Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação (SCC)
Disciplina: Introdução à Ciência de Computação II

PROJETO 2 – MAIS ORDENAÇÃO

Kattryel Henrique Santos Rezende
Marcos Vinicius Cota Rodrigues da Trindade

1. INTRODUÇÃO

O presente relatório tem como objetivo apresentar os resultados e análises decorrentes da implementação e avaliação de diferentes algoritmos de ordenação. Esse trabalho visa comparar o desempenho de diversos métodos de ordenação clássicos em termos de três métricas fundamentais: número de comparações, número de movimentações de registros e tempo de execução. Os algoritmos implementados incluem **BubbleSort**, **SelectionSort**, **InsertionSort**, **ShellSort**, **QuickSort** (com mediana de 3 como estratégia para escolha do pivô), **HeapSort**, **MergeSort**, **Contagem dos Menores** e **RadixSort**.

O problema consistia em analisar a eficiência desses algoritmos de ordenação, avaliando seu comportamento em diferentes cenários. Para resolver o problema, implementei todos os métodos solicitados em um único arquivo e, utilizando os vetores mencionados, medi as métricas de desempenho para cada caso. Dessa forma, foi possível obter dados que permitiram avaliar a eficiência de cada algoritmo sob diferentes condições.

Para avaliar o desempenho de cada método, utilizamos vetores de números inteiros com tamanhos variados (100, 1.000, 10.000 e 100.000 elementos) e três configurações distintas: vetores previamente ordenados, inversamente ordenados e com elementos aleatórios. No caso dos vetores aleatórios, foram executadas cinco iterações com diferentes elementos para cada tamanho, permitindo o cálculo da média dos resultados obtidos.

2. ALGORITMOS DE ORDENAÇÃO

Nesta seção, apresentamos a descrição dos algoritmos de ordenação implementados no projeto. Cada método é detalhado com base em seu funcionamento, estratégia utilizada para reorganizar os elementos do vetor e características que influenciam seu desempenho em diferentes cenários. É importante destacar que, para este trabalho, adotamos uma definição específica de movimentação de registros: consideramos como **movimentação** toda ação que envolve a **troca de dois registros de suas respectivas posições no vetor**. Essa métrica será fundamental para a análise de desempenho e comparação entre os algoritmos apresentados.

2.1. BUBBLE SORT

O BubbleSort é um dos algoritmos de ordenação mais simples e conhecidos, baseado na ideia de "borbulhar" os maiores elementos de um vetor para suas posições corretas ao final, por meio de comparações e trocas sucessivas entre pares de elementos adjacentes. Apesar de sua simplicidade, ele é menos eficiente em termos de desempenho em relação a outros algoritmos mais avançados.

O algoritmo percorre o vetor repetidamente, comparando elementos adjacentes. Se o elemento atual for maior que o próximo, os dois são trocados de posição. Esse processo garante que, ao final de cada passada, o maior elemento da sequência ainda não ordenada estará na posição correta. O algoritmo então reduz a área de comparação, ignorando os elementos já ordenados nas próximas iterações. O processo continua até que nenhuma troca seja necessária em uma iteração completa, indicando que o vetor está ordenado.

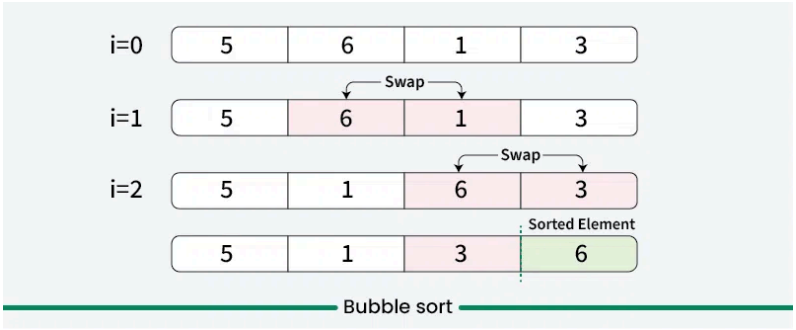


Diagrama ilustrativo retirado de https://www.geeksforgeeks.org/bubble-sort-algorithm/

A cada iteração, o algoritmo realiza comparações entre pares de elementos adjacentes. Em um vetor de tamanho n, no pior caso, o BubbleSort realiza $n - 1$ comparações na primeira iteração, $n - 2$ na segunda, e assim por diante, resultando em um total de:

$$C(n) = \frac{n(n-1)}{2}$$

Isso equivale à complexidade de O(n^2) no pior e no caso médio. No melhor caso, quando o vetor já está ordenado, o algoritmo pode ser otimizado para interromper as iterações após a primeira passada, resultando em O(n) comparações.

Uma movimentação ocorre sempre que dois elementos são trocados de posição. No pior caso, quando o vetor está ordenado de forma inversa, o algoritmo realiza o mesmo número de trocas que de comparações, ou seja, O(n^2). No melhor caso, quando o vetor já está ordenado, nenhuma troca é realizada, resultando em O(1) movimentações.

1. **Complexidade de Tempo:**
- **Melhor caso:** O(n), quando o vetor já está ordenado.
- **Pior caso:** O(n^2), quando o vetor está em ordem inversa.
- **Caso médio:** O(n^2).
2. **Complexidade de Espaço:**
- O(1) (*in-place*), já que o algoritmo opera diretamente sobre o vetor e não requer armazenamento adicional significativo.

Resultados do Bubblesort:

Quantidade de elementos	Tempo de execução (s)			Comparações de registros			Movimentações de registros		
	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente
100	0.000032	0.000011	0.000046	9900	9900	9900	2250	0	4950
1,000	0.002518	0.000933	0.004260	999000	999000	999000	249622	0	499500
10,000	0.247207	0.014000	0.421964	9999000	9999000	9999000	24982568	0	49995000
100,000	27.979000	0.903300	42.196400	99999000	99999000	99999000	249825652	0	4999950000

2.2. SELECTION SORT

O SelectionSort é um algoritmo de ordenação baseado na ideia de selecionar o menor (ou maior) elemento em cada iteração e colocá-lo em sua posição correta no vetor. É simples de implementar e não depende de trocas desnecessárias, o que o torna um pouco mais eficiente do que o BubbleSort em termos de movimentações de registros. No entanto, seu número elevado de comparações mantém sua complexidade de tempo em O(n^2).

