0.000036 segundos
Tamanho n = 80 | Tempo original: 0.000048 segundos
Tamanho n = 90 | Tempo original: 0.000056 segundos
Tamanho n = 100 | Tempo original: 0.000065 segundos
Tamanho n = 110 | Tempo original: 0.000074 segundos
Tamanho n = 120 | Tempo original: 0.000083 segundos
Tamanho n = 130 | Tempo original: 0.000090 segundos
Tamanho n = 140 | Tempo original: 0.000100 segundos
Tamanho n = 150 | Tempo original: 0.000118 segundos
Tamanho n = 160 | Tempo original: 0.000129 segundos
Tamanho n = 170 | Tempo original: 0.000138 segundos
Tamanho n = 180 | Tempo original: 0.000164 segundos
Tamanho n = 190 | Tempo original: 0.000170 segundos
Tamanho n = 200 | Tempo original: 0.000188 segundos

Quick Sort - Tempo para vetores ordenados (crescente)

Tamanho n = 10 | Tempo ordenado crescente: 0.000006 segundos
Tamanho n = 20 | Tempo ordenado crescente: 0.000017 segundos
Tamanho n = 30 | Tempo ordenado crescente: 0.000033 segundos
Tamanho n = 40 | Tempo ordenado crescente: 0.000054 segundos
Tamanho n = 50 | Tempo ordenado crescente: 0.000080 segundos
Tamanho n = 60 | Tempo ordenado crescente: 0.000148 segundos
Tamanho n = 70 | Tempo ordenado crescente: 0.000152 segundos
Tamanho n = 80 | Tempo ordenado crescente: 0.000197 segundos
Tamanho n = 90 | Tempo ordenado crescente: 0.000245 segundos
Tamanho n = 100 | Tempo ordenado crescente: 0.000300 segundos
Tamanho n = 110 | Tempo ordenado crescente: 0.000394 segundos
Tamanho n = 120 | Tempo ordenado crescente: 0.000439 segundos
Tamanho n = 130 | Tempo ordenado crescente: 0.000509 segundos
Tamanho n = 140 | Tempo ordenado crescente: 0.000743 segundos
Tamanho n = 150 | Tempo ordenado crescente: 0.000684 segundos
Tamanho n = 160 | Tempo ordenado crescente: 0.000772 segundos
Tamanho n = 170 | Tempo ordenado crescente: 0.000854 segundos
Tamanho n = 180 | Tempo ordenado crescente: 0.000952 segundos
Tamanho n = 190 | Tempo ordenado crescente: 0.001064 segundos
Tamanho n = 200 | Tempo ordenado crescente: 0.001169 segundos

Quick Sort - Tempo para vetores ordenados (decrescente)

