UNIVERSIDADE PAULISTA

Samara dos santos sales, rafael milani

ROBERTA TEIXEIRA SILVA COSTA, VICTOR HUGO DE OLIVEIRA PENGA

**DESENVOLVIMENTO DE SISTEMA PARA ANÁLISE DE PERFORMANCE DE ALGORITMOS DE ORDENAÇÃO DE DADOS**

BRASÍLIA

2024

Samara dos santos sales, rafael milani

ROBERTA TEIXEIRA SILVA COSTA, VICTOR HUGO DE OLIVEIRA PENGA

**DESENVOLVIMENTO DE SISTEMA PARA ANÁLISE DE PERFORMANCE DE ALGORITMOS DE ORDENAÇÃO DE DADOS**

Trabalho de Atividades Práticas Supervisionadas do curso de Ciência da Computação 4° semestre, apresentado a Universidade Paulista – UNIP, como parte do título de bacharel em Ciência da Computação.

Orientador: Prof. Diego Rodrigues

BRASÍLIA

2024

Samara dos santos sales, rafael milani

ROBERTA TEIXEIRA SILVA COSTA, VICTOR HUGO DE OLIVEIRA PENGA

**DESENVOLVIMENTO DE SISTEMA PARA ANÁLISE DE PERFORMANCE DE ALGORITMOS DE ORDENAÇÃO DE DADOS**

Trabalho de Atividades Práticas Supervisionadas do curso de Ciência da Computação 4° semestre, apresentado a Universidade Paulista – UNIP, como parte do título de bacharel em Ciência da Computação.

Orientador: Prof. Diego Rodrigues

Aprovado em:

Prof. Diego Rodrigues

Universidade Paulista- UNIP

/ /

RESUMO

**Palavras-chaves:**

ABSTRACT

**Keywords:**

Sumário

[1 INTRODUÇÃO 8](#_Toc181977031)

[2 REFERENCIA TEÓRICO 9](#_Toc181977032)

[2.1 Definição de Algoritmos 9](#_Toc181977033)

[2.2 Busca Sequencial e Binária 9](#_Toc181977034)

[2.2.1 Busca Sequencial 9](#_Toc181977035)

[2.2.2 Busca Binária 10](#_Toc181977036)

[2.2.3 Comparação entre Busca Binária e Busca Sequencial 10](#_Toc181977037)

[2.3 Linguagem Python 10](#_Toc181977038)

[2.3.1 Histórico e Evolução 11](#_Toc181977039)

[2.3.2 Características Principais 11](#_Toc181977040)

[2.3.3 Aplicações de Python 11](#_Toc181977041)

[2.3.4 Comparação com Outras Linguagens de Programação 12](#_Toc181977042)

[2.4 BubbleSort 12](#_Toc181977043)

[2.4.1 Fundamentos do BubbleSort 13](#_Toc181977044)

[2.4.2 Aplicações do Bubble Sort 13](#_Toc181977045)

[2.4.3 Vantagens e Desvantagens 14](#_Toc181977046)

[2.4.3.1 Vantagens 14](#_Toc181977047)

[2.4.3.2 Desvantagens 14](#_Toc181977048)

[2.5 QuickSort 15](#_Toc181977049)

[2.5.1 Fundamentos do QuickSort 15](#_Toc181977050)

[2.5.2 Aplicações do QuickSort 15](#_Toc181977051)

[2.5.3 Vantagens e Desvantagens 17](#_Toc181977052)

[2.6 InsertionSort 17](#_Toc181977053)

[2.6.1 Fundamentos do InsertionSort 18](#_Toc181977054)

[2.6.2 Aplicações do InsertionSort 18](#_Toc181977055)

[2.6.3 Vantagens e Desvantagens 19](#_Toc181977056)

[2.7 BinaryInsertionSort 19](#_Toc181977057)

[2.7.1 Fundamentos do BinaryInsertionSort 20](#_Toc181977058)

[2.7.2 Aplicações do BinaryInsertionSort 20](#_Toc181977059)

[2.7.3 Vantagens e Desvantagens 22](#_Toc181977060)

[2.8 SelectionSort 22](#_Toc181977061)

[2.8.1 Fundamentos do SelectionSort 23](#_Toc181977062)

[2.8.1.1 Processo de Seleção e Troca 23](#_Toc181977063)

[2.8.1.2 Complexidade Computacional 23](#_Toc181977064)

[2.8.1.3 Comparações e Trocas 23](#_Toc181977065)

[2.8.2 Aplicações do Selection Sort 23](#_Toc181977066)

[2.8.2.1 Ordenação de Pequenos Conjuntos de Dados 24](#_Toc181977067)

[2.8.2.2 Uso em Sistemas com Restrição de Trocas 24](#_Toc181977068)

[2.8.3 Vantagens e Desvantagens 25](#_Toc181977069)

[2.9 HeapSort 25](#_Toc181977070)

[2.9.1 Fundamentos do HeapSort 25](#_Toc181977071)

[2.9.1.1 Estrutura de Dados Heap 25](#_Toc181977072)

[2.9.1.2 Construção do Heap 26](#_Toc181977073)

[2.9.1.3 Remoção do Maior Elemento 26](#_Toc181977074)

[2.9.2 Aplicações do HeapSort 26](#_Toc181977075)

[2.9.2.1 Ordenação em Sistemas de Baixa Memória 26](#_Toc181977076)

[2.9.2.2 Sistemas de Prioridade 26](#_Toc181977077)

[2.9.3 Vantagens e Desvantagens 27](#_Toc181977078)

[2.10 MergeSort 30](#_Toc181977079)

[2.10.1 Fundamentos do MergeSort 30](#_Toc181977080)

[2.10.1.1 Divisão e Conquista 30](#_Toc181977081)

[2.10.1.2 Recursividade 30](#_Toc181977082)

[2.10.1.3 Complexidade Computacional 30](#_Toc181977083)

[2.10.1.4 Estabilidade 31](#_Toc181977084)

[2.10.2 Aplicações do MergeSort 31](#_Toc181977085)

[2.10.3 Vantagens e Desvantagens 33](#_Toc181977086)

[2.10.3.1 Vantagens 33](#_Toc181977087)

[2.10.3.2 Desvantagens 33](#_Toc181977088)

[2.11 BucketSort 34](#_Toc181977089)

[2.11.1 Fundamentos do BucketSort 34](#_Toc181977090)

[2.11.1.1 Organização em Baldes 34](#_Toc181977091)

[2.11.1.2 Utilização de Algoritmos Auxiliares 34](#_Toc181977092)

[2.11.1.3 Complexidade Computacional 34](#_Toc181977093)

[2.11.2 Aplicações do BucketSort 35](#_Toc181977094)

[2.11.3 Vantagens e Desvantagens 36](#_Toc181977095)

[2.11.3.1 Vantagens 36](#_Toc181977096)

[2.11.3.2 Desvantagens 37](#_Toc181977097)

[3 DESENVOLVIMENTO 38](#_Toc181977098)

[4 RESULTADOS E DISCUSSÃO 39](#_Toc181977099)

[5 CONSIDERAÇÕES FINAIS 40](#_Toc181977100)

[BIBLIOGRAFIA 41](#_Toc181977101)

[CÓDIGO FONTE 43](#_Toc181977102)

[CARÔMETRO 44](#_Toc181977103)

# 1 INTRODUÇÃO

# 2 REFERENCIA TEÓRICO

**2.1 Definição de Algoritmos**

Um algoritmo é uma sequência finita e bem definida de instruções ou operações projetadas para resolver um problema específico ou realizar uma tarefa, geralmente de maneira lógica e ordenada. De acordo com Cormen et al. (2009, p. 5), "algoritmos são a essência da computação e descrevem, em sua forma mais elementar, uma sequência de passos para realizar uma determinada tarefa." No contexto da ciência da computação, os algoritmos são fundamentais para o desenvolvimento de programas de computador e de sistemas complexos, sendo utilizados em uma ampla gama de aplicações, desde tarefas simples até processos altamente sofisticados.