O SelectionSort divide o vetor em duas partes:

1. Uma sublista ordenada (no início do vetor).
2. Uma sublista não ordenada (no restante do vetor).

Em cada iteração, o algoritmo busca o menor elemento na sublista não ordenada e o troca com o elemento na primeira posição dessa sublista. A área da sublista ordenada cresce a cada iteração, enquanto a sublista não ordenada diminui. Esse processo continua até que todos os elementos estejam na sublista ordenada e o vetor esteja completamente ordenado.

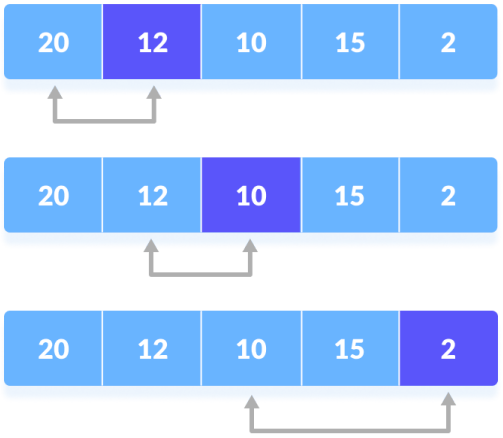


Diagrama ilustrativo retirado de https://www.programiz.com/dsa/selection-sort

O algoritmo realiza uma quantidade fixa de comparações por iteração para encontrar o menor elemento na sublista não ordenada. No total, para um vetor de tamanho n, ele realiza:

$$C(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$$

Assim, o número de comparações segue uma complexidade de tempo O(n^2) tanto no pior quanto no caso médio e melhor caso, já que o processo de busca pelo menor elemento é sempre executado completamente, independentemente da ordenação inicial.

No SelectionSort, uma movimentação ocorre apenas quando o menor elemento encontrado é diferente do primeiro elemento da sublista não ordenada. Isso resulta em, no máximo, $n - 1$ movimentações de registros para um vetor de n elementos, pois há no máximo uma troca por iteração. Essa característica o torna mais eficiente do que o BubbleSort em termos de movimentações.

1. **Complexidade de Tempo:**
   - **Melhor caso:** O(n^2).
   - **Pior caso:** O(n^2).
   - **Caso médio:** O(n^2).

   A complexidade de tempo é independente da ordenação inicial, pois o algoritmo realiza todas as comparações necessárias para encontrar o menor elemento em cada iteração.

2. **Complexidade de Espaço:**
   - O(1) *(in-place)*, já que o algoritmo opera diretamente sobre o vetor e não requer armazenamento adicional significativo.

Resultados do Selectionsort:

Quantidade de elementos	Tempo de execução (s)			Comparações de registros			Movimentações de registros		
	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente
100	0.000014	0.000013	0.000014	4950	4950	4950	93	0	50
1,000	0.001156	0.001134	0.011680	499500	499500	499500	992	0	500
10,000	0.116168	0.115707	0.117979	49995000	49995000	49995000	9990	0	5000
100,000	11.669452	11.763043	11.700141	4999950000	4999950000	4999950000	99990	0	50000

2.3. INSERTION SORT

O **InsertionSort** é um algoritmo de ordenação baseado no conceito de construir gradualmente uma sublista ordenada, inserindo elementos da parte não ordenada na posição correta dentro da parte já ordenada. É conhecido por sua simplicidade e eficiência relativa para vetores pequenos ou parcialmente ordenados, apresentando bom desempenho em comparação com outros algoritmos básicos.

O algoritmo divide o vetor em duas partes:

1. Uma sublista ordenada (inicialmente contendo apenas o primeiro elemento do vetor).
2. Uma sublista não ordenada (os elementos restantes).

A cada iteração, o primeiro elemento da sublista não ordenada é retirado e comparado com os elementos da sublista ordenada, de trás para frente. O elemento retirado é então inserido na posição correta dentro da sublista ordenada, deslocando os elementos maiores para a direita. Esse processo se repete até que todos os elementos estejam na sublista ordenada.

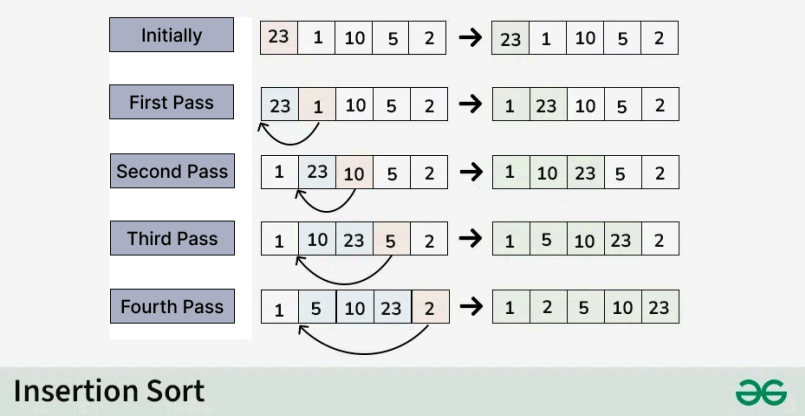


Diagrama ilustrativo retirado de https://www.geeksforgeeks.org/insertion-sort-algorithm/

O número de comparações depende diretamente da ordem inicial dos elementos:

- **Melhor caso:** Ocorre quando o vetor já está ordenado. Nesse caso, apenas uma comparação é feita por elemento, totalizando $n - 1$ comparações (O(n)).

- **Pior caso:** Ocorre quando o vetor está em ordem inversa. Nesse cenário, cada elemento da sublista não ordenada é comparado com todos os elementos da sublista ordenada, resultando em $\frac{n(n-1)}{2}$ comparações (O(n^2)).
- **Caso médio:** Em uma entrada aleatória, o número médio de comparações também é O(n^2).