Tamanho n = 10 | Tempo ordenado decrescente: 0.000009 segundos
Tamanho n = 20 | Tempo ordenado decrescente: 0.000011 segundos
Tamanho n = 30 | Tempo ordenado decrescente: 0.000020 segundos
Tamanho n = 40 | Tempo ordenado decrescente: 0.000032 segundos
Tamanho n = 50 | Tempo ordenado decrescente: 0.000047 segundos
Tamanho n = 60 | Tempo ordenado decrescente: 0.000065 segundos
Tamanho n = 70 | Tempo ordenado decrescente: 0.000086 segundos
Tamanho n = 80 | Tempo ordenado decrescente: 0.000109 segundos
Tamanho n = 90 | Tempo ordenado decrescente: 0.000136 segundos
Tamanho n = 100 | Tempo ordenado decrescente: 0.000165 segundos
Tamanho n = 110 | Tempo ordenado decrescente: 0.000196 segundos
Tamanho n = 120 | Tempo ordenado decrescente: 0.000231 segundos
Tamanho n = 130 | Tempo ordenado decrescente: 0.000268 segundos
Tamanho n = 140 | Tempo ordenado decrescente: 0.000309 segundos
Tamanho n = 150 | Tempo ordenado decrescente: 0.000351 segundos
Tamanho n = 160 | Tempo ordenado decrescente: 0.000397 segundos
Tamanho n = 170 | Tempo ordenado decrescente: 0.000446 segundos
Tamanho n = 180 | Tempo ordenado decrescente: 0.000497 segundos
Tamanho n = 190 | Tempo ordenado decrescente: 0.000551 segundos
Tamanho n = 200 | Tempo ordenado decrescente: 0.000608 segundos
```

Figura 5: Resultado Counting Sort



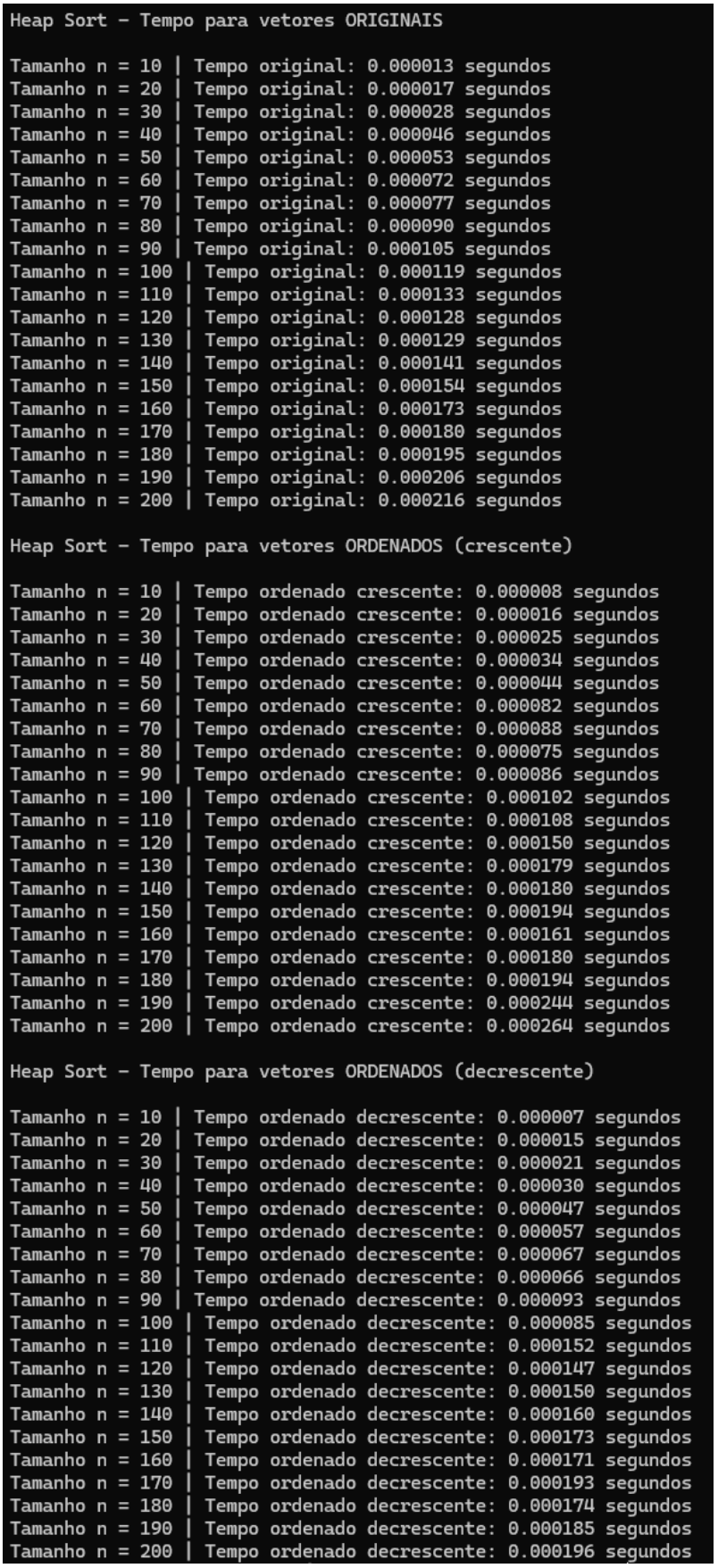
```
PS C:\Users\johna\OneDrive\Área de Trabalho\TrabalhoPAA> python .\Counting_tempo.py
Original:   [1, 1, 2, 4, 4, 0, 3, 3, 6, 5]
Crescente: [0, 1, 1, 2, 3, 3, 4, 4, 5, 6]
Decrescente: [6, 5, 4, 4, 3, 3, 2, 1, 1, 0]