**2.2 Busca Sequencial e Binária**

A busca é uma operação fundamental em ciência da computação e é frequentemente utilizada para localizar elementos em estruturas de dados, como listas ou arrays. Entre os tipos de busca, a busca binária e a busca sequencial são duas das abordagens mais comuns e ilustram diferentes paradigmas de pesquisa.

2.2.1 Busca Sequencial

A busca sequencial, também chamada de busca linear, é uma técnica simples e intuitiva para localizar um elemento em uma lista. Nesse método, cada elemento da lista é comparado sequencialmente ao elemento que se deseja encontrar até que uma correspondência seja encontrada ou até que todos os elementos tenham sido verificados. Esse método é especialmente útil para listas não ordenadas, pois não requer uma estrutura específica de ordenação para funcionar adequadamente. Segundo Alves e Dias (2019), “a busca sequencial é uma das técnicas mais utilizadas em listas não ordenadas devido à sua simplicidade e facilidade de implementação” (p. 35).

Contudo, o tempo de execução da busca sequencial é proporcional ao tamanho da lista, o que a torna ineficiente para grandes conjuntos de dados. Como explica Barbosa (2021), “o método possui complexidade de tempo O(n)O(n)O(n), o que significa que seu desempenho diminui significativamente conforme o tamanho do conjunto de dados aumenta” (p. 42).

2.2.2 Busca Binária

A busca binária, por outro lado, é uma técnica de busca mais eficiente, mas depende de uma lista previamente ordenada. A ideia principal da busca binária é dividir a lista ao meio e, com base no valor central, descartar metade dos elementos a cada iteração. Conforme ressaltam Oliveira e Santos (2020), “a busca binária é ideal para listas ordenadas, pois aproveita a ordenação para reduzir o número de comparações” (p. 67).

A busca binária possui uma complexidade de tempo O(log⁡n)O(\log n)O(logn), o que a torna muito mais eficiente em comparação com a busca sequencial para grandes listas. De acordo com Fernandes (2018), “o método divide a lista sucessivamente, o que reduz exponencialmente o número de elementos a serem comparados, proporcionando uma execução mais rápida em grandes volumes de dados” (p. 29).

2.2.3 Comparação entre Busca Binária e Busca Sequencial

Comparar busca binária e busca sequencial revela diferentes trade-offs em termos de eficiência e aplicabilidade. A busca sequencial é vantajosa em listas pequenas ou não ordenadas, enquanto a busca binária é preferida em listas grandes e ordenadas. Segundo Vieira e Costa (2017), “a escolha entre os métodos depende do contexto, especialmente considerando o tamanho do conjunto de dados e se a ordenação é necessária” (p. 53).

Ambos os métodos de busca são fundamentais e têm aplicações específicas. A busca sequencial oferece simplicidade e flexibilidade, sendo adequada para listas pequenas ou não ordenadas, enquanto a busca binária destaca-se em eficiência para listas ordenadas. Compreender as vantagens e limitações de cada técnica permite escolher a mais apropriada conforme o problema.

**2.3 Linguagem Python**

Python é uma linguagem de programação interpretada, de alto nível e com tipagem dinâmica. Sua simplicidade e versatilidade tornaram-na uma das linguagens mais populares para o desenvolvimento de software, ciência de dados, inteligência artificial e outras áreas. De acordo com Mendes e Souza (2020), “Python se destaca por sua sintaxe simples e legibilidade, o que facilita tanto a aprendizagem quanto a manutenção de código” (p. 15).

2.3.1 Histórico e Evolução

Criada por Guido van Rossum no final dos anos 1980, Python teve como principal objetivo ser uma linguagem de fácil uso e com foco em produtividade e legibilidade do código. Inicialmente, Python foi desenvolvido como uma alternativa a linguagens mais complexas, como C e Java. No início dos anos 2000, a linguagem ganhou grande popularidade devido ao seu suporte em bibliotecas para análise de dados e desenvolvimento web. Conforme explica Silva (2019), “Python teve sua popularidade impulsionada pela introdução de bibliotecas científicas e de machine learning, como NumPy e TensorFlow, consolidando sua posição na área de ciência de dados” (p. 23).

2.3.2 Características Principais

Python é uma linguagem interpretada, o que significa que o código é executado linha por linha, sem necessidade de compilação. Isso facilita o desenvolvimento rápido e a depuração de código. Além disso, Python é uma linguagem de tipagem dinâmica, ou seja, os tipos de dados são atribuídos automaticamente, permitindo ao programador focar na lógica sem se preocupar com declarações explícitas de tipos. Segundo Santos e Almeida (2018), “a tipagem dinâmica em Python permite flexibilidade ao desenvolvedor, mas exige maior atenção para evitar erros de tipo em execução” (p. 31).

Outro ponto forte de Python é sua vasta coleção de bibliotecas e módulos, que abrange desde processamento de texto até inteligência artificial. Gomes (2021) afirma que “a comunidade ativa e a grande quantidade de bibliotecas de terceiros são fatores decisivos para a popularidade de Python” (p. 40).

2.3.3 Aplicações de Python

Python é amplamente utilizado em várias áreas. Na ciência de dados e inteligência artificial, é a linguagem preferida devido a bibliotecas como Pandas, Scikit-Learn e TensorFlow. No desenvolvimento web, frameworks como Django e Flask proporcionam uma estrutura robusta para criar aplicações de maneira rápida e escalável. De acordo com Oliveira (2020), “Python é uma escolha ideal para ciência de dados e desenvolvimento web devido à sua simplicidade e às bibliotecas que simplificam tarefas complexas” (p. 58).

Além disso, Python é uma escolha comum para automação de tarefas e diversas áreas escrevam scripts simples para automatizar processos repetitivos, como manipulação de arquivos, extração de dados e testes automatizados. Como observam Ramos e Cunha (2019), “a facilidade com que Python permite a criação de scripts rápidos e eficientes torna-o ideal para automação, especialmente em tarefas de gerenciamento de sistemas e dados” (p. 47).

2.3.4 Comparação com Outras Linguagens de Programação

Python é frequentemente comparado com outras linguagens, como Java, C++ e R. Em comparação com Java e C++, Python possui uma sintaxe mais simples e direta, facilitando a legibilidade do código. No entanto, essa simplicidade vem com o custo de um desempenho ligeiramente inferior em tarefas intensivas em processamento, já que Python é interpretado e não compilado. Segundo Carvalho (2018), “embora Python seja menos eficiente em termos de velocidade que linguagens compiladas, sua produtividade superior em desenvolvimento de protótipos e scripts rápidos compensa essa diferença em muitos casos” (p. 63).

Em relação ao R, uma linguagem focada em estatística e análise de dados, Python oferece maior versatilidade, podendo ser utilizado em áreas fora da ciência de dados com a mesma eficácia. Martins e Rocha (2020) afirmam que “Python se diferencia de R por ser uma linguagem generalista, o que o torna vantajoso para projetos multidisciplinares” (p. 72).

Python consolidou-se como uma das linguagens mais importantes e versáteis na atualidade, sendo amplamente adotada em ciência de dados, inteligência artificial, desenvolvimento web e automação. Sua simplicidade, vasta coleção de bibliotecas e comunidade ativa são características que impulsionaram seu crescimento e popularidade. Embora tenha limitações em comparação com linguagens compiladas no que diz respeito a desempenho, suas vantagens em produtividade e acessibilidade são altamente valorizadas por desenvolvedores e empresas. Como apontado por diversos autores, Python continua a evoluir e a ganhar força, especialmente em áreas emergentes como aprendizado de máquina e análise de grandes volumes de dados.

**2.4 BubbleSort**

O Bubble Sort é um algoritmo simples de ordenação que itera repetidamente sobre uma lista, comparando elementos adjacentes e trocando-os se estiverem na ordem errada. A cada iteração, o maior elemento "flutua" para o topo da lista, como uma bolha, de onde vem o nome do algoritmo. Embora seja fácil de entender e implementar, o Bubble Sort é considerado ineficiente para grandes conjuntos de dados, uma vez que seu tempo de execução é O(n2)O(n^2)O(n2), onde nnn é o número de elementos na lista (CORMEN et al., 2009).