Uma movimentação ocorre sempre que um elemento é deslocado para a direita para dar espaço ao elemento sendo inserido. Assim como nas comparações, o número de movimentações depende da ordem inicial do vetor:

- **Melhor caso:** O(1) movimentação por elemento, pois não há necessidade de deslocamentos, totalizando O(n).
- **Pior caso:** O vetor em ordem inversa exige deslocamentos para cada comparação, resultando em $\frac{n(n-1)}{2}$ movimentações (O(n^2)).
- **Caso médio:** Segue O(n^2).

Resultados do Insertionsort:

Quantidade de elementos	Tempo de execução (s)			Comparações de registros			Movimentações de registros		
	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente
100	0.000008	0.000002	0.000011	2250	0	4950	2349	99	5049
1,000	0.000359	0.000020	0.000703	249622	0	499500	250621	999	500499
10,000	0.021216	0.000044	0.039660	24982568	0	49995000	24992567	9999	50004999
100,000	2.261475	0.000419	3.752967	249825652	0	4999950000	246975235	99999	5000049999

2.4. SHELL SORT

O **ShellSort** é uma extensão do **InsertionSort** que melhora seu desempenho ao permitir a comparação e troca de elementos que estão distantes no vetor, diminuindo gradativamente essa distância (gap) até que a ordenação seja concluída. Essa abordagem reduz consideravelmente o número de movimentações e comparações necessárias para alcançar um vetor ordenado, especialmente para conjuntos de dados grandes.

Nesta implementação, utilizamos a **sequência de Knuth** para definir os valores dos gaps, proporcionando uma escolha eficiente de distâncias para as iterações do algoritmo. A sequência de Knuth é definida como:

$$h_k = 1,\ 3,\ 7,\ 15,\ 31,\ \ldots,\ (3^k - 1)/2,\ com\ h_k < n$$

Onde n é o tamanho do vetor.

1. Inicialmente, define-se uma sequência de gaps com base na fórmula de Knuth, começando com o maior gap menor que o tamanho do vetor.
2. Para cada valor de gap, o vetor é particionado em sublistas formadas por elementos separados por essa distância.
3. Cada sublista é ordenada individualmente utilizando o **InsertionSort**.
4. O gap é reduzido de acordo com a sequência de Knuth, até que o valor do gap seja 1, momento em que o vetor é ordenado como um todo, similar ao **InsertionSort**.

Essa estratégia permite que os elementos "se movam" mais rapidamente para suas posições corretas, reduzindo significativamente o número de comparações e movimentações nas etapas finais.

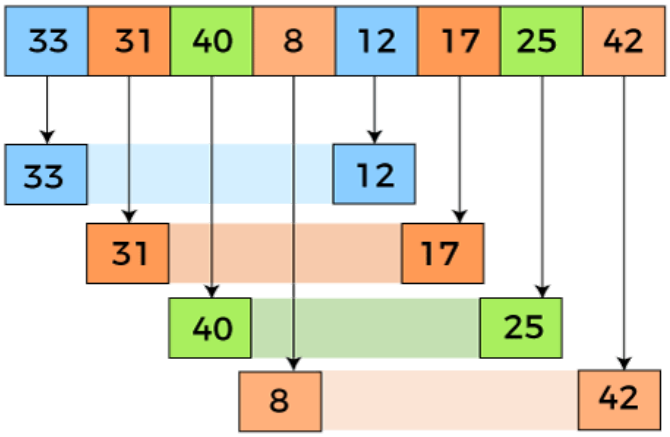


Diagrama ilustrativo retirado de https://www.javatpoint.com/shell-sort

O número de comparações realizadas pelo ShellSort depende diretamente da escolha dos gaps e da ordem inicial dos elementos. A sequência de Knuth garante uma redução eficiente do número de comparações em comparação ao **InsertionSort**, especialmente para vetores desordenados.

- **Melhor caso:** Ocorre quando os elementos já estão quase ordenados. As comparações são limitadas, com complexidade O($nlogn$).
- **Pior caso:** Depende da sequência de gaps e pode aproximar-se de O(n^2), embora, com a sequência de Knuth, a complexidade seja geralmente inferior a n^2.
- **Caso médio:** Aproxima-se de O($n^{1.5}$) para a sequência de Knuth.

Assim como as comparações, as movimentações também dependem dos gaps escolhidos e da ordenação inicial do vetor. A sequência de Knuth reduz significativamente o número de deslocamentos em comparação com o **InsertionSort**, já que os elementos são "levados" para mais perto de suas posições finais mais rapidamente.

- **Melhor caso:** O($nlogn$), com poucos deslocamentos necessários.
- **Pior caso:** Aproxima-se de O(n^2), embora menos frequente com gaps bem escolhidos.
- **Caso médio:** O($n^{1.5}$).

Complexidade de Tempo:

- **Melhor caso:** O($nlogn$).
- **Pior caso:** Depende da sequência de gaps; com Knuth, usualmente O($n^{1.5}$).
- **Caso médio:** Aproxima-se de O($n^{1.5}$) com a sequência de Knuth.

Complexidade de Espaço:

- O(1) *(in-place)*, já que não requer memória adicional significativa.

Resultados do Shellsort:

Quantidade de elementos	Tempo de execução (s)			Comparações de registros			Movimentações de registros		
	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente
100	0.000005	0.000002	0.000004	394	0	230	736	342	572
1,000	0.000058	0.000020	0.000030	9381	0	5990	14202	4821	10811
10,000	0.000810	0.000165	0.000631	347612	0	441002	397433	49821	490823
100,000	0.028298	0.000879	0.043974	23568519	0	41551654	24068340	499821	42051475

2.5. QUICK SORT

O **QuickSort** é um algoritmo de ordenação eficiente e amplamente utilizado, baseado no paradigma de divisão e conquista. Ele particiona o vetor em duas sublistas ao redor de um pivô, ordenando recursivamente cada sublista até que o vetor esteja completamente ordenado.