Quick Sort - Tempo para vetores ORIGINAIS

Counting Sort - Tempo para vetores ORIGINAIS

Tamanho n = 10 | Tempo original: 0.000010 segundos
Tamanho n = 20 | Tempo original: 0.000010 segundos
Tamanho n = 30 | Tempo original: 0.000009 segundos
Tamanho n = 40 | Tempo original: 0.000011 segundos
Tamanho n = 50 | Tempo original: 0.000011 segundos
Tamanho n = 60 | Tempo original: 0.000013 segundos
Tamanho n = 70 | Tempo original: 0.000015 segundos
Tamanho n = 80 | Tempo original: 0.000016 segundos
Tamanho n = 90 | Tempo original: 0.000018 segundos
Tamanho n = 100 | Tempo original: 0.000020 segundos
Tamanho n = 110 | Tempo original: 0.000021 segundos
Tamanho n = 120 | Tempo original: 0.000023 segundos
Tamanho n = 130 | Tempo original: 0.000024 segundos
Tamanho n = 140 | Tempo original: 0.000024 segundos
Tamanho n = 150 | Tempo original: 0.000023 segundos
Tamanho n = 160 | Tempo original: 0.000024 segundos
Tamanho n = 170 | Tempo original: 0.000024 segundos
Tamanho n = 180 | Tempo original: 0.000034 segundos
Tamanho n = 190 | Tempo original: 0.000036 segundos
Tamanho n = 200 | Tempo original: 0.000030 segundos

Counting Sort - Tempo para vetores ordenados (crescente)

Tamanho n = 10 | Tempo ordenado crescente: 0.000004 segundos
Tamanho n = 20 | Tempo ordenado crescente: 0.000005 segundos
Tamanho n = 30 | Tempo ordenado crescente: 0.000006 segundos
Tamanho n = 40 | Tempo ordenado crescente: 0.000008 segundos
Tamanho n = 50 | Tempo ordenado crescente: 0.000008 segundos
Tamanho n = 60 | Tempo ordenado crescente: 0.000010 segundos
Tamanho n = 70 | Tempo ordenado crescente: 0.000011 segundos
Tamanho n = 80 | Tempo ordenado crescente: 0.000012 segundos
Tamanho n = 90 | Tempo ordenado crescente: 0.000014 segundos
Tamanho n = 100 | Tempo ordenado crescente: 0.000015 segundos
Tamanho n = 110 | Tempo ordenado crescente: 0.000016 segundos
Tamanho n = 120 | Tempo ordenado crescente: 0.000018 segundos
Tamanho n = 130 | Tempo ordenado crescente: 0.000019 segundos
Tamanho n = 140 | Tempo ordenado crescente: 0.000021 segundos
Tamanho n = 150 | Tempo ordenado crescente: 0.000022 segundos
Tamanho n = 160 | Tempo ordenado crescente: 0.000023 segundos
Tamanho n = 170 | Tempo ordenado crescente: 0.000024 segundos
Tamanho n = 180 | Tempo ordenado crescente: 0.000026 segundos
Tamanho n = 190 | Tempo ordenado crescente: 0.000027 segundos
Tamanho n = 200 | Tempo ordenado crescente: 0.000028 segundos

Counting Sort - Tempo para vetores ordenados (decrescente)