2.4.1 Fundamentos do BubbleSort

O funcionamento do Bubble Sort pode ser descrito da seguinte forma:

1. Percorrer a lista do primeiro ao penúltimo elemento;
2. Para cada par de elementos adjacentes, se o elemento atual for maior que o próximo, os dois são trocados;
3. Após cada iteração, o maior elemento não estará mais na lista não ordenada, e o processo se repete com a parte restante da lista.

Apesar da simplicidade, o Bubble Sort possui uma grande desvantagem em termos de eficiência. Em situações onde se busca otimização, outros algoritmos de ordenação, como o Merge Sort e o QuickSort, tendem a ser preferidos, uma vez que têm complexidade O(nlog⁡n)O(n \log n)O(nlogn) (CORMEN et al., 2009; SEDGEWICK, 2002).

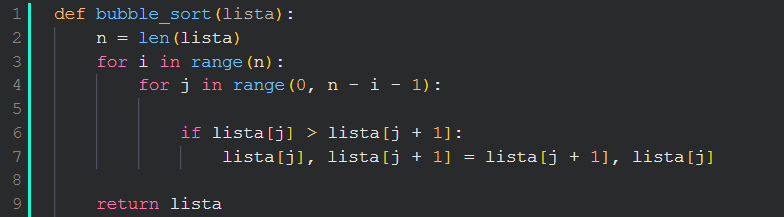
2.4.2 Aplicações do Bubble Sort

O Bubble Sort é geralmente utilizado em contextos educacionais, pois oferece uma introdução intuitiva ao conceito de ordenação e algoritmos iterativos. No entanto, seu uso em sistemas reais é raro devido à sua baixa eficiência em comparação com algoritmos mais avançados. Segundo Menezes (2014), o Bubble Sort pode ser útil em situações onde o número de elementos a ser ordenado é muito pequeno, ou quando já se sabe que a lista está quase ordenada, aproveitando a possibilidade de detecção de uma lista ordenada antes do final do processo.

Em termos práticos, em sistemas computacionais modernos, o Bubble Sort raramente é utilizado diretamente, sendo mais comum em exemplos acadêmicos e didáticos, para explicar a lógica básica de trocas e iterações (NIVENS, 2011).

A implementação do BubbleSort envolve uma função. Abaixo, um exemplo de código Python:

Figura 1 – Implementação BubbleSort



Fonte: Do auto, 2023.

1. Na linha 2, definimos o número de elementos na lista.
2. Na linha 3, criamos um loop externo para passar por toda a lista número de elementos que definimos na linha 2 vezes.
3. Na linha 4, definimos um loop interno para comparar os elementos adjacentes. À medida que i aumenta, não precisamos comparar os últimos elementos, pois eles já estão ordenados, então o limite é 'n - i - 1'.
4. A linha 6 verifica se o elemento atual é maior que o próximo, se for, troca de posição com o próximo elemento como mostrado na linha 7.
5. A linha 9 retorna a lista ordenada após todas as iterações.

2.4.3 Vantagens e Desvantagens

2.4.3.1 Vantagens

1. Simplicidade de implementação: É um dos algoritmos mais simples de ser entendido e implementado, especialmente para iniciantes, o que facilita seu uso em contextos didáticos (NIVENS, 2011).
2. Detecção de lista ordenada: Caso a lista esteja quase ordenada, o Bubble Sort pode encerrar a execução mais cedo, pois ele verifica, em cada passagem, se houve ou não troca de elementos, o que otimiza o tempo de execução em alguns cenários específicos (MENEZES, 2014).

2.4.3.2 Desvantagens

1. Simplicidade de implementação: É um dos algoritmos mais simples de ser entendido e implementado, especialmente para iniciantes, o que facilita seu uso em contextos didáticos (NIVENS, 2011).
2. Detecção de lista ordenada: Caso a lista esteja quase ordenada, o Bubble Sort pode encerrar a execução mais cedo, pois ele verifica, em cada passagem, se houve ou não troca de elementos, o que otimiza o tempo de execução em alguns cenários específicos (MENEZES, 2014).

**2.5 QuickSort**

O QuickSort, ou ordenação rápida, foi desenvolvido por Tony Hoare em 1960. É um algoritmo de ordenação recursivo que utiliza a técnica de divisão e conquista. Seu princípio básico é dividir o array em duas partes com base em um elemento denominado pivô.

2.5.1 Fundamentos do QuickSort

O QuickSort realiza a ordenação da seguinte maneira (HOARE, 1962):

1. Seleciona um pivô (geralmente o primeiro ou último elemento da lista, ou um elemento escolhido aleatoriamente);
2. Organiza os elementos de modo que todos os elementos menores que o pivô fiquem à esquerda, e todos os elementos maiores fiquem à direita;
3. Recursivamente, aplica o mesmo procedimento às sublistas esquerda e direita.

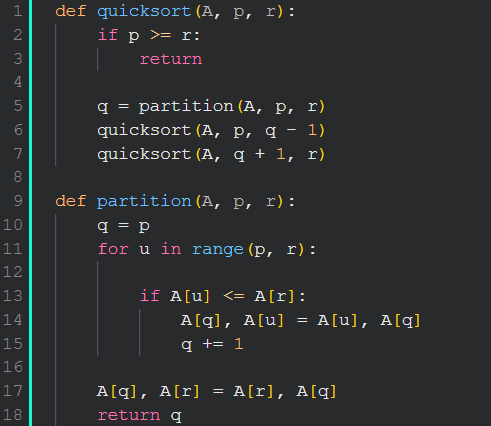
O tempo de execução do QuickSort, no melhor caso e no caso médio, é O(nlog⁡n)O(n \log n)O(nlogn), sendo, portanto, um dos algoritmos de ordenação mais rápidos para a maioria das entradas. No entanto, no pior caso, quando o pivô escolhido não é ideal, a complexidade pode atingir O(n2)O(n^2)O(n2). Para mitigar esse problema, diversas variações do QuickSort empregam técnicas de escolha de pivô mais sofisticadas, como o "pivô aleatório" ou o "pivô mediano" (SEDGEWICK, 2002).

2.5.2 Aplicações do QuickSort

Devido à sua eficiência e flexibilidade, o QuickSort é amplamente utilizado em contextos de ordenação de grandes volumes de dados. Ele é particularmente eficaz em situações em que os dados podem ser carregados na memória principal e a ordenação precisa ser feita rapidamente. Segundo Cormen et al. (2009), o QuickSort é uma escolha preferida em sistemas de alto desempenho, como bancos de dados, sistemas operacionais e aplicações de ciência de dados, onde a velocidade e a eficiência são críticas.

A implementação do QuickSort envolve duas funções. Abaixo, um exemplo de código Python:

Figura 2 – Implementação do QuickSort



Fonte: Do autor, 2024.

1. Na linha 2, na função quicksort, verifica-se se p é maior ou igual a r. Esse é o caso base da recursão: se o subarray contém 0 ou 1 elemento, ele já está ordenado, e a função retorna sem fazer nada.
2. Na linha 5, se o caso base não for atendido, chama-se a função partition para dividir o array em duas partes e obter o índice q, que é a posição correta do pivô.
3. Na linha 6, a função quicksort é chamada recursivamente para a sublista à esquerda do pivô, que vai de p até q - 1.
4. Na linha 7, a função quicksort é chamada novamente para a sublista à direita do pivô, que vai de q + 1 até r.
5. Na linha 10, inicia-se a função partition, onde q é inicialmente definido como p, representando o índice onde o próximo elemento menor ou igual ao pivô será colocado.
6. Na linha 11, um loop percorre os elementos do índice p até r - 1.
7. Na linha 13, verifica-se se o elemento atual A[u] é menor ou igual ao pivô (que é o elemento A[r]). Se for, o código troca A[q] com A[u], colocando A[u] na posição correta de um elemento menor ou igual ao pivô.
8. Na linha 14, ocorre a troca A[q], A[u] = A[u], A[q], e q é incrementado para o próximo índice.
9. Após o loop, na linha 17, o pivô A[r] é trocado com A[q], garantindo que o pivô esteja em sua posição correta no array ordenado.
10. Finalmente, na linha 18, a função partition retorna o índice q, indicando a posição do pivô após a partição.

