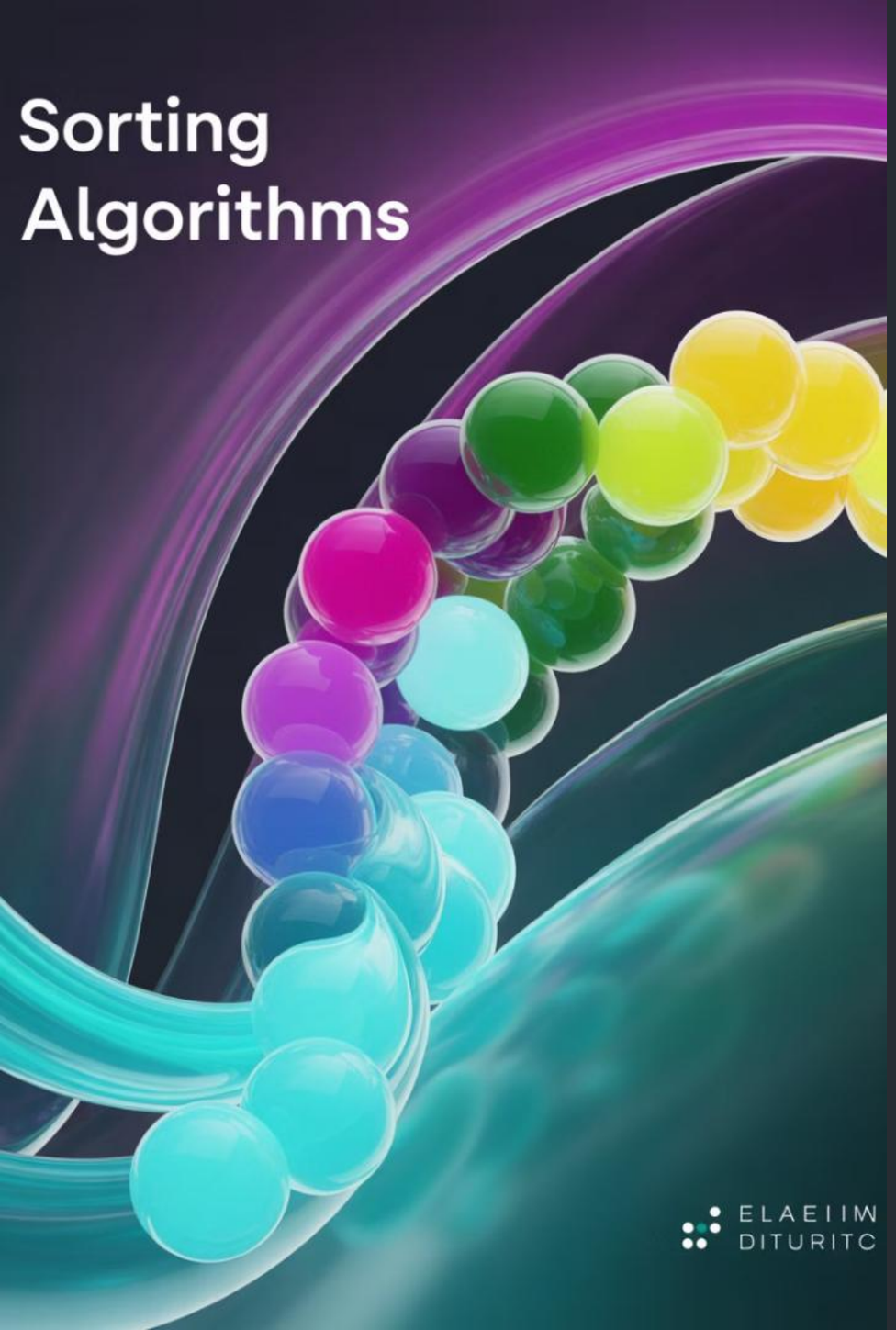


Algoritmos de Ordenação e Pesquisa: Uma Jornada Prática

Explore os fundamentos dos algoritmos que organizam e localizam dados. Uma jornada completa pelos métodos mais importantes da computação.

por Mayana Duarte

Por Que Ordenar e Pesquisar?

Organização Eficiente

Facilita organização e busca de dados em sistemas complexos.

Base dos Sistemas

Fundamental em sistemas, bancos de dados e aplicações modernas.

Aplicação Universal

Utilizado do comércio eletrônico à biomedicina e pesquisa científica.

Situando o Problema

Necessidade Diária

Listas ordenadas são essenciais no cotidiano. Facilitam localização e análise de informações.

Exemplos Práticos

Ranking de alunos, agenda telefônica, catálogos de produtos. Todos dependem de ordenação.

Definição

Ordenar significa reorganizar elementos segundo critério específico. Base para organização eficiente.

Objetivos da Ordenação e Pesquisa

Consultas Rápidas

Aumentar eficiência nas consultas de dados.

Melhor Apresentação

Melhorar apresentação e visualização dos dados.

Operações Otimizadas

Otimizar análises e operações computacionais.

Tipos Comuns de Dados

Dados Numéricos

Números inteiros e reais. Dados quantitativos fundamentais em análises matemáticas.

Dados Textuais

Palavras como nomes, cidades, produtos. Essenciais para organização alfabética.

Objetos Complexos

Estruturas mais elaboradas. Combinam múltiplos tipos de dados.

Aplicações Reais

Apps Móveis

Listas de contatos organizadas alfabeticamente em aplicações móveis.