Nesta implementação, a escolha do pivô foi feita utilizando a estratégia da **mediana de três**, que seleciona como pivô a mediana entre o primeiro elemento, o último elemento e o elemento central da sublista. Essa abordagem melhora a eficiência do QuickSort, reduzindo a probabilidade de particionamentos desbalanceados, especialmente em vetores quase ordenados.

1. Escolhe-se um pivô utilizando a estratégia de mediana de três.
2. Os elementos do vetor são rearranjados de modo que todos os valores menores que o pivô fiquem à esquerda e todos os valores maiores fiquem à direita.
3. O algoritmo é chamado recursivamente para as sublistas à esquerda e à direita do pivô.
4. O processo continua até que cada sublista tenha no máximo um elemento, o que garante que o vetor esteja ordenado.

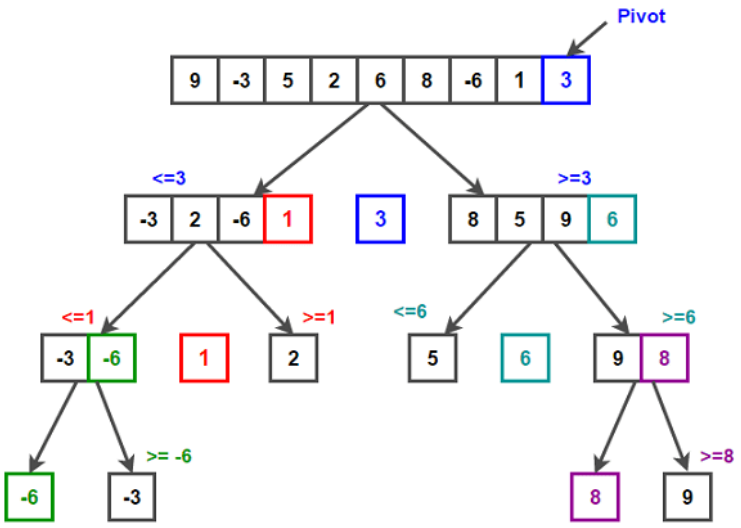


Diagrama ilustrativo retirado de https://dev.to/dormin/ordenacao-rapida-quick-sort-171f

A eficiência do QuickSort em termos de comparações depende fortemente do balanceamento dos particionamentos:

- **Melhor caso:** Ocorre quando o pivô divide o vetor em duas partes aproximadamente iguais a cada iteração. Nesse caso, o número de comparações é proporcional a O($nlogn$).
- **Pior caso:** Ocorre quando os particionamentos são extremamente desbalanceados (por exemplo, um dos lados contém todos os elementos). Isso resulta em O(n^2) comparações. A escolha da mediana de três reduz significativamente a probabilidade desse caso.
- **Caso médio:** Para uma entrada aleatória, o QuickSort realiza O($nlogn$) comparações.

Cada movimentação ocorre ao reorganizar os elementos em torno do pivô. A quantidade de movimentações segue um padrão similar ao das comparações, dependendo do balanceamento das partições:

- **Melhor caso:** O($nlogn$).
- **Pior caso:** O(n^2), embora raro com a mediana de três.
- **Caso médio:** O($nlogn$).

Complexidade de Tempo:

- **Melhor caso:** O($nlogn$), com particionamentos balanceados.
- **Pior caso:** O(n^2), com particionamentos desbalanceados.
- **Caso médio:** O($nlogn$).

Complexidade de Espaço:

- **Melhor caso:** O($logn$), devido à pilha de recursão.
- **Pior caso:** O(n), no caso de particionamentos extremamente desbalanceados.,

Resultados do Quicksort:

Quantidade de elementos	Tempo de execução (s)			Comparações de registros			Movimentações de registros		
	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente
100	0.000008	0.000008	0.000008	884	705	710	188	63	112
1,000	0.000068	0.000034	0.000030	12642	10008	10016	2695	511	1010
10,000	0.000803	0.000250	0.000340	169160	135438	135452	34553	5904	10904
100,000	0.008364	0.004238	0.002569	2123048	1700015	1700030	412165	65535	115534

2.6. HEAP SORT

O **HeapSort** é um algoritmo de ordenação eficiente e confiável que utiliza a estrutura de dados de **heap binário** (máximo ou mínimo) para organizar os elementos. Ele combina a ideia de seleção de máximos/mínimos, como no **SelectionSort**, com a eficiência proporcionada pelo heap, resultando em um desempenho garantido de O($nlogn$) em qualquer situação. Além disso, o HeapSort funciona de maneira ***in-place***, não necessitando de memória adicional significativa, o que o torna uma excelente escolha em sistemas com restrições de espaço.

Diferentemente de algoritmos como o **QuickSort**, cuja eficiência depende da escolha do pivô e pode ter O(n^2) como pior caso, o HeapSort oferece um desempenho consistente em qualquer ordenação inicial, sendo particularmente útil para situações em que a previsibilidade e a estabilidade de tempo de execução são essenciais.

1. O vetor de entrada é convertido em um **heap máximo**, garantindo que o maior elemento esteja na raiz (primeira posição).
2. O maior elemento, localizado na raiz, é trocado com o último elemento do heap.
3. O tamanho do heap é reduzido, e a estrutura do heap é ajustada para manter a propriedade de heap máximo (**heapify**).
4. Esse processo é repetido até que todos os elementos estejam ordenados, com os maiores valores sendo progressivamente movidos para o final do vetor.

Essa abordagem divide o trabalho em duas etapas principais: a construção inicial do heap e a repetição do processo de troca e reorganização, cada uma contribuindo para a eficiência geral do algoritmo.