2.5.3 Vantagens e Desvantagens

O QuickSort se destaca por sua eficiência em média, sua implementação relativamente simples e pelo fato que não requer espaço adicional significativo, uma vez que é um algoritmo in-place. Além disso, ele é amplamente utilizado em várias bibliotecas padrão de linguagens de programação, como o C e o Python, devido à sua performance em entradas diversas (MENEZES, 2014).

No entanto, uma das principais desvantagens do QuickSort é sua instabilidade, já que a ordem relativa de elementos iguais pode não ser preservada após a ordenação. Além disso, como mencionado, no pior caso, seu desempenho pode ser degradado para O(n2)O(n^2)O(n2), mas isso é raramente observado na prática quando implementado com boas heurísticas de escolha de pivô (NIVENS, 2011).

**2.6 InsertionSort**

O InsertionSort é um algoritmo de ordenação que funciona de maneira semelhante à forma como organizamos cartas em um baralho. Ele percorre a lista e insere cada elemento em sua posição correta, movendo para a esquerda os elementos maiores que o elemento atual, até encontrar o ponto exato para inserção. Devido a essa abordagem, ele é considerado eficiente para listas pequenas e parcialmente ordenadas (CORMEN et al., 2009).

2.6.1 Fundamentos do InsertionSort

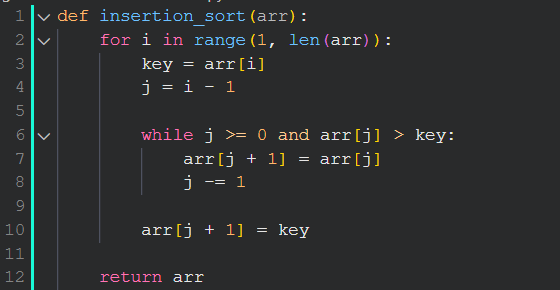
O funcionamento do InsertionSort envolve percorrer a lista de elementos e mover os elementos maiores para a direita, de modo a liberar espaço para o elemento atual. A complexidade do algoritmo no pior caso é O(n2), pois ele pode realizar várias comparações e movimentações para posicionar cada elemento. No entanto, em cenários onde a lista já está parcialmente ordenada, o InsertionSort pode se aproximar de uma complexidade linear, 𝑂(𝑛), tornando-o uma escolha viável para certas aplicações (SEDGEWICK, 2002).

2.6.2 Aplicações do InsertionSort

O InsertionSort é comumente utilizado em contextos que envolvem listas pequenas ou parcialmente ordenadas. Ele é útil, por exemplo, para algoritmos de ordenação híbridos, como o Timsort, que aplicam o InsertionSort em sublistas pequenas para otimizar o desempenho geral do algoritmo. Além disso, devido à sua simplicidade e estabilidade, é usado em ambientes com restrições de memória ou quando o número de elementos a serem ordenados é limitado (KNUTH, 1998).

A implementação do InsertionSort envolve uma função. Abaixo, um exemplo de código Python:

Figura 3 – Implementação InsertionSort



Fonte: Do autor, 2024.

1. Na linha 2, a função insertion\_sort recebe uma lista arr como entrada. Um laço for percorre a lista a partir do segundo elemento (índice 1) até o final, com i representando o índice atual do elemento a ser inserido na posição correta.
2. Na linha 3, a variável key é definida como o valor do elemento atual (arr[i]), que será comparado com os elementos anteriores para encontrar sua posição correta na sublista já ordenada à esquerda.
3. Na linha 4, a variável j é inicializada com o valor de i - 1, representando o índice do elemento imediatamente à esquerda de key.
4. Na linha 6, um laço while começa, que continua enquanto j for maior ou igual a 0 e o valor em arr[j] for maior que key. Esse laço desloca os elementos maiores que key uma posição para a direita, criando espaço para inserir key na posição correta.
5. Na linha 7, arr[j + 1] é atualizado para arr[j], efetivamente deslocando o elemento arr[j] uma posição para a direita. Em seguida, j é decrementado para continuar a verificação com os elementos anteriores.
6. Na linha 10, quando a posição correta de key é encontrada, arr[j + 1] é atualizado para key, inserindo-o no local apropriado da sublista ordenada.
7. Finalmente, na linha 12, a lista arr é retornada já ordenada após todas as iterações do algoritmo.

2.6.3 Vantagens e Desvantagens

Entre as principais vantagens do InsertionSort, destaca-se sua simplicidade e o fato de ser um algoritmo estável, que preserva a ordem dos elementos iguais. Ele também é eficiente para listas pequenas ou quase ordenadas. Contudo, uma das desvantagens é sua ineficiência para listas grandes e desordenadas, devido à sua complexidade quadrática no pior caso. Essas características limitam seu uso em cenários de grandes volumes de dados (CORMEN et al., 2009; SEDGEWICK, 2002).

**2.7 BinaryInsertionSort**

O BinaryInsertionSort é uma variação do InsertionSort que utiliza a busca binária para encontrar a posição correta de inserção de cada elemento. Em vez de realizar uma busca sequencial, ele reduz o número de comparações, dividindo a lista em partes e, com isso, localizando a posição ideal de forma mais rápida. Esse método é mais eficiente em termos de comparações, embora o deslocamento de elementos ainda ocorra (CORMEN et al., 2009).

2.7.1 Fundamentos do BinaryInsertionSort

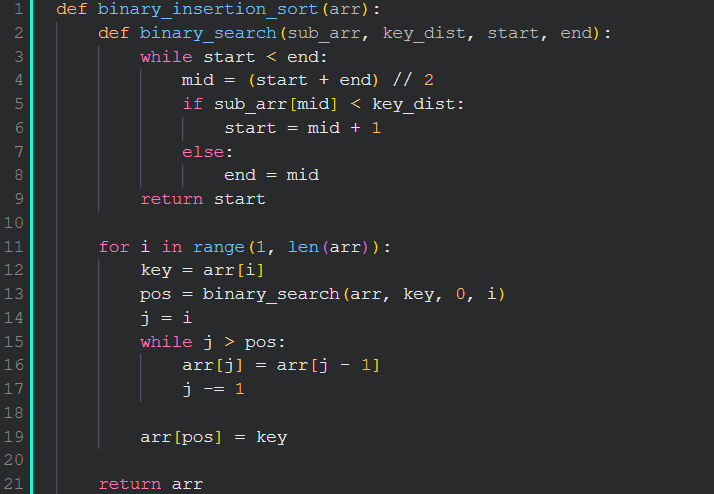
No BinaryInsertionSort, a busca binária é utilizada para reduzir o número de comparações na fase de inserção, enquanto a complexidade de movimentação dos elementos permanece O(n) para cada inserção. Assim, embora o número de comparações seja reduzido para O(logn) em cada etapa, a complexidade geral do algoritmo permanece 𝑂(𝑛2) no pior caso, pois os elementos ainda precisam ser deslocados até suas posições corretas. Esse método é preferido quando a lista já está parcialmente ordenada, o que facilita o deslocamento dos elementos e melhora a eficiência (SEDEGEWICK, 2002).

2.7.2 Aplicações do BinaryInsertionSort

O BinaryInsertionSort é particularmente útil em algoritmos de ordenação híbridos e situações em que a lista é parcialmente ordenada e o número de elementos é relativamente pequeno. Sua utilização é comum em otimizações internas de algoritmos mais complexos, uma vez que combina a busca binária com a simplicidade do InsertionSort, o que pode reduzir o tempo de ordenação em situações específicas, como em implementações de ordenação de blocos menores dentro de algoritmos híbridos (KNUTH, 1998).