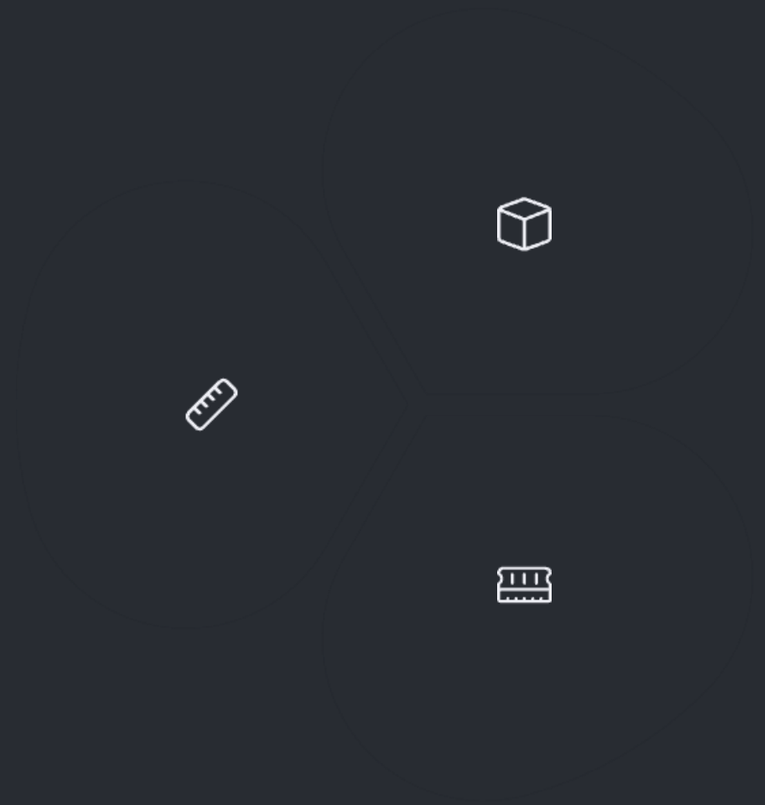
Busca Web

Ordenação de resultados de busca por relevância na internet.

Classificações

Ranking de notas em concursos e avaliações acadêmicas.

Como Escolher um Algoritmo?



Tamanho dos Dados

Volume do conjunto determina método apropriado.

Estrutura

Dados simples ou complexos requerem abordagens diferentes.

Recursos

Restrições de memória e processamento influenciam escolha.

Ordenação: Conceito Básico

1. **Estado Inicial**

   Elementos em ordem aleatória ou não desejada.

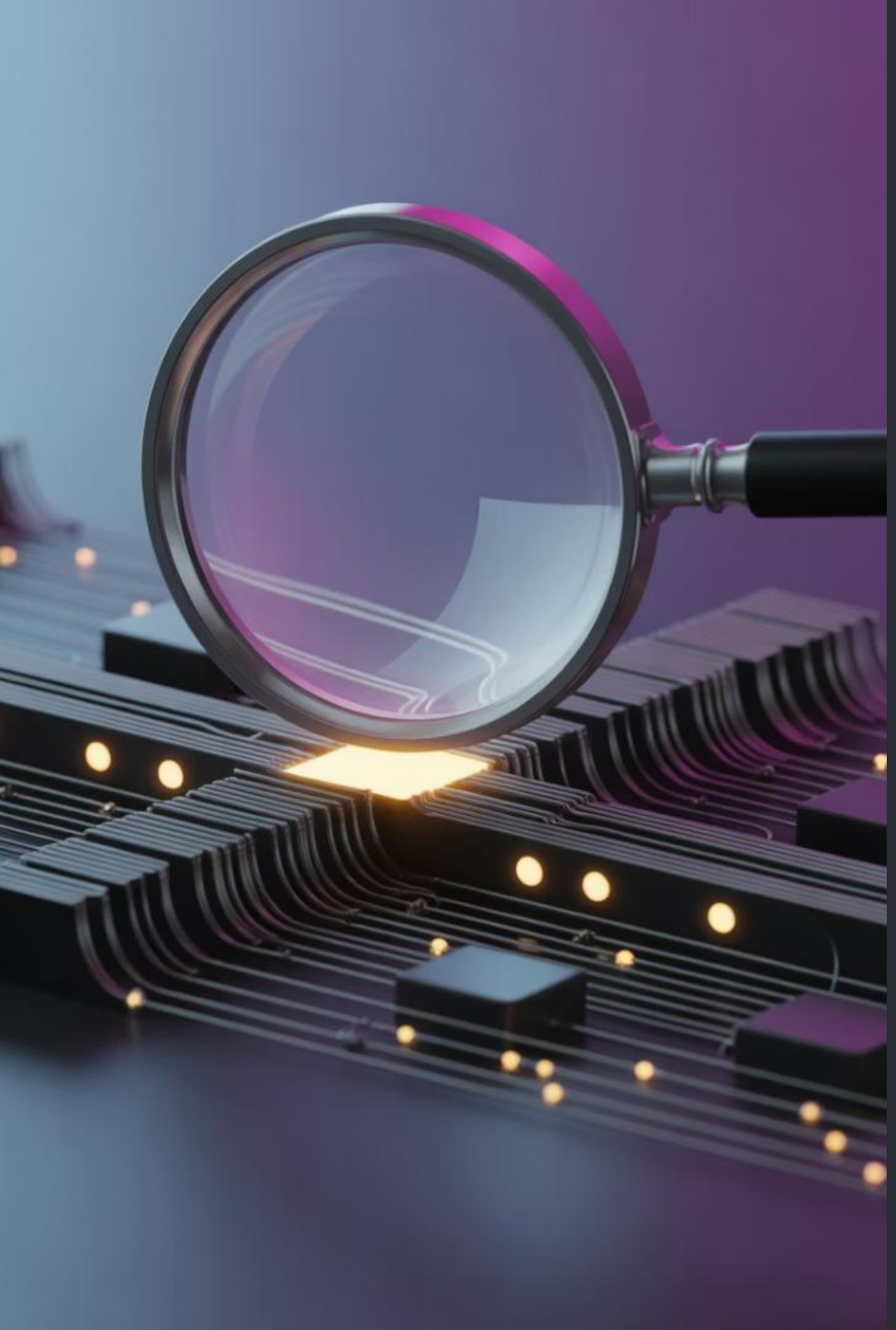
2. **Processo**

   Aplicação de algoritmo de ordenação específico.