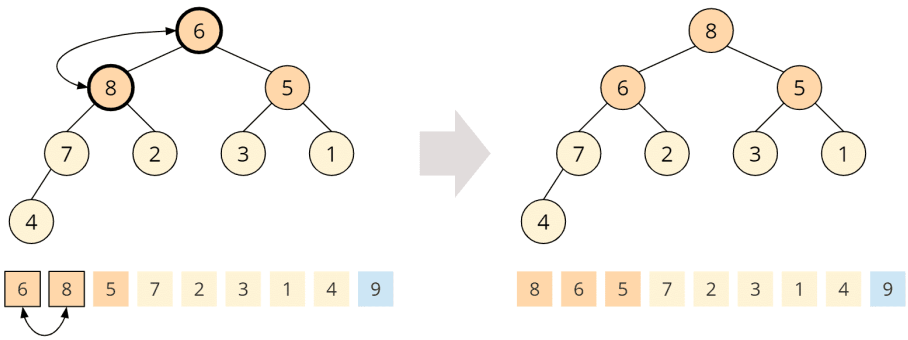


Diagrama ilustrativo retirado de https://www.happycoders.eu/algorithms/heapsort/

As comparações no HeapSort ocorrem durante:

- **Construção do heap:** Na formação inicial do heap, o algoritmo organiza os elementos em O(n) comparações, utilizando uma abordagem bottom-up para otimizar o processo.
- **Reorganização após cada troca:** Após cada remoção do maior elemento (raiz), a propriedade de heap máximo é restaurada, exigindo O($logn$) comparações por operação.

No total, o HeapSort realiza comparações com uma complexidade de tempo O($nlogn$), independentemente da ordenação inicial.

As movimentações de registros ocorrem nas trocas entre a raiz do heap e o último elemento não ordenado, bem como durante o processo de heapificação.

- Cada troca reposiciona o maior elemento em sua posição final, enquanto a heapificação desloca elementos dentro do heap para restaurar sua propriedade.
- Assim, o número de movimentações segue a mesma complexidade O($nlogn$).

Complexidade de Tempo:

- **Melhor caso:** O($nlogn$), devido à consistência das operações do algoritmo.
- **Pior caso:** O($nlogn$), mesmo para entradas altamente desordenadas.
- **Caso médio:** O($nlogn$), para entradas aleatórias.

Complexidade de Espaço:

- O(1) *(in-place)*, já que o HeapSort utiliza o próprio vetor para realizar as operações.

Resultados do Heapsort:

Quantidade de elementos	Tempo de execução (s)			Comparações de registros			Movimentações de registros		
	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente
100	0.000027	0.000028	0.000027	682	742	695	503	544	516
1,000	0.000315	0.000252	0.000253	11882	12363	11670	8342	8716	8316
10,000	0.004846	0.003095	0.002915	168275	174141	166954	116751	121964	116696
100,000	0.052120	0.046194	0.036210	2182197	2241811	2171066	1500644	1550864	1497434

2.7. MERGE SORT

O **MergeSort** é um algoritmo de ordenação eficiente e estável, baseado no paradigma de divisão e conquista. Ele divide repetidamente o vetor em partes menores até que cada sublista contenha apenas um elemento, e então combina essas sublistas de maneira ordenada para reconstruir o vetor original. Sua complexidade O($nlogn$) é garantida em todos os casos, tornando-o uma escolha confiável para aplicações que exigem desempenho consistente e estabilidade.

Embora o MergeSort não seja in-place, pois exige memória adicional para combinar as sublistas, sua estabilidade — ou seja, a preservação da ordem relativa de elementos iguais — o torna uma excelente opção em situações onde isso é necessário.

1. O vetor é dividido recursivamente em duas metades até que cada sublista tenha tamanho 1.
2. As sublistas são então combinadas (**merge**) de forma ordenada, comparando os elementos de ambas as partes e inserindo-os em uma nova lista auxiliar.
3. A fusão continua até que todas as sublistas sejam unidas, resultando em um vetor ordenado.

Essa abordagem é inerentemente recursiva, com o vetor sendo dividido logaritmicamente e combinado linearmente em cada nível de recursão.

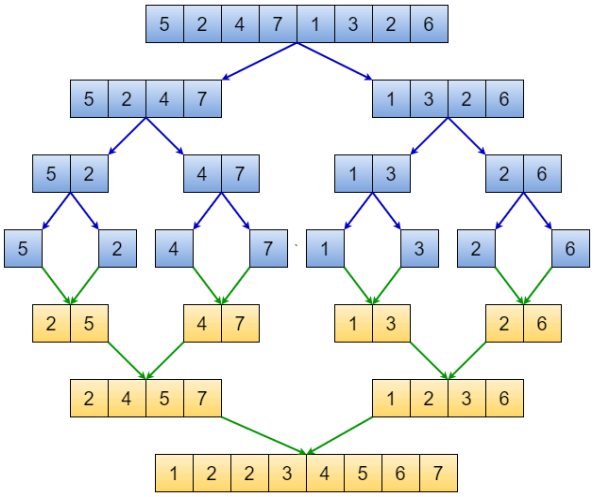


Diagrama ilustrativo retirado de

https://pkerbynn.medium.com/learn-merge-sort-algorithm-like-abc-w-javascript-9e406e35a5f1

As comparações ocorrem durante a fase de fusão, quando elementos das duas sublistas são comparados para determinar a ordem de inserção.

- **Melhor caso:** O($nlogn$), já que o número de comparações é proporcional ao tamanho das sublistas e à profundidade da recursão.
- **Pior caso:** O($nlogn$), mesmo para entradas altamente desordenadas, pois todas as comparações são necessárias para a fusão.
- **Caso médio:** O($nlogn$), com desempenho consistente devido à divisão uniforme e processo de fusão sistemático.

As movimentações são realizadas ao inserir elementos das sublistas no vetor auxiliar durante a fusão. Como cada elemento é movido uma vez por nível de recursão, o número total de movimentações também é O($nlogn$).

Complexidade de Tempo:

- **Melhor caso:** O($nlogn$), devido à divisão uniforme e fusão eficiente.
- **Pior caso:** O($nlogn$), já que a fusão sempre envolve todas as comparações necessárias.
- **Caso médio:** O($nlogn$).

Complexidade de Espaço:

- O(n), devido à necessidade de memória adicional para as sublistas durante a fusão.