A implementação do BinaryInsertionSort envolve duas funções, sendo uma delas interna. Abaixo, um exemplo de código Python:

Figura 4 – Implementação BinaryInsertionSort



Fonte: Do autor, 2024.

1. Na linha 2, a função binary\_insertion\_sort recebe uma lista arr como entrada. Uma função interna binary\_search é definida para encontrar a posição correta onde o elemento key deve ser inserido na sublista já ordenada.
2. Na linha 3, a função binary\_search realiza uma busca binária para localizar a posição onde key deve ser inserido entre os índices start e end em sub\_arr. Esse método reduz o tempo de busca pela posição correta de inserção comparado à busca linear.
3. Na linha 4, a variável mid é definida como o ponto médio entre start e end. Este valor é usado para dividir a lista em duas partes durante a busca binária.
4. Na linha 5, o valor em sub\_arr[mid] é comparado com key\_dist (o valor de key). Se sub\_arr[mid] for menor que key\_dist, o início da busca (start) é atualizado para mid + 1, restringindo a busca para a metade direita.
5. Na linha 7, se sub\_arr[mid] não for menor que key\_dist, o fim da busca (end) é atualizado para mid, restringindo a busca para a metade esquerda. O processo se repete até encontrar a posição correta.
6. Na linha 11, o laço for percorre a lista a partir do segundo elemento (índice 1) até o final. key é definido como arr[i], e pos é calculado chamando binary\_search para encontrar a posição onde key deve ser inserido.
7. Na linha 14, a variável j é inicializada com o valor de i, e um laço while é usado para deslocar os elementos maiores que key uma posição para a direita, a fim de abrir espaço para inserir key na posição correta.
8. Na linha 16, arr[j] é atualizado para arr[j - 1], deslocando o elemento para a direita. j é decrementado até que key seja inserido na posição correta (pos).
9. Na linha 19, o valor key é inserido em arr[pos], concluindo a inserção na sublista ordenada.
10. Finalmente, na linha 21, a lista arr é retornada já ordenada após todas as iterações do algoritmo.

2.7.3 Vantagens e Desvantagens

O BinaryInsertionSort possui como principal vantagem a redução no número de comparações ao empregar a busca binária, o que o torna ligeiramente mais rápido do que o InsertionSort tradicional em certas situações. No entanto, sua desvantagem principal é que, apesar da busca binária, ele ainda possui uma complexidade quadrática, O(n2)O(n^2)O(n2), devido ao custo de movimentação de elementos. Esse fator o torna inadequado para listas grandes e desordenadas, limitando seu uso a casos específicos (CORMEN et al., 2009; SEDGEWICK, 2002).

**2.8 SelectionSort**

O SelectionSort é um método de ordenação por seleção, onde é feito uma troca de elementos das matrizes. O método é pegar o menor elemento e trocar sua posição com o primeiro, após isso, o próximo menor elemento troca de posição com o segundo e assim continua até que o algoritmo seja ordenado corretamente, podendo ser ordenado de forma crescente ou decrescente.

2.8.1 Fundamentos do SelectionSort

2.8.1.1 Processo de Seleção e Troca

O Selection Sort realiza a ordenação selecionando, a cada passo, o menor elemento da porção não ordenada do array e o trocando com o primeiro elemento não ordenado. Este processo de seleção e troca é repetido até que todos os elementos estejam em suas posições corretas. Weiss (2012, p. 42) explica que "a simplicidade do algoritmo advém de sua estratégia de seleção direta, em que cada elemento é colocado diretamente em sua posição final após ser encontrado."

2.8.1.2 Complexidade Computacional

A complexidade de tempo do Selection Sort é O(n²) para o melhor, o pior e o caso médio. Esse comportamento quadrático ocorre porque o algoritmo percorre toda a lista para encontrar o menor elemento em cada iteração. Knuth (1998, p. 130) descreve que "o Selection Sort é consistentemente O(n²), independentemente da organização inicial dos elementos, o que limita sua aplicabilidade para grandes conjuntos de dados."

2.8.1.3 Comparações e Trocas

No Selection Sort, a quantidade de comparações realizadas é consistente, já que ele sempre percorre a lista completa para encontrar o menor elemento. No entanto, o número de trocas é menor em comparação com outros algoritmos quadráticos, como o Bubble Sort. Sedgewick e Wayne (2011, p. 78) comentam que "apesar do número elevado de comparações, o Selection Sort tem o benefício de realizar um número limitado de trocas, o que o torna interessante em cenários onde as trocas de elementos são mais custosas que as comparações."

2.8.2 Aplicações do Selection Sort

Embora o Selection Sort não seja o algoritmo de ordenação mais eficiente, ele possui algumas aplicações específicas em que sua simplicidade e baixo número de trocas o tornam útil. Segundo Cormen et al. (2009, p. 103), o Selection Sort "é adequado para listas pequenas e em cenários onde a simplicidade do código e a previsibilidade do desempenho são mais importantes que a eficiência."

2.8.2.1 Ordenação de Pequenos Conjuntos de Dados

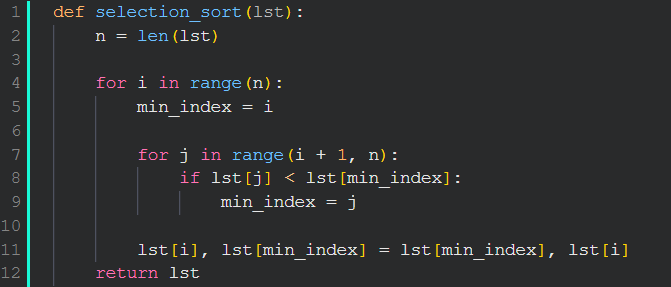
O Selection Sort é frequentemente utilizado para ordenar pequenos conjuntos de dados ou em contextos educacionais, onde sua lógica simples ajuda a ilustrar os princípios básicos de ordenação. Weiss (2012, p. 45) destaca que "em conjuntos de dados pequenos, o Selection Sort oferece uma solução direta e fácil de implementar, com desempenho aceitável."

2.8.2.2 Uso em Sistemas com Restrição de Trocas

Em sistemas onde a troca de elementos é uma operação custosa, o Selection Sort pode ser preferível ao Bubble Sort, pois, embora tenha a mesma complexidade, realiza menos trocas. Knuth (1998, p. 132) comenta que "o baixo número de trocas do Selection Sort o torna vantajoso em ambientes onde a movimentação de elementos é restrita ou limitada por custos."

A implementação do SelectionSort envolve uma função. Abaixo, um exemplo de código Python:

Figure 5 – Implementação do SelectionSort



Fonte: Do autor, 2024.

1. Na linha 2, a variável n é definida como o comprimento da lista lst, representando o número de elementos na lista.
2. Na linha 4, o primeiro loop for percorre a lista, de 0 até n - 1. Em cada iteração, o índice i representa a posição atual que será preenchida com o menor valor encontrado no restante da lista.
3. Na linha 5, o índice min\_index é inicializado com o valor de i, assumindo que o menor elemento do subarray começa nessa posição.
4. Na linha 7, o segundo loop for percorre o subarray que começa em i + 1 até o final da lista (n), buscando o menor valor entre esses elementos.
5. Na linha 8, verifica-se se o elemento lst[j] é menor do que o valor em lst[min\_index]. Se for, min\_index é atualizado com o índice j, indicando a nova posição do menor elemento encontrado.
6. Na linha 11, ao final do segundo loop, ocorre a troca entre lst[i] e lst[min\_index], posicionando o menor elemento encontrado na posição i.
7. Finalmente, na linha 12, a lista lst já ordenada é retornada após todas as iterações.

2.8.3 Vantagens e Desvantagens

O SelectionSort é simples e fácil de implementar, porém não muito eficaz para listas grandes, algo também observado por Thomas Cormen, autor do livro Desmitificando Algritmos, de acordo com sua analise “A primeira é que veremos que seu tempo de execução assintótico de Θ (n2) é o pior dos algoritmos de ordenação que examinaremos.” (Desmitificando Algoritmos, p.29).