3. **Resultado**

   Elementos organizados crescente ou decrescentemente.

Efficiency through order



Pesquisa: Conceito Básico

Definir Elemento

Especificar o valor que se deseja localizar na estrutura.

Aplicar Algoritmo

Usar método de busca apropriado para o tipo de dados.

Obter Resultado

Localizar posição ou confirmar ausência do elemento.

Algoritmos Apresentados

1

Avançados

Heap, Quick, Merge Sort

2

Intermediários

Shell, Comb Sort

3

Básicos

Bubble, Insertion, Selection

4

Especializados

Counting, Bucket, Radix

Bubble Sort: Introdução

1

Simplicidade

Um dos métodos mais simples de implementar.

2

Comparação

Compara pares de elementos consecutivos sistematicamente.

3

Fundamento

Base para entender algoritmos mais complexos.

Bubble Sort: Funcionamento Passo a Passo

Percorrer

Percorre a lista múltiplas vezes sequencialmente.

Comparar

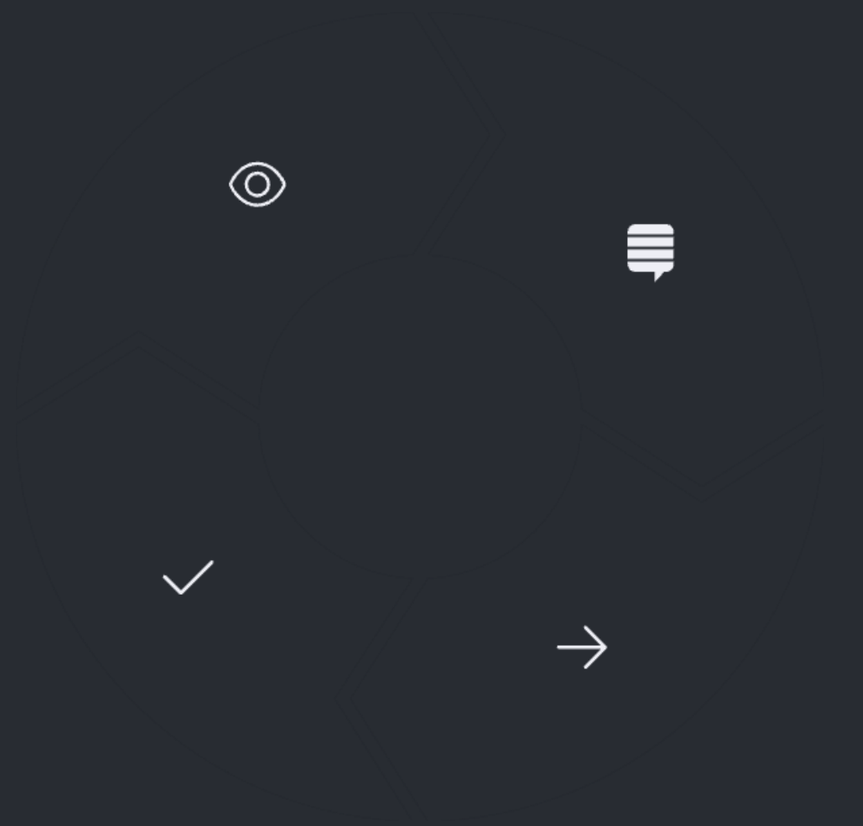
Compara elementos adjacentes em cada passagem.