Resultados do Mergesort:

Quantidade de elementos	Tempo de execução (s)			Comparações de registros			Movimentações de registros		
	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente
100	0.000019	0.000018	0.000019	541	356	316	1344	1344	1344
1,000	0.000141	0.000158	0.000135	8707	5044	4932	19952	19952	19952
10,000	0.001429	0.001037	0.000920	120463	69008	64608	267232	267232	267232
100,000	0.017396	0.016820	0.015980	1536337	853904	815024	3337856	3337856	3337856

2.8. CONTAGEM DE MENORES

O algoritmo **Contagem dos Menores** (em inglês, **Counting of Smaller Elements**) é um método simples e não comparativo de ordenação, adequado para entradas pequenas ou casos específicos onde a memória não é um problema. Ele utiliza o conceito de contar o número de elementos menores que cada elemento do vetor para determinar diretamente sua posição na ordenação final.

Embora intuitivo e fácil de implementar, o **Contagem dos Menores** não é eficiente para vetores grandes devido à sua complexidade O(n^2). No entanto, é útil como ferramenta didática para compreender algoritmos baseados em contagem e a noção de ordenação direta.

A ideia principal do algoritmo é que, se soubermos quantos elementos são menores que um determinado valor, podemos determinar a posição que esse valor ocupará no vetor ordenado. Por exemplo, se existem 5 valores menores que o elemento 7, então o elemento 7 ocupará a sexta posição no vetor ordenado.

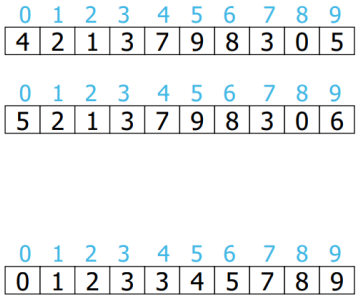


Diagrama ilustrativo retirado dos slides de aula

O algoritmo funciona em duas fases principais:

1. **Contagem dos Menores:** Para cada elemento x_i do vetor original, conta-se quantos elementos são menores que x_i. Essa contagem pode ser realizada ao percorrer o vetor e verificar quantos elementos têm valores menores que o valor de x_i.
2. **Construção do Vetor Ordenado:** Utilizando a contagem dos menores para cada elemento, o algoritmo monta o vetor ordenado. Para isso, um vetor auxiliar é usado para armazenar o

número de elementos menores para cada posição do vetor original, e os elementos são inseridos nas posições apropriadas no vetor ordenado, baseando-se nessas contagens.

As comparações no algoritmo ocorrem durante a contagem dos elementos menores que cada valor. Para um vetor de tamanho n, o algoritmo realiza aproximadamente $\frac{n(n-1)}{2}$ comparações, resultando em uma complexidade de tempo $O(n^2)$.

O algoritmo faz movimentações apenas quando os elementos são posicionados no vetor ordenado. Cada elemento é movido para sua posição final com base na contagem dos elementos menores. Como o algoritmo não realiza trocas sucessivas como em outros métodos de ordenação, o número total de movimentações é $O(n)$, já que cada elemento é movido uma vez para o seu lugar.

Complexidade de Tempo:

- **Melhor caso:** $O(n^2)$, já que a contagem dos elementos menores exige comparações em relação a todos os elementos.
- **Pior caso:** $O(n^2)$, em entradas desordenadas ou em qualquer situação onde o número de comparações é o mesmo.
- **Caso médio:** $O(n^2)$, pois a contagem de menores é sempre feita de maneira semelhante.

Complexidade de Espaço:

- O algoritmo utiliza três vetores (o vetor original, o vetor auxiliar para o arranjo ordenado e o vetor de contagem), logo a complexidade de espaço total é $O(3n)$, o que é $O(n)$.

Resultados do Contagem de Menores:

Quantidade de elementos	Tempo de execução (s)			Comparações de registros			Movimentações de registros		
	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente
100	0.000019	0.000007	0.000008	4950	4950	4950	200	200	200
1,000	0.000992	0.000549	0.000640	499500	499500	499500	2000	2000	2000
10,000	0.132954	0.046897	0.065088	49995000	49995000	49995000	20000	20000	20000
100,000	13.984145	12.985000	13.150000	4999950000	4999950000	4999950000	200000	200000	200000

2.9. RADIX SORT

O **RadixSort** é um algoritmo de ordenação não comparativo altamente eficiente, especialmente quando se trata de ordenar grandes volumes de números inteiros ou strings com um número fixo de dígitos ou caracteres. Ele organiza os elementos com base em seus dígitos ou bits, processando um dígito por vez. Esse processo é repetido para cada posição de dígito, do menos significativo para o mais significativo, ou vice-versa, até que todos os dígitos sejam considerados, resultando em um vetor ordenado.

Por ser um algoritmo estável e não comparativo, o **RadixSort** é particularmente útil quando os dados são números inteiros ou strings de comprimento fixo e quando a ordem relativa dos elementos iguais precisa ser preservada. Seu desempenho é geralmente superior a algoritmos baseados em comparação em situações específicas, especialmente quando o intervalo dos valores é limitado e os dados têm uma estrutura bem definida. O algoritmo trabalha ordenando os elementos por seus dígitos, de modo que os elementos com base em cada dígito são agrupados e ordenados em uma sequência específica. O processo é repetido para cada dígito, começando do menos significativo (casas menores) até o mais significativo (casas maiores).

1. **Ordenação por Dígitos:** Para cada posição de dígito (unidades, dezenas, centenas, etc.), o algoritmo utiliza um método de ordenação estável (como **CountingSort**) para agrupar os elementos com base nesse dígito específico.
2. **Repetição para Cada Dígito:** O algoritmo repete o processo de ordenação para cada dígito, movendo-se para a próxima posição até que todos os dígitos sejam considerados.