**2.9 HeapSort**

Diferente do método ‘SelectionSort’, o método ‘HeapSort’ não faz comparações entre o menor elemento com os maiores, mas sim do maior, reorganizando de uma forma que o maior elemento fique na última posição e o próximo maior, caso exista, depois dele ocupando assim a última posição, continuando até que a lista esteja ordenada.

Esse método usa uma estrutura de dados ‘Binary Heap’ que simboliza uma árvore binária, onde o maior elemento é a raiz dessa árvore, os elementos filhos a esquerda são verificados para saber se são maiores que a raiz, os filhos a direita são verificados para saber se são maiores que o maior elemento até agora. A prioridade é levar o maior elemento para esquerda primeiramente.

2.9.1 Fundamentos do HeapSort

2.9.1.1 Estrutura de Dados Heap

O HeapSort depende da estrutura heap, na qual o elemento raiz é sempre o maior (em um max-heap) ou o menor (em um min-heap). A propriedade de ordem do heap permite que o algoritmo selecione repetidamente o maior ou menor elemento, facilitando a construção de uma lista ordenada. Weiss (2012, p. 210) afirma que "a estrutura de heap fornece um método eficaz para manter e acessar o elemento de maior prioridade, fundamental para o HeapSort."

2.9.1.2 Construção do Heap

Para aplicar o HeapSort, o primeiro passo é construir o max-heap a partir do array não ordenado. Esse processo transforma o array em uma estrutura onde o maior elemento se encontra na raiz, o que facilita a ordenação subsequente. De acordo com Knuth (1998, p. 176), "a construção de um max-heap pode ser feita em tempo linear, garantindo a eficiência da primeira etapa do HeapSort."

2.9.1.3 Remoção do Maior Elemento

Após construir o max-heap, o algoritmo remove o maior elemento (a raiz) e o coloca no final do array. Em seguida, o heap é reestruturado para que o próximo maior elemento se torne a nova raiz, e o processo é repetido até que todos os elementos estejam ordenados. Sedgewick e Wayne (2011, p. 235) descrevem que "o HeapSort utiliza a estrutura de heap de forma a garantir a extração ordenada dos maiores elementos sem a necessidade de armazenamento extra."

2.9.2 Aplicações do HeapSort

2.9.2.1 Ordenação em Sistemas de Baixa Memória

O HeapSort é particularmente útil em sistemas com restrições de memória, já que sua ordenação in-place o torna mais eficiente do que algoritmos que requerem armazenamento auxiliar. Segundo Knuth (1998, p. 178), "o HeapSort oferece uma solução de ordenação eficaz para ambientes com memória limitada, onde outras técnicas de ordenação poderiam não ser viáveis."

2.9.2.2 Sistemas de Prioridade

A estrutura de heap permite a implementação eficiente de filas de prioridade, tornando o HeapSort uma base para sistemas que exigem uma classificação rápida e constante dos maiores ou menores elementos. Cormen et al. (2009, p. 166) observam que "a estrutura do heap é ideal para cenários de filas de prioridade, onde o acesso ao elemento de maior prioridade deve ser mantido de forma rápida."

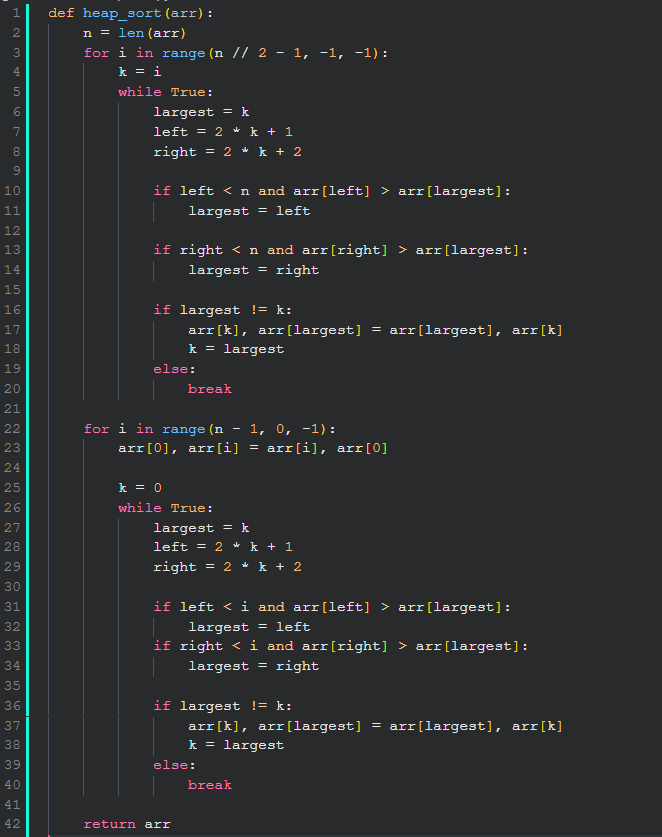
2.9.3 Vantagens e Desvantagens

O Heap Sort tem uma complexidade de tempo de O(n log n) no pior caso, o que o torna eficiente para ordenar listas grandes e utiliza uma estrutura de dados chamada heap, que é útil em várias aplicações, como gerenciamento de prioridades.

Porém, comparado a outros algoritmos como Quick Sort e Bubble Sort, o Heap Sort pode ser menos intuitivo e mais difícil de entender, especialmente na construção e manutenção da estrutura de heap. Embora a complexidade teórica seja O(n log n), na prática, o Heap Sort pode ser mais lento que outros algoritmos de ordenação, como Quick Sort, devido

A implementação do HeapSort envolve uma função. Abaixo, um exemplo de código Python:

Figura 6 – Implementação do HeapSort



Fonte: Do autor, 2024.

1. Na linha 2, n é definida como o comprimento do array arr, representando o número de elementos no array.
2. Na linha 3, o primeiro for percorre metade da lista de trás para frente, construindo um heap máximo (max-heap) a partir do meio do array até o início.
3. Na linha 4, a variável k é inicializada como i, representando o índice atual do elemento que está sendo ajustado no heap.
4. Na linha 5, o while interno começa o ajuste do heap a partir do índice k.
5. Nas linhas 6 a 8, as variáveis left e right são calculadas como os índices dos filhos esquerdo e direito de k, respectivamente.
6. Na linha 10, verifica-se se o filho esquerdo (left) está dentro do limite do array (n) e se seu valor é maior que o elemento atual (arr[largest]). Se for, largest é atualizado para o índice do filho esquerdo.
7. Na linha 13, o mesmo é feito para o filho direito (right). Se o filho direito tiver um valor maior que o elemento atual em arr[largest], largest é atualizado com o índice do filho direito.
8. Na linha 16, se largest foi atualizado (ou seja, largest != k), ocorre uma troca entre arr[k] e arr[largest], movendo o maior elemento para a posição correta no heap. A variável k é atualizada para largest, e o processo continua para ajustar o sub-heap.
9. Na linha 18, se largest é igual a k, o while é interrompido, pois o heap está ajustado.
10. Na linha 22, o segundo for percorre o array de trás para frente, extraindo o maior elemento (a raiz do heap) e movendo-o para o final do array.
11. Na linha 23, ocorre a troca entre arr[0] (raiz do heap) e arr[i], colocando o maior elemento na sua posição final.
12. Na linha 26, o while interno começa a reorganizar o heap para o restante do array, agora com tamanho reduzido.
13. Nas linhas 31 a 40, o mesmo processo de ajuste do heap é repetido, ajustando largest, left, right e realizando trocas se necessário, até que o heap esteja novamente ordenado.
14. Na linha 42, o array arr ordenado é retornado após todas as operações de construção e ajuste do heap.

**2.10 MergeSort**

O MergeSort é um algoritmo de ordenação baseado no paradigma de dividir para conquistar (*divide and conquer*), que foi inicialmente descrito por John von Neumann em 1945. O algoritmo divide repetidamente a lista ou array em duas metades, ordena cada metade de maneira recursiva e, em seguida, combina as duas metades para produzir uma lista final ordenada. Segundo Cormen et al. (2009, p. 34), "MergeSort é um algoritmo eficiente de ordenação com complexidade de tempo O(n log n) em todos os casos, sendo também estável."