Repetir

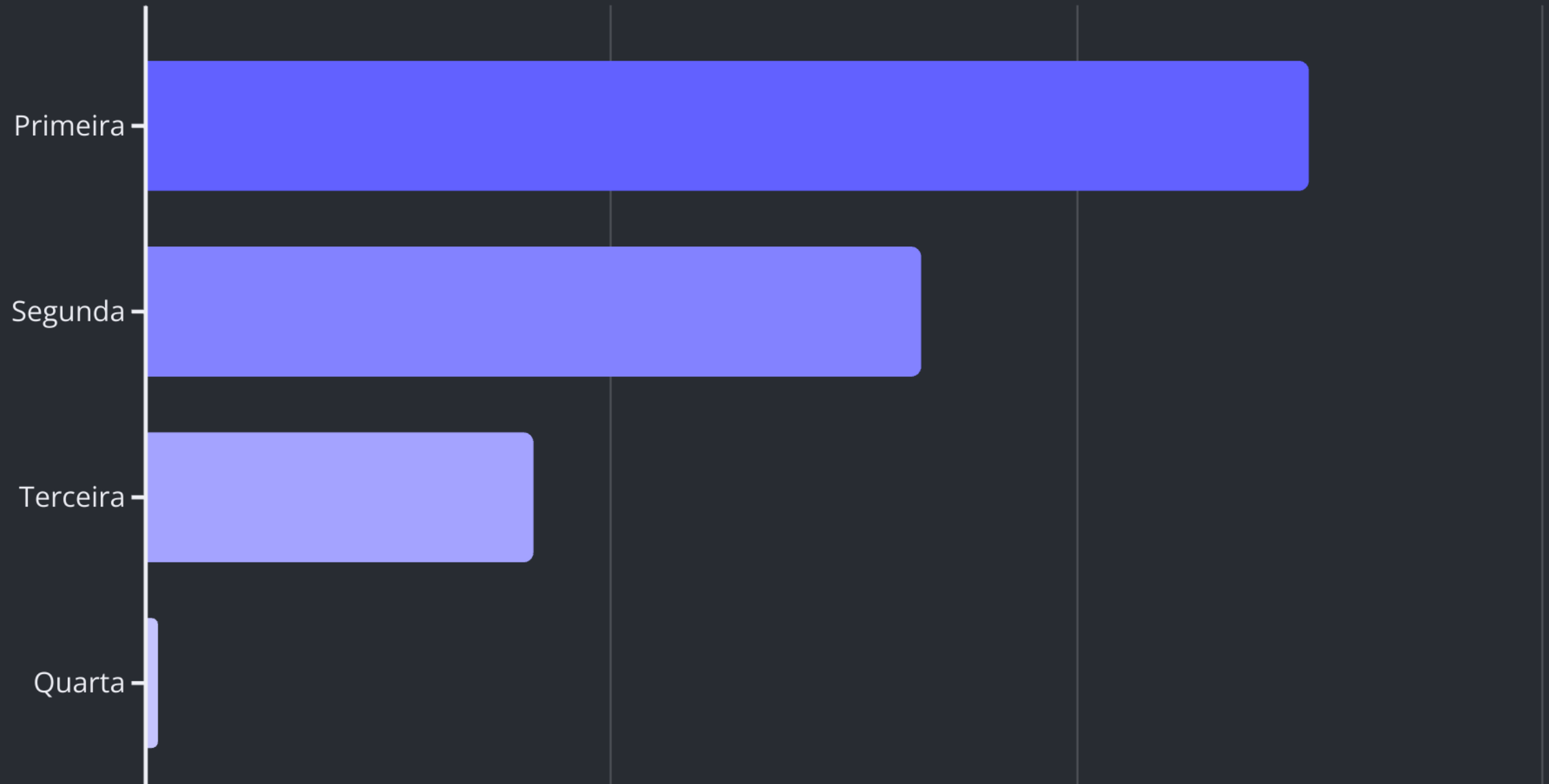
Repete até não haver mais trocas necessárias.

Trocar

Troca posições se estiverem fora de ordem.



Bubble Sort: Exemplo Prático



Bubble Sort: Vantagens e Limitações

100%

Compreensão

Fácil implementação e entendimento completo

O(n²)