Tamanho n = 10 | Tempo ordenado decrescente: 0.000004 segundos
Tamanho n = 20 | Tempo ordenado decrescente: 0.000005 segundos
Tamanho n = 30 | Tempo ordenado decrescente: 0.000006 segundos
Tamanho n = 40 | Tempo ordenado decrescente: 0.000007 segundos
Tamanho n = 50 | Tempo ordenado decrescente: 0.000008 segundos
Tamanho n = 60 | Tempo ordenado decrescente: 0.000010 segundos
Tamanho n = 70 | Tempo ordenado decrescente: 0.000011 segundos
Tamanho n = 80 | Tempo ordenado decrescente: 0.000013 segundos
Tamanho n = 90 | Tempo ordenado decrescente: 0.000014 segundos
Tamanho n = 100 | Tempo ordenado decrescente: 0.000015 segundos
Tamanho n = 110 | Tempo ordenado decrescente: 0.000016 segundos
Tamanho n = 120 | Tempo ordenado decrescente: 0.000018 segundos
Tamanho n = 130 | Tempo ordenado decrescente: 0.000019 segundos
Tamanho n = 140 | Tempo ordenado decrescente: 0.000021 segundos
Tamanho n = 150 | Tempo ordenado decrescente: 0.000022 segundos
Tamanho n = 160 | Tempo ordenado decrescente: 0.000023 segundos
Tamanho n = 170 | Tempo ordenado decrescente: 0.000025 segundos
Tamanho n = 180 | Tempo ordenado decrescente: 0.000026 segundos
Tamanho n = 190 | Tempo ordenado decrescente: 0.000027 segundos
Tamanho n = 200 | Tempo ordenado decrescente: 0.000028 segundos
```

Figura 6: Resultado Radix Sort



```
PS C:\Users\johna\OneDrive\Área de Trabalho\TrabalhoPAA> python .\radixSort.py
Original: [27287, 74058, 30845, 64876, 97470, 59348, 25483, 90564, 39644, 78264]
Crescente:  [25483, 27287, 30845, 39644, 59348, 64876, 74058, 78264, 90564, 97470]
Decrescente: [97470, 90564, 78264, 74058, 64876, 59348, 39644, 30845, 27287, 25483]


Radix Sort - Tempo para vetores ORIGINAIS

Tamanho n = 10 | Tempo original: 0.000030 segundos
Tamanho n = 20 | Tempo original: 0.000034 segundos
Tamanho n = 30 | Tempo original: 0.000044 segundos
Tamanho n = 40 | Tempo original: 0.000054 segundos
Tamanho n = 50 | Tempo original: 0.000065 segundos
Tamanho n = 60 | Tempo original: 0.000076 segundos
Tamanho n = 70 | Tempo original: 0.000088 segundos
Tamanho n = 80 | Tempo original: 0.000102 segundos
Tamanho n = 90 | Tempo original: 0.000114 segundos
Tamanho n = 100 | Tempo original: 0.000124 segundos
Tamanho n = 110 | Tempo original: 0.000133 segundos
Tamanho n = 120 | Tempo original: 0.000147 segundos
Tamanho n = 130 | Tempo original: 0.000156 segundos
Tamanho n = 140 | Tempo original: 0.000169 segundos
Tamanho n = 150 | Tempo original: 0.000176 segundos
Tamanho n = 160 | Tempo original: 0.000188 segundos
Tamanho n = 170 | Tempo original: 0.000198 segundos
Tamanho n = 180 | Tempo original: 0.000211 segundos
Tamanho n = 190 | Tempo original: 0.000315 segundos
Tamanho n = 200 | Tempo original: 0.000240 segundos

Radix Sort - Tempo para vetores ordenados (crescente)