2.10.1 Fundamentos do MergeSort

O algoritmo merge sort divide, recursivamente, o array em duas partes até que cada posição dele seja considerada como um array de um único elemento. Em seguida, o algoritmo combina dois arrays de forma a obter um array maior e ordenado. Essa combinação dos arrays é feita intercalando seus elementos de acordo com o sentido da ordenação (crescente ou decrescente). Esse processo se repete até que exista apenas um array. (BACKES, André, 2022, p. 34)

2.10.1.1 Divisão e Conquista

O MergeSort exemplifica o paradigma de dividir para conquistar, onde um problema é decomposto em partes menores e mais gerenciáveis. Cada subproblema é resolvido de forma recursiva, e as soluções dessas partes são combinadas. Conforme Sedgewick e Wayne (2011, p. 145), o princípio fundamental do MergeSort está na "divisão da lista original em sublistas até que o problema seja trivial, ou seja, cada sublista contenha apenas um elemento."

2.10.1.2 Recursividade

O uso de recursividade é essencial para a operação do MergeSort. O processo de divisão recursiva continua até que as sublistas tenham apenas um elemento. De acordo com Weiss (2012, p. 78), "a recursividade do MergeSort permite que listas de qualquer tamanho sejam divididas até o ponto onde a fusão seja direta e rápida."

2.10.1.3 Complexidade Computacional

A complexidade de tempo do MergeSort é O(n log n**)** no melhor, pior e caso médio, o que o torna um dos algoritmos mais consistentes. Knuth (1998, p. 172) observa que "a eficiência do MergeSort em tempo O(n log n) se deve à divisão da lista em duas metades, seguida da combinação das duas listas ordenadas em tempo linear."

2.10.1.4 Estabilidade

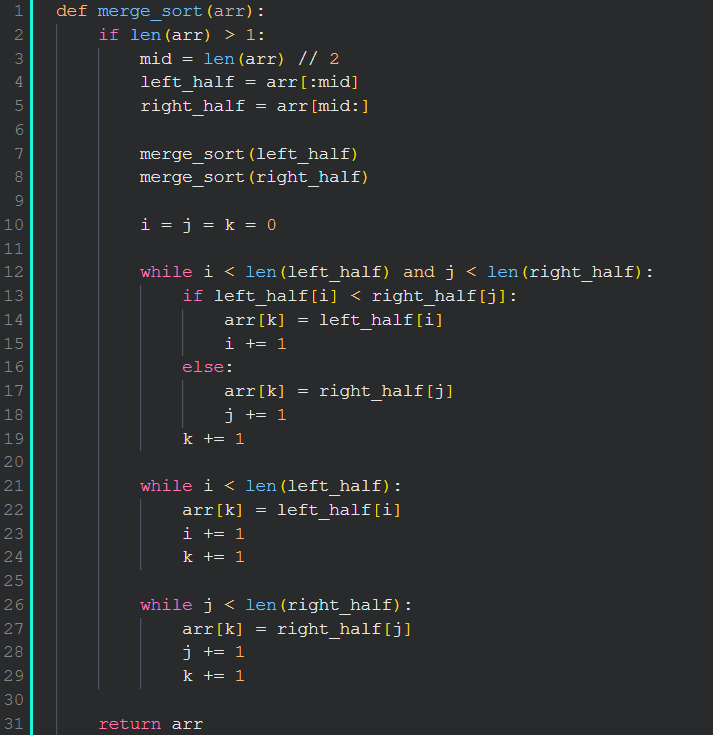
A estabilidade do MergeSort é uma de suas características mais notáveis. Algoritmos estáveis de ordenação, como o MergeSort, garantem que a ordem relativa de elementos iguais seja mantida após a ordenação. Cormen et al. (2009, p. 39) ressaltam que "a estabilidade do MergeSort o torna útil em situações em que a preservação da ordem original é desejável."

2.10.2 Aplicações do MergeSort

O MergeSort é amplamente utilizado em sistemas que lidam com grandes volumes de dados. Ele é especialmente eficaz em situações onde os dados precisam ser processados em memória externa, como discos rígidos, uma vez que pode lidar eficientemente com grandes conjuntos de dados que não cabem na memória RAM. Knuth (1998, p. 181) afirma que "o MergeSort é particularmente útil em sistemas de arquivos externos e grandes bases de dados, onde a eficiência da memória é crucial."

A implementação do MergeSort envolve uma função. Abaixo, um exemplo de código Python:

Figura 7 – Implementação do Mergesort



Fonte: Do autor, 2024.

1. Na linha 2, a função verifica se o comprimento da lista arr é maior que 1, pois listas de tamanho 1 ou 0 já estão ordenadas.
2. Na linha 3, a variável mid é definida como o ponto médio da lista, dividindo a lista em duas metades.
3. Na linha 4, left\_half recebe a primeira metade da lista, indo do início até mid.
4. Na linha 5, right\_half recebe a segunda metade da lista, indo de mid até o final.
5. Na linha 7, a função merge\_sort é chamada recursivamente para ordenar a metade esquerda (left\_half).
6. Na linha 8, a função merge\_sort é chamada recursivamente para ordenar a metade direita (right\_half).
7. Na linha 10, as variáveis i, j e k são inicializadas com 0. Elas serão usadas como índices para percorrer left\_half, right\_half e arr, respectivamente.
8. Na linha 12, um while é iniciado para percorrer ambas as metades ao mesmo tempo, enquanto houver elementos em left\_half e right\_half.
9. Na linha 13, verifica-se se o elemento atual de left\_half (left\_half[i]) é menor que o elemento atual de right\_half (right\_half[j]). Se for, o elemento é copiado para arr[k], e os índices i e k são incrementados.
10. Na linha 16, caso contrário, o elemento de right\_half[j] é copiado para arr[k], e os índices j e k são incrementados.
11. Na linha 21, outro while percorre os elementos restantes de left\_half (caso algum tenha sobrado), copiando-os para arr.
12. Na linha 26, um terceiro while percorre os elementos restantes de right\_half (caso algum tenha sobrado), copiando-os para arr.
13. Finalmente, na linha 31, a lista arr é retornada já ordenada após todas as chamadas recursivas e fusões.

2.10.3 Vantagens e Desvantagens

2.10.3.1 Vantagens

Entre as principais vantagens do MergeSort, destacam-se a estabilidade e a complexidade de tempo previsível. Cormen et al. (2009, p. 50) apontam que "um dos principais benefícios do MergeSort é a sua estabilidade e a consistência da complexidade O(n log n) em todos os casos."

2.10.3.2 Desvantagens

Um dos pontos negativos do MergeSort é o uso de memória adicional. Durante o processo de fusão, o algoritmo precisa de memória extra para armazenar as sublistas temporárias. Weiss (2012, p. 91) ressalta que "o principal problema do MergeSort é a necessidade de espaço adicional de O(n), o que pode limitar sua aplicabilidade em sistemas com restrições severas de memória.".

**2.11 BucketSort**

O BucketSort é um algoritmo de ordenação que distribui os elementos de uma lista em diferentes baldes (*buckets*), os quais são posteriormente ordenados individualmente e combinados para produzir a lista final ordenada. Esse algoritmo é eficaz quando os elementos são distribuídos de forma relativamente uniforme dentro de um intervalo finito. Segundo Cormen et al. (2009, p. 201), o BucketSort "funciona bem para dados que podem ser uniformemente distribuídos em intervalos, pois aproveita essa propriedade para organizar os elementos em grupos, simplificando o processo de ordenação."

2.11.1 Fundamentos do BucketSort

2.11.1.1 Organização em Baldes

O principal conceito do BucketSort é a distribuição dos elementos em baldes, ou recipientes, que representam intervalos definidos. Cada balde contém elementos que pertencem a um determinado intervalo, facilitando a ordenação local em cada balde. Conforme Sedgewick e Wayne (2011, p. 210), "a eficiência do BucketSort reside na capacidade de dividir os dados em subconjuntos menores, que podem ser ordenados independentemente."