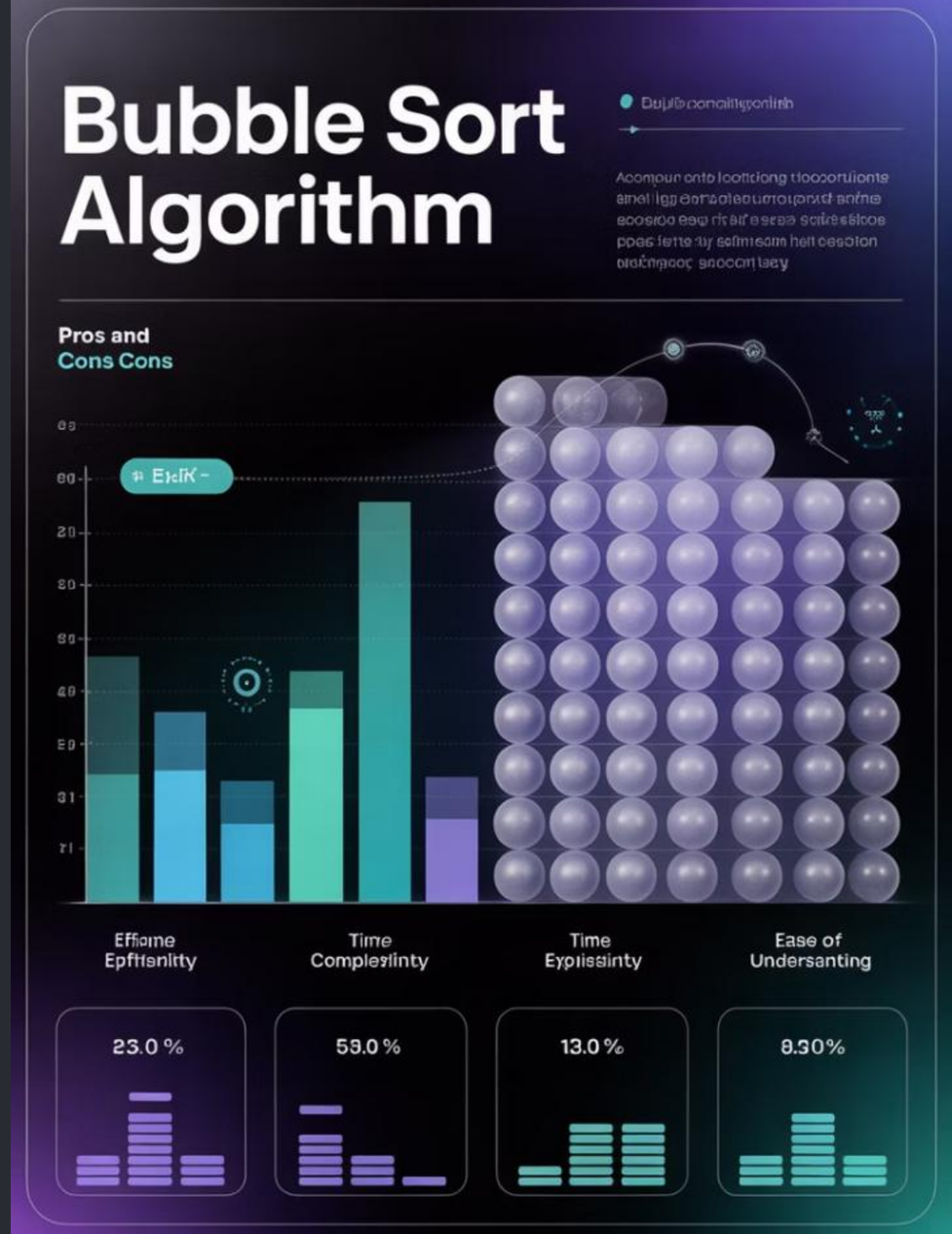
Eficiência

Ineficiente para listas grandes

1

Memória

Requer espaço adicional mínimo



Insertion Sort: Introdução

Analogia das Cartas

Ordena elementos como inserindo cartas numa mão ordenada.

Processo Intuitivo

Método natural que espelha organização humana.

Insertion Sort: Funcionamento

Selecionar

Escolhe próximo elemento não ordenado da sequência.

Encontrar Posição

Localiza posição correta na parte já ordenada.

Inserir

Insere elemento na posição adequada, deslocando outros.

Continuar

Repete processo até todos elementos estarem ordenados.

Insertion Sort: Exemplo Prático

Passo	Array	Elemento Inserido
Inicial	[7, 2, 4, 1]	-
1	[2, 7, 4, 1]	2
2	[2, 4, 7, 1]	4
3	[1, 2, 4, 7]	1

Cada elemento é inserido na posição correta. Parte esquerda sempre permanece ordenada.

Insertion Sort: Vantagens e Limitações

Eficaz para listas pequenas ou quase ordenadas. Performance degrada com tamanho crescente.

Selection Sort: Introdução

Seleção Direta

Seleciona o menor elemento e coloca na posição correta.

Abordagem Sistemática

Método direto de organização por seleção sequencial.

Busca Ativa

Procura ativamente pelo elemento ideal para cada posição.

Selection Sort: Funcionamento Detalhado

Buscar Mínimo

Percorre toda lista procurando menor elemento restante.

Posicionar

Coloca menor elemento na primeira posição disponível.

Avançar

Move para próxima posição e repete processo.

Finalizar

Continua até todos elementos estarem organizados.

Selection Sort: Exemplo Visual

Estado Inicial

Lista: [4, 1, 3, 2]

Busca menor valor em toda lista.

Primeira Seleção

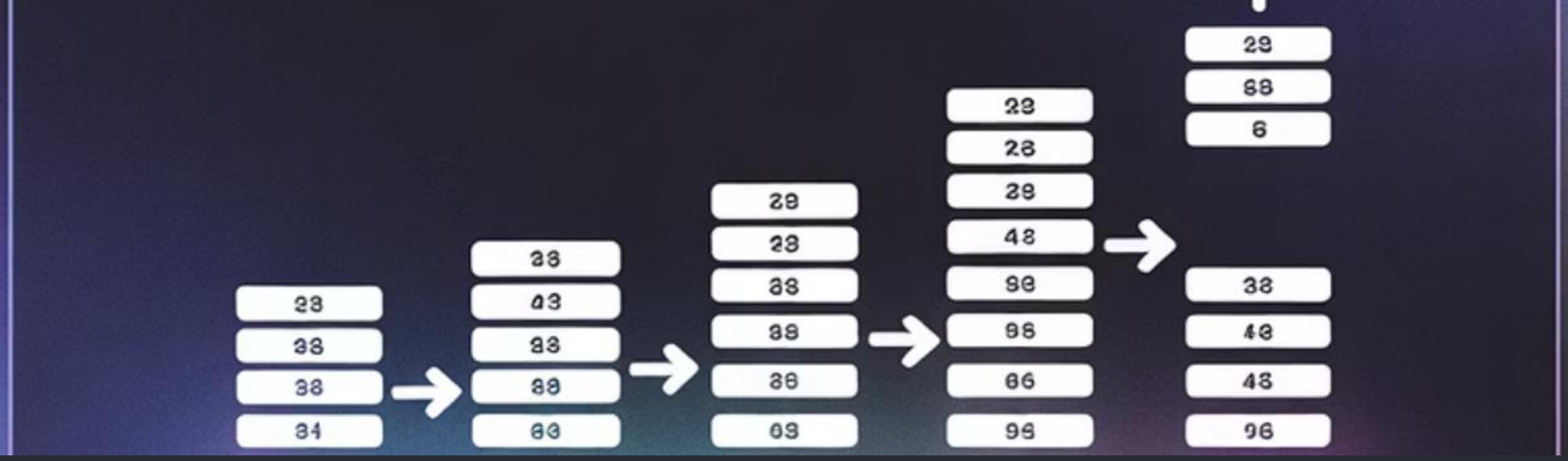
Menor valor: 1

Resultado: [1, 4, 3, 2]

Progresso

Continua selecionando menores valores.

Final: [1, 2, 3, 4]



Selection Sort: Aplicações e Observações

Simplicidade

Implementação direta e compreensível. Ideal para ensino de algoritmos básicos.

Limitações

Pouco eficiente para listas extensas. Performance consistente independente da ordem inicial.

Uso Prático

Adequado para conjuntos pequenos. Base para algoritmos mais sofisticados.

Shell Sort: Ideia Central

Evolução do Insertion

Versão aprimorada do Insertion Sort com saltos estratégicos.

Redução de Movimentos

Usa saltos para reduzir deslocamentos desnecessários de elementos.

Eficiência Melhorada

Melhora significativa na performance comparado ao método base.