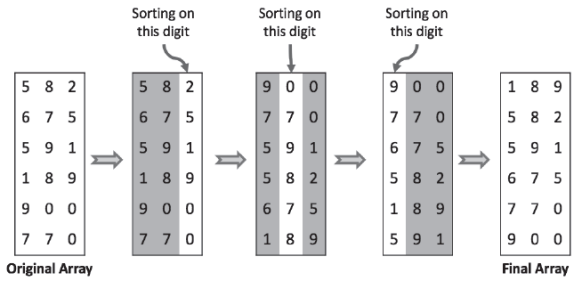


Diagrama ilustrativo retirado de https://www.ritambhara.in/radix-sort/

O RadixSort não realiza comparações diretas entre os elementos. Ao invés disso, ele distribui os elementos nas diferentes "caixas" ou "baldes" de acordo com o valor do dígito

atual. A quantidade de "comparações" realizadas é refletida pelo número de passos de ordenação que ocorrem para cada dígito de cada número no vetor. Isso resulta em uma complexidade de tempo que depende principalmente do número de dígitos do maior número e do número de elementos.

Cada vez que um dígito é considerado, os elementos são movidos para os baldes e, depois, para o vetor original na ordem correta. Como o algoritmo utiliza uma abordagem estável para ordenar por dígitos, as movimentações acontecem de forma que o ordenamento de dígitos anteriores seja preservado. O número total de movimentações no RadixSort é proporcional ao número de elementos no vetor e ao número de dígitos a serem considerados.

Complexidade de Tempo:

- **Melhor caso:** O(nk), onde n é o número de elementos e k é o número de dígitos no maior número (ou o comprimento das cadeias de caracteres, caso seja esse o caso).
- **Pior caso:** O(nk), já que o tempo de execução depende da quantidade de dígitos de cada elemento e não da distribuição dos elementos.
- **Caso médio:** O(nk), com o mesmo comportamento de tempo para entradas médias. A chave para a eficiência do RadixSort é que ele é linear em relação ao número de elementos e ao número de dígitos, ao invés de quadrático como comparações diretas.

Complexidade de Espaço:

- O(n), pois o algoritmo utiliza memória adicional proporcional ao número de elementos para armazenar os baldes e os resultados temporários das ordenações parciais.

Resultados do Radixsort:

Quantidade de elementos	Tempo de execução (s)			Comparações de registros			Movimentações de registros		
	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente	Vetor Aleatório (média)	Vetor Crescente	Vetor Decrescente
100	0.000004	0.000004	0.000005	99	99	99	618	618	618
1,000	0.000023	0.000032	0.000033	999	999	999	9027	9027	9027
10,000	0.000230	0.000232	0.000223	9999	9999	9999	120036	120036	120036
100,000	0.002689	0.002593	0.002492	99999	99999	99999	1500045	1500045	1500045

3. IMPLEMENTAÇÃO

A implementação do projeto foi guiada por uma abordagem metódica e modular, garantindo organização e clareza na coleta de métricas e execução dos experimentos. Todo o código foi projetado para atender aos requisitos de comparação entre métodos de ordenação, enquanto mantinha consistência e precisão nos resultados. É importante destacar que todos os métodos de ordenação foram implementados no mesmo código, selecionados a partir das chamadas do usuário.

3.1. ORGANIZAÇÃO E FLUXO DE EXECUÇÃO

Estrutura Modular

- A base do programa foi dividida em módulos que desempenham papéis específicos, como geração de dados, execução de algoritmos e coleta de métricas. Essa organização permitiu reutilizar funções auxiliares e simplificar a manutenção do código. Além disso, cada algoritmo foi implementado em uma função separada, com assinaturas padronizadas, permitindo fácil integração com o restante do sistema.

Entrada de Dados

- O código inicia com a preparação dos vetores que serão usados como entrada. Para garantir a diversidade e atender aos requisitos do projeto, foram implementadas funções que geram:
    - Vetores ordenados em ordem crescente.
    - Vetores ordenados em ordem decrescente.
    - Vetores aleatórios, com múltiplas versões para cada tamanho de entrada.

  Esses vetores são criados no início do programa e armazenados para serem reutilizados em cada execução, assegurando que todos os algoritmos trabalhem com os mesmos dados de entrada.

Instrumentação para Coleta de Métricas

- Um dos focos principais do projeto foi a coleta de métricas de desempenho (tempo de execução, número de comparações e número de movimentações). Para isso:
    - Contadores globais foram introduzidos no código, sendo zerados antes da execução de cada algoritmo.
    - Funções auxiliares foram instrumentadas nos pontos críticos, como comparações entre elementos e trocas, para registrar esses eventos de forma precisa.
    - O tempo de execução foi medido utilizando a biblioteca padrão <time.h>, capturando com precisão a duração de cada execução.

Execução Sequencial

- O programa principal é responsável por coordenar a execução de todos os algoritmos para cada configuração de vetor (ordenado, inversamente ordenado e aleatório) e tamanho (100, 1.000, 10.000 e 100.000 elementos). Para os vetores aleatórios, cada algoritmo foi executado múltiplas vezes (cinco repetições por tamanho), e as métricas foram calculadas como médias, proporcionando maior confiança nos resultados.