Tamanho n = 10 | Tempo ordenado crescente: 0.000021 segundos
Tamanho n = 20 | Tempo ordenado crescente: 0.000026 segundos
Tamanho n = 30 | Tempo ordenado crescente: 0.000034 segundos
Tamanho n = 40 | Tempo ordenado crescente: 0.000042 segundos
Tamanho n = 50 | Tempo ordenado crescente: 0.000053 segundos
Tamanho n = 60 | Tempo ordenado crescente: 0.000060 segundos
Tamanho n = 70 | Tempo ordenado crescente: 0.000070 segundos
Tamanho n = 80 | Tempo ordenado crescente: 0.000082 segundos
Tamanho n = 90 | Tempo ordenado crescente: 0.000088 segundos
Tamanho n = 100 | Tempo ordenado crescente: 0.000100 segundos
Tamanho n = 110 | Tempo ordenado crescente: 0.000108 segundos
Tamanho n = 120 | Tempo ordenado crescente: 0.000115 segundos
Tamanho n = 130 | Tempo ordenado crescente: 0.000125 segundos
Tamanho n = 140 | Tempo ordenado crescente: 0.000146 segundos
Tamanho n = 150 | Tempo ordenado crescente: 0.000142 segundos
Tamanho n = 160 | Tempo ordenado crescente: 0.000148 segundos
Tamanho n = 170 | Tempo ordenado crescente: 0.000158 segundos
Tamanho n = 180 | Tempo ordenado crescente: 0.000164 segundos
Tamanho n = 190 | Tempo ordenado crescente: 0.000174 segundos
Tamanho n = 200 | Tempo ordenado crescente: 0.000183 segundos

Radix Sort - Tempo para vetores ordenados (decrescente)

Tamanho n = 10 | Tempo ordenado decrescente: 0.000015 segundos
Tamanho n = 20 | Tempo ordenado decrescente: 0.000024 segundos
Tamanho n = 30 | Tempo ordenado decrescente: 0.000032 segundos
Tamanho n = 40 | Tempo ordenado decrescente: 0.000041 segundos
Tamanho n = 50 | Tempo ordenado decrescente: 0.000049 segundos
Tamanho n = 60 | Tempo ordenado decrescente: 0.000061 segundos
Tamanho n = 70 | Tempo ordenado decrescente: 0.000067 segundos
Tamanho n = 80 | Tempo ordenado decrescente: 0.000076 segundos
Tamanho n = 90 | Tempo ordenado decrescente: 0.000085 segundos
Tamanho n = 100 | Tempo ordenado decrescente: 0.000095 segundos
Tamanho n = 110 | Tempo ordenado decrescente: 0.000103 segundos
Tamanho n = 120 | Tempo ordenado decrescente: 0.000112 segundos
Tamanho n = 130 | Tempo ordenado decrescente: 0.000120 segundos
Tamanho n = 140 | Tempo ordenado decrescente: 0.000130 segundos
Tamanho n = 150 | Tempo ordenado decrescente: 0.000140 segundos
Tamanho n = 160 | Tempo ordenado decrescente: 0.000146 segundos
Tamanho n = 170 | Tempo ordenado decrescente: 0.000153 segundos
Tamanho n = 180 | Tempo ordenado decrescente: 0.000165 segundos
Tamanho n = 190 | Tempo ordenado decrescente: 0.000171 segundos
Tamanho n = 200 | Tempo ordenado decrescente: 0.000186 segundos
```

Figura 7: Resultado Heap Sort



```
Heap Sort - Tempo para vetores ORIGINAIS

Tamanho n = 10 | Tempo original: 0.000013 segundos
Tamanho n = 20 | Tempo original: 0.000017 segundos
Tamanho n = 30 | Tempo original: 0.000028 segundos
Tamanho n = 40 | Tempo original: 0.000046 segundos
Tamanho n = 50 | Tempo original: 0.000053 segundos
Tamanho n = 60 | Tempo original: 0.000072 segundos
Tamanho n = 70 | Tempo original: 0.000077 segundos
Tamanho n = 80 | Tempo original: 0.000090 segundos
Tamanho n = 90 | Tempo original: 0.000105 segundos
Tamanho n = 100 | Tempo original: 0.000119 segundos
Tamanho n = 110 | Tempo original: 0.000133 segundos
Tamanho n = 120 | Tempo original: 0.000128 segundos
Tamanho n = 130 | Tempo original: 0.000129 segundos
Tamanho n = 140 | Tempo original: 0.000141 segundos
Tamanho n = 150 | Tempo original: 0.000154 segundos
Tamanho n = 160 | Tempo original: 0.000173 segundos
Tamanho n = 170 | Tempo original: 0.000180 segundos
Tamanho n = 180 | Tempo original: 0.000195 segundos
Tamanho n = 190 | Tempo original: 0.000206 segundos
Tamanho n = 200 | Tempo original: 0.000216 segundos