Shell Sort: Como Funciona?

Dividir

Divide lista em sublistas usando intervalos específicos.

Ordenar Sublistas

Aplica insertion sort em cada sublista separadamente.

Finalizar

Termina com insertion sort tradicional na lista inteira.

Reduzir Intervalo

Diminui tamanho do intervalo progressivamente.

Shell Sort: Exemplo Prático

1

Gap = 1

Insertion sort final

2

Gap = 2

Sublistas menores

3

Gap = 4

Sublistas iniciais

Lista [9, 7, 3, 1] com gaps decrescentes. Cada etapa aproxima elementos da posição final.

Shell Sort: Onde Usar

Dados Médios

Eficiente para conjuntos de tamanho intermediário a grande.

Alternativa Rápida

Substituto eficiente para métodos simples tradicionais.

Equilíbrio

Combina simplicidade com performance melhorada.

Comb Sort: Ideia Base

Base Bubble

Variante aprimorada do Bubble Sort clássico.

Saltos Maiores

Utiliza saltos maiores nas etapas iniciais.

Melhoria

Performance superior ao algoritmo original.

Comb Sort: Funcionamento e Saltos

1

Gap Inicial

Compara elementos distantes, diminuindo gap gradualmente.

2

Redução

Gap dividido por fator constante a cada iteração.

3

Convergência

Aproxima-se do bubble sort quando gap chega a 1.