4. ANÁLISE DOS RESULTADOS

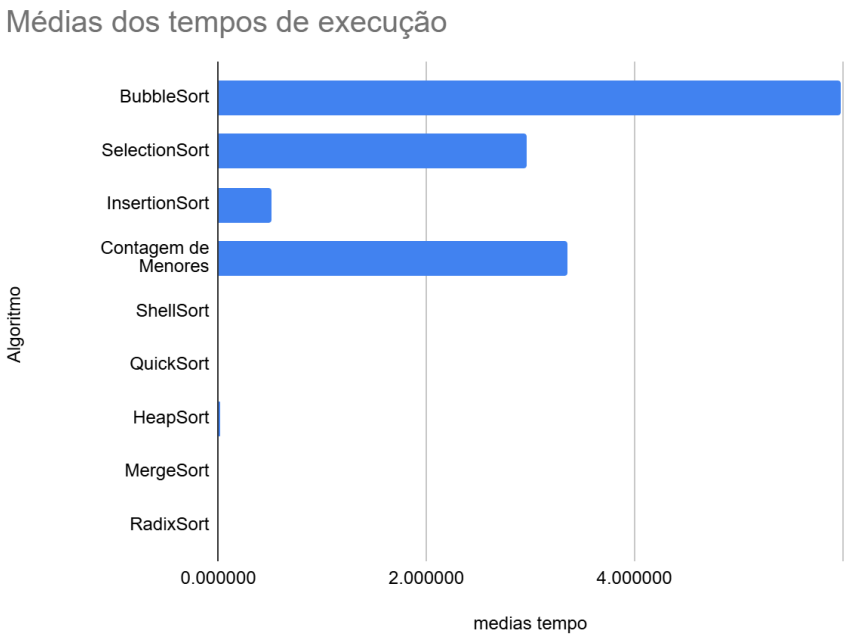
A análise dos algoritmos de ordenação mostrou diferenças marcantes em desempenho. O **RadixSort** foi consistentemente o mais eficiente, graças à sua abordagem baseada em contagens, que evita comparações tradicionais e oferece desempenho linear mesmo para grandes conjuntos de dados, independentemente da ordem inicial. Em contraste, o **BubbleSort** foi o menos eficiente, devido à sua alta complexidade quadrática e ao número excessivo de trocas e comparações, especialmente em listas grandes ou inversamente ordenadas.

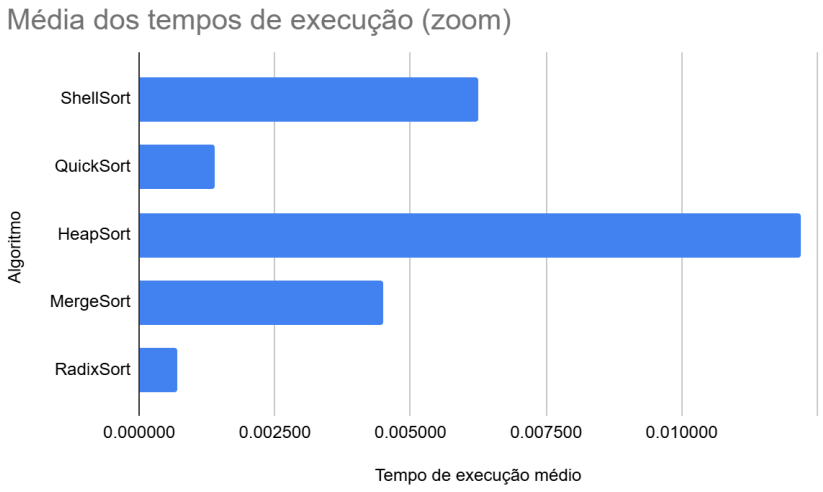
Os demais algoritmos apresentaram desempenhos intermediários, variando conforme o tamanho e a organização dos dados. Algoritmos como o **QuickSort**, **HeapSort** e **MergeSort** mantiveram boa eficiência na maioria dos casos, enquanto o **InsertionSort** foi competitivo em listas pequenas ou já ordenadas. Outros, como o **SelectionSort**, tiveram um desempenho inferior, superados por alternativas mais avançadas.

Em resumo, o **RadixSort** é a melhor escolha para conjuntos grandes devido à sua eficiência consistente, enquanto o **BubbleSort** é impraticável para aplicações reais, servindo apenas como ferramenta didática. A escolha adequada do algoritmo é crucial para garantir eficiência no processamento e uso otimizado de recursos computacionais.

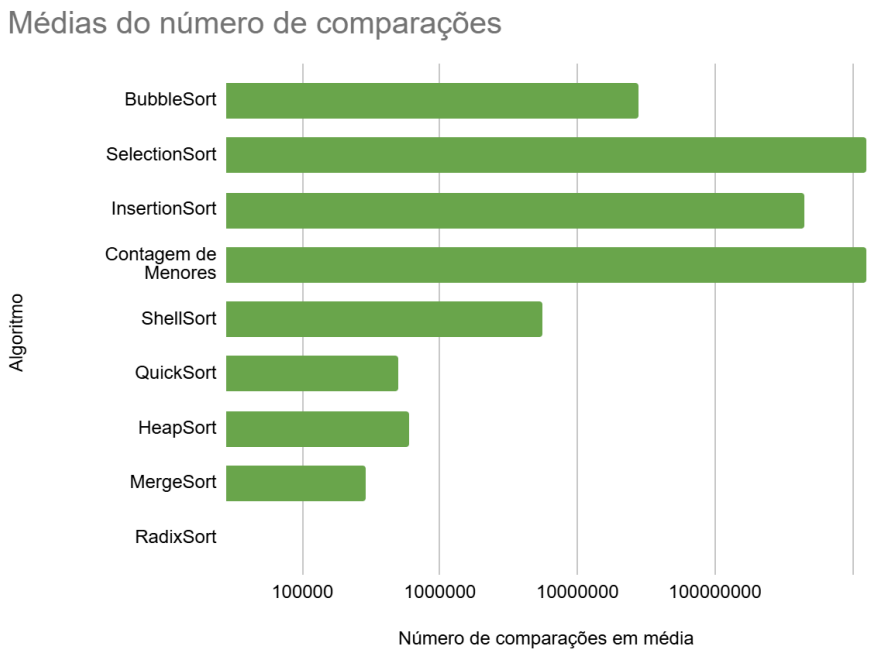
Para facilitar a visualização das diferenças de eficiência entre os algoritmos de ordenação avaliados, organizamos suas médias em gráficos que representam três métricas principais: tempo de execução, número de comparações realizadas e número de movimentações efetuadas. Em alguns casos, barras podem parecer vazias, indicando que os valores são ordens de magnitude menores em relação aos demais, destacando diferenças significativas de desempenho entre os algoritmos. Seguem abaixo os gráficos comparando as três categorias supracitadas:

ANÁLISE 1: TEMPO MÉDIO DE EXECUÇÃO





ANÁLISE 2: QUANTIDADE MÉDIA DE COMPARAÇÕES



ANÁLISE 3: QUANTIDADE MÉDIA DE MOVIMENTAÇÕES