Heap Sort - Tempo para vetores ORDENADOS (crescente)

Tamanho n = 10 | Tempo ordenado crescente: 0.000008 segundos
Tamanho n = 20 | Tempo ordenado crescente: 0.000016 segundos
Tamanho n = 30 | Tempo ordenado crescente: 0.000025 segundos
Tamanho n = 40 | Tempo ordenado crescente: 0.000034 segundos
Tamanho n = 50 | Tempo ordenado crescente: 0.000044 segundos
Tamanho n = 60 | Tempo ordenado crescente: 0.000082 segundos
Tamanho n = 70 | Tempo ordenado crescente: 0.000088 segundos
Tamanho n = 80 | Tempo ordenado crescente: 0.000075 segundos
Tamanho n = 90 | Tempo ordenado crescente: 0.000086 segundos
Tamanho n = 100 | Tempo ordenado crescente: 0.000102 segundos
Tamanho n = 110 | Tempo ordenado crescente: 0.000108 segundos
Tamanho n = 120 | Tempo ordenado crescente: 0.000150 segundos
Tamanho n = 130 | Tempo ordenado crescente: 0.000179 segundos
Tamanho n = 140 | Tempo ordenado crescente: 0.000180 segundos
Tamanho n = 150 | Tempo ordenado crescente: 0.000194 segundos
Tamanho n = 160 | Tempo ordenado crescente: 0.000161 segundos
Tamanho n = 170 | Tempo ordenado crescente: 0.000180 segundos
Tamanho n = 180 | Tempo ordenado crescente: 0.000194 segundos
Tamanho n = 190 | Tempo ordenado crescente: 0.000244 segundos
Tamanho n = 200 | Tempo ordenado crescente: 0.000264 segundos

Heap Sort - Tempo para vetores ORDENADOS (decrescente)

Tamanho n = 10 | Tempo ordenado decrescente: 0.000007 segundos
Tamanho n = 20 | Tempo ordenado decrescente: 0.000015 segundos
Tamanho n = 30 | Tempo ordenado decrescente: 0.000021 segundos
Tamanho n = 40 | Tempo ordenado decrescente: 0.000030 segundos
Tamanho n = 50 | Tempo ordenado decrescente: 0.000047 segundos
Tamanho n = 60 | Tempo ordenado decrescente: 0.000057 segundos
Tamanho n = 70 | Tempo ordenado decrescente: 0.000067 segundos
Tamanho n = 80 | Tempo ordenado decrescente: 0.000066 segundos
Tamanho n = 90 | Tempo ordenado decrescente: 0.000093 segundos
Tamanho n = 100 | Tempo ordenado decrescente: 0.000085 segundos
Tamanho n = 110 | Tempo ordenado decrescente: 0.000152 segundos
Tamanho n = 120 | Tempo ordenado decrescente: 0.000147 segundos
Tamanho n = 130 | Tempo ordenado decrescente: 0.000150 segundos
Tamanho n = 140 | Tempo ordenado decrescente: 0.000160 segundos
Tamanho n = 150 | Tempo ordenado decrescente: 0.000173 segundos
Tamanho n = 160 | Tempo ordenado decrescente: 0.000171 segundos
Tamanho n = 170 | Tempo ordenado decrescente: 0.000193 segundos
Tamanho n = 180 | Tempo ordenado decrescente: 0.000174 segundos
Tamanho n = 190 | Tempo ordenado decrescente: 0.000185 segundos
Tamanho n = 200 | Tempo ordenado decrescente: 0.000196 segundos
```