Comb Sort: Exemplo Prático

Comb Sort: Características

1.3

Fator Shrink

Divisor padrão para redução do gap

60%

Melhoria

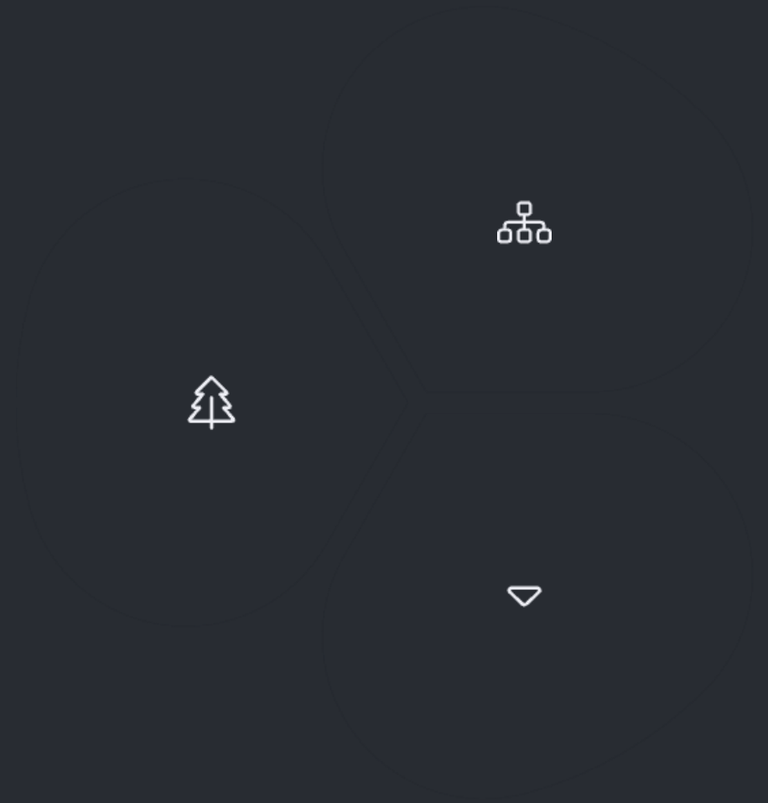
Performance superior ao Bubble Sort

$O(n^2)$

Pior Caso

Complexidade similar aos métodos simples

Heap Sort: Fundamentos



Estrutura Heap

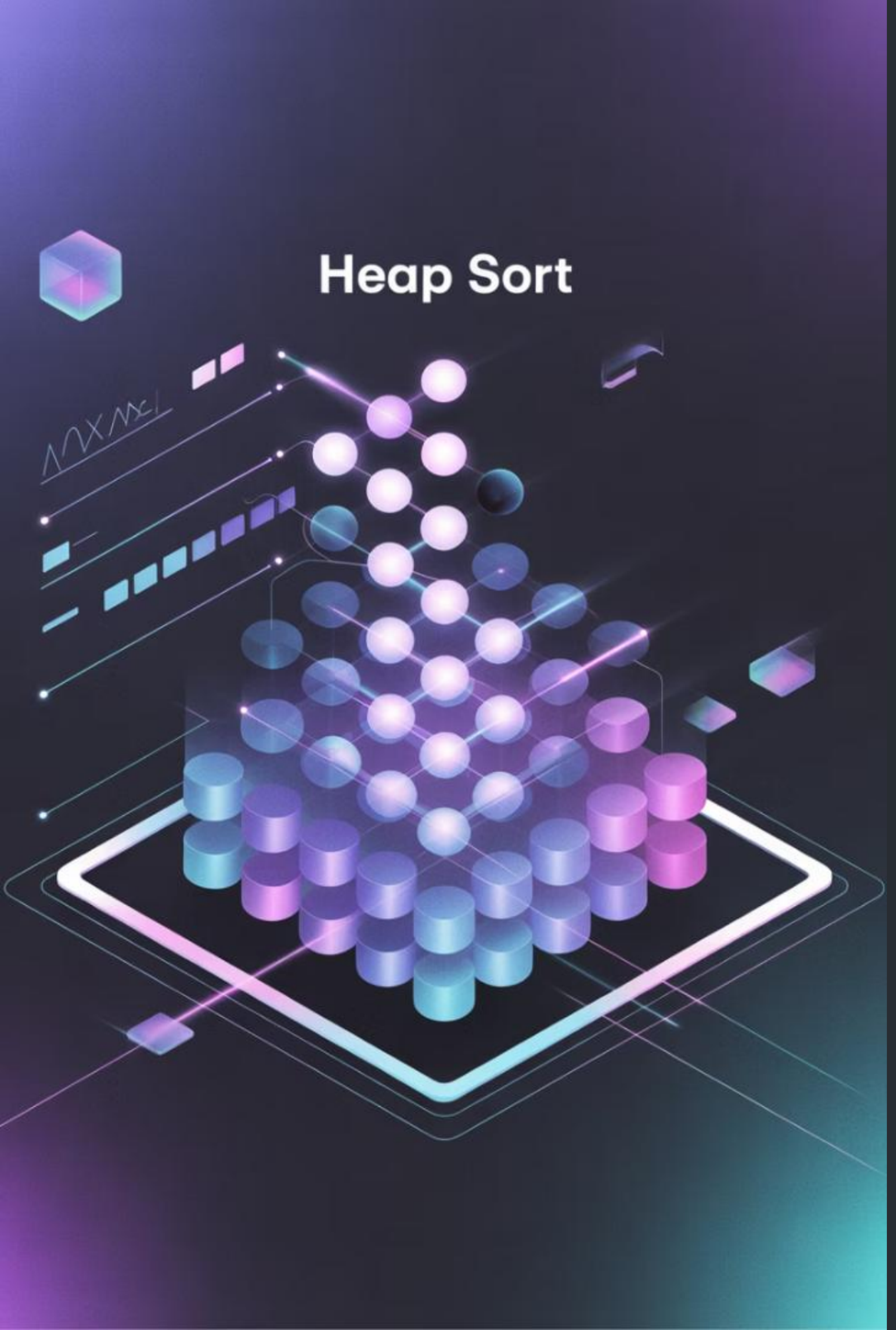
Usa estrutura de heap binário para organização.

Árvore Binária

Organiza elementos como árvore com propriedades específicas.

Ordenação

Extrai elementos em ordem através da estrutura.



Heap Sort: Funcionamento

1. **Construir Heap**

   Transforma array em heap máximo ou mínimo.

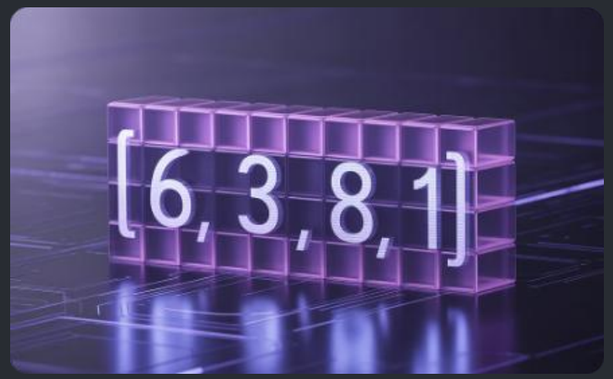
2. **Extrair Raiz**

   Remove maior elemento e reorganiza heap.

3. **Repetir**

   Continua extraindo até heap vazio.

Heap Sort: Exemplo de Execução



Array Inicial

Lista [6, 3, 8, 1] antes da construção do heap.

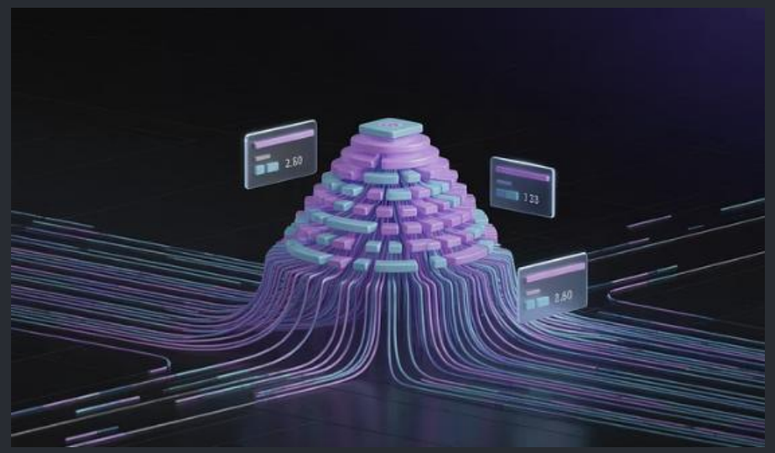
Heap Máximo

Estrutura reorganizada com maior elemento na raiz.

Resultado Final

Array completamente ordenado após extrações sucessivas.

Heap Sort: Aplicação e Eficiência



Utilizado em grandes volumes de dados. Eficiência consistente e uso moderado de memória.