2.11.1.2 Utilização de Algoritmos Auxiliares

Após distribuir os elementos nos baldes, o algoritmo utiliza um método auxiliar, como o InsertionSort ou o QuickSort, para ordenar os elementos em cada balde. De acordo com Weiss (2012, p. 120), "o BucketSort se beneficia de algoritmos de ordenação secundários, que podem ser aplicados aos baldes com poucos elementos, onde algoritmos como o InsertionSort podem ser extremamente eficazes."

2.11.1.3 Complexidade Computacional

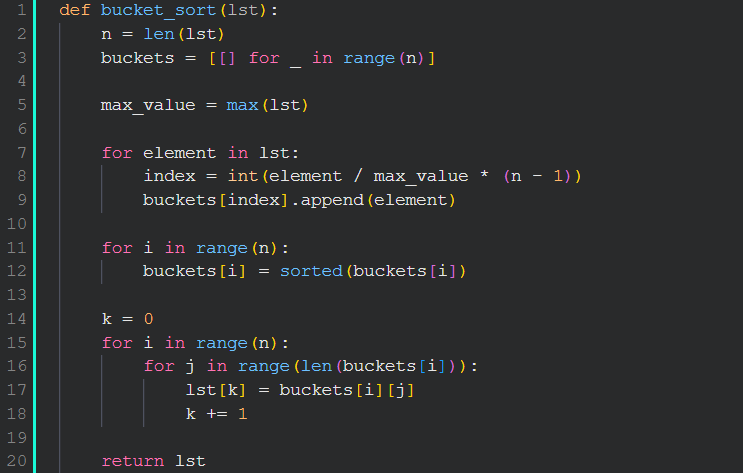
A complexidade de tempo do BucketSort depende da distribuição uniforme dos elementos e da eficiência do algoritmo auxiliar usado em cada balde. No melhor caso, a complexidade é O(n), enquanto no pior caso pode ser O(n²), se todos os elementos forem colocados em um único balde. Knuth (1998, p. 256) ressalta que "a complexidade do BucketSort pode ser extremamente favorável em situações onde a distribuição dos dados é homogênea, oferecendo um desempenho linear."

2.11.2 Aplicações do BucketSort

O BucketSort é amplamente utilizado em aplicações onde os dados são distribuídos uniformemente e podem ser mapeados para intervalos finitos. É comumente aplicado em sistemas que lidam com números flutuantes, uma vez que sua divisão em intervalos facilita o processamento. Knuth (1998, p. 259) aponta que "em sistemas de análise estatística e gráficos de dispersão, o BucketSort demonstra seu valor, principalmente ao ordenar grandes conjuntos de dados numéricos."

A implementação do BucketSort envolve uma função. Abaixo, um exemplo de código Python:

Figura 8 – Implementação do BucketSort



Fonte: Do autor, 2024.

1. Na linha 2, a função bucket\_sort recebe uma lista lst como entrada e determina seu comprimento n, que representa o número de elementos na lista.
2. Na linha 3, cria-se uma lista chamada buckets contendo n listas vazias (buckets), onde cada bucket será utilizado para agrupar elementos da lista de acordo com seu valor.
3. Na linha 5, max\_value é definido como o valor máximo da lista lst, que será usado para calcular o índice do bucket em que cada elemento será inserido.
4. Na linha 7, inicia-se um for que percorre cada element em lst. Para cada elemento, calcula-se index, que representa o índice do bucket onde o elemento será inserido. Este índice é determinado pela divisão do elemento pelo valor máximo max\_value, multiplicado por (n - 1) e convertido em um inteiro.
5. Na linha 8, o elemento element é adicionado ao bucket apropriado dentro de buckets, com base no índice index calculado.
6. Na linha 10, outro for percorre cada bucket em buckets. Cada bucket é ordenado individualmente usando a função sorted, para garantir que os elementos dentro de cada bucket estejam em ordem.
7. Na linha 13, a variável k é inicializada com 0. Ela será utilizada como índice para reconstruir a lista lst com os elementos já ordenados.
8. Na linha 14, um laço for percorre cada bucket em buckets. Para cada bucket, um segundo laço for percorre cada elemento j dentro do bucket.
9. Na linha 16, cada elemento j é colocado na posição k de lst, e o índice k é incrementado para a próxima posição.
10. Finalmente, na linha 19, a lista lst é retornada já ordenada após o processo de ordenação por bucket.

2.11.3 Vantagens e Desvantagens

2.11.3.1 Vantagens

Entre as principais vantagens do BucketSort, destaca-se a sua eficiência em cenários onde os dados são distribuídos uniformemente. Além disso, ao dividir o problema em subproblemas menores, o algoritmo pode se beneficiar de algoritmos mais simples e eficientes em casos pequenos. Cormen et al. (2009, p. 215) afirmam que "o BucketSort pode ser altamente eficiente em situações onde os dados possuem uma distribuição uniforme, o que possibilita um desempenho linear."

2.11.3.2 Desvantagens

Uma desvantagem significativa do BucketSort é a sua dependência da distribuição uniforme dos dados. Caso os elementos estejam concentrados em um pequeno número de baldes, o algoritmo pode perder sua eficiência e apresentar desempenho semelhante ao do InsertionSort. Weiss (2012, p. 123) alerta que "a eficiência do BucketSort depende fortemente da capacidade de distribuir os elementos de forma equitativa entre os baldes, o que pode não ocorrer em todos os casos práticos."

# 3 DESENVOLVIMENTO

# 4 RESULTADOS E DISCUSSÃO

# 5 CONSIDERAÇÕES FINAIS

**BIBLIOGRAFIA**

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009.

KNUTH, D. E. *The Art of Computer Programming: Sorting and Searching*. 2nd ed. Reading: Addison-Wesley, 1998.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4th ed. Reading: Addison-Wesley, 2011.

WEISS, M. A. *Data Structures and Algorithm Analysis in C*. 4th ed. Boston: Pearson, 2012.

Drozdek, A. Estrutura de Dados e Algoritmos em C++: Tradução da 4ª Edição Norte-Americana. 2ª Edição. São Paulo: Cengage Learning, 2017.

CORMEN, Thomas H. et al. Algoritmos: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2009.

HOARE, Charles Antony Richard. Quicksort. *The Computer Journal*, v. 5, n. 1, p. 10-15, 1962.

MENEZES, Ricardo. Algoritmos de ordenação: uma abordagem didática. São Paulo: Editora XYZ, 2014.

NIVENS, Scott. Sorting Algorithms Explained. Porto Alegre: Novatec, 2011.

SEDGEWICK, Robert. Algoritmos em C. 3. ed. São Paulo: Addison-Wesley, 2002.

Drozdek, A. Estrutura de Dados e Algoritmos em C++: Tradução da 4ª Edição Norte-Americana. 2ª Edição. São Paulo: Cengage Learning, 2017.

ALVES, João; DIAS, Maria. Estruturas de Dados e Algoritmos. 3. ed. São Paulo: Atlas, 2019. p. 35.

BARBOSA, Lucas. Introdução à Ciência da Computação. Rio de Janeiro: LTC, 2021. p. 42.

FERNANDES, Tiago. Algoritmos e Estruturas de Dados: uma abordagem prática. Curitiba: Intersaberes, 2018. p. 29.

OLIVEIRA, Pedro; SANTOS, Luana. Fundamentos de Algoritmos. Porto Alegre: Bookman, 2020. p. 67.

VIEIRA, Antonio; COSTA, Júlia. Introdução à Programação. Belo Horizonte: Editora UFMG, 2017. p. 53.

**CÓDIGO FONTE**

# CARÔMETRO

|  |  |  |  |
| --- | --- | --- | --- |
| Samara dos Santos Sales | Victor Hugo de Oliveira Penga | Roberta Teixeira Silva Costa | Rafael Milani |
|  |  |  |  |