Quick Sort: Introdução

Método do Pivô

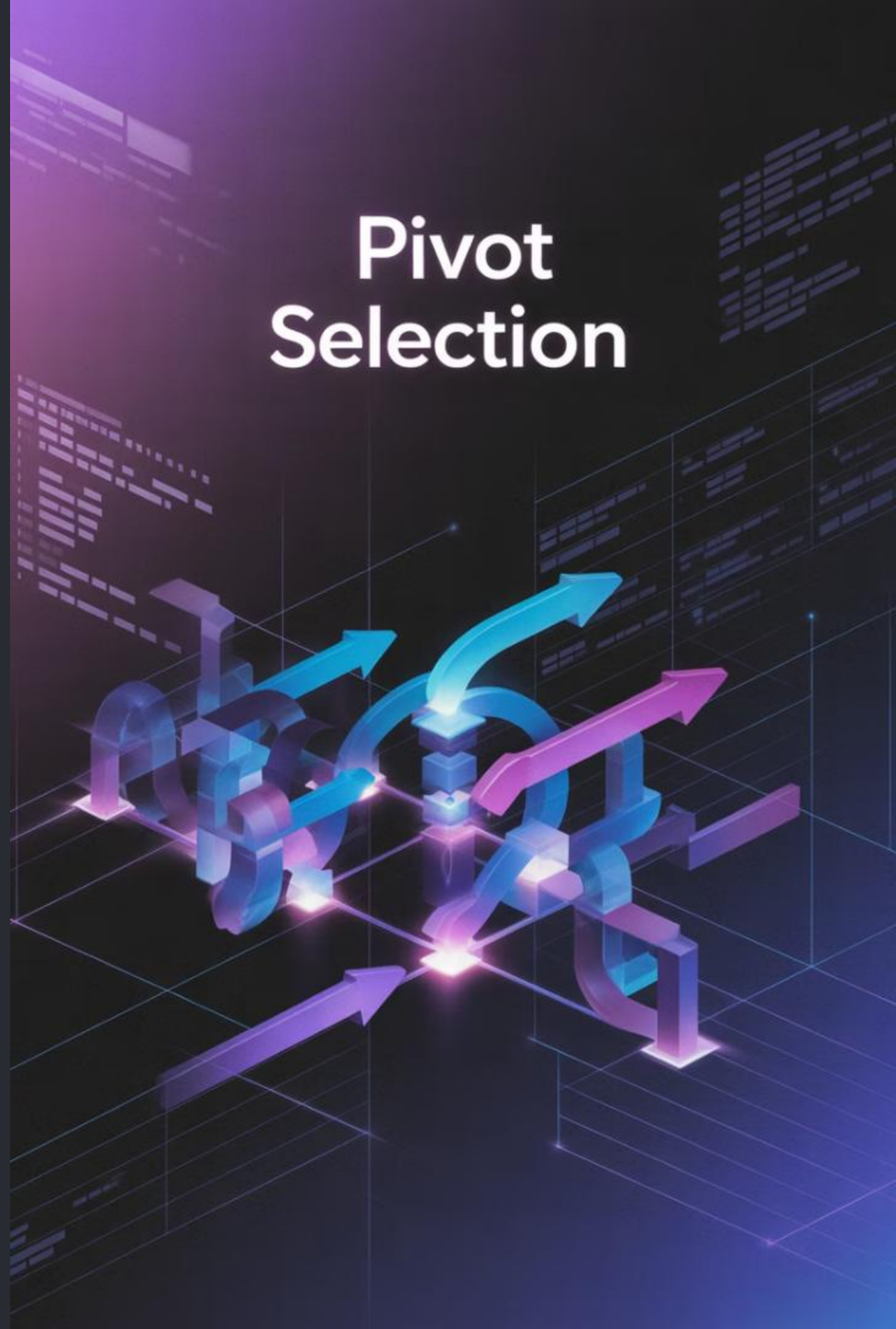
Algoritmo eficiente baseado em seleção de elemento pivô.

Uso Amplo

Amplamente implementado em diversas linguagens de programação.

Performance

Reconhecido pela excelente performance em casos médios.



Quick Sort: Funcionamento Passo a Passo

Escolher Pivô

Seleciona elemento como referência para particionamento.

Particionar

Separa elementos menores e maiores que o pivô.

Recursão

Aplica recursivamente o processo em subconjuntos.

Combinar

Junta resultados das chamadas recursivas.

Quick Sort: Exemplo Didático

Etapa	Array	Pivô	Ação
Inicial	[10, 7, 8, 9]	9	Particionar
Partição	[7, 8] [9] [10]	-	Recursão
Final	[7, 8, 9, 10]	-	Ordenado

Pivô 9 divide array em menores [7,8] e maiores [10]. Recursão ordena subconjuntos.

Quick Sort: Vantagens

Velocidade

Muito rápido em listas grandes com boa distribuição de dados.

Eficiência Espacial

Requer pouco espaço adicional comparado a outros métodos avançados.

Adaptabilidade

Performance pode ser otimizada através de diferentes estratégias de pivô.

Merge Sort: Visão Geral

Divisão

Baseado na técnica "divide e conquista" para resolver problemas.

Conquista

Divide listas recursivamente até obter elementos únicos.

Combinação

Mescla listas ordenadas progressivamente até resultado final.

Merge Sort: Processo de Divisão

1

Unidades

Elementos individuais

2

Pares

Grupos de 2 elementos

3

Quadruplos

Grupos de 4 elementos

4

Array Original

Lista completa

Fragmenta até listas unitárias, depois junta ordenadamente. Processo sistemático e previsível.

Merge Sort: Exemplo Prático

Divisão

[4,6,2,5] → [4,6] [2,5]

[4,6] → [4] [6]

[2,5] → [2] [5]

Fusão

[4] [6] → [4,6]

[2] [5] → [2,5]

[4,6] [2,5] → [2,4,5,6]

Merge Sort: Quando Usar

Estruturas Ligadas

Preferido em listas ligadas devido ao acesso sequencial.

Grandes Volumes

Excelente para processamento de grandes quantidades de dados.

Estabilidade

Mantém ordem relativa de elementos iguais.

Counting Sort: Conceito Diferente

1

Sem Comparação

Não usa comparação direta entre elementos para ordenar.

2

Contagem

Conta ocorrências de cada elemento no conjunto.

3

Reconstrução

Reconstrói array ordenado baseado nas contagens.

Counting Sort: Funcionamento

Contar

Conta aparições de cada valor único no array.

Acumular

Calcula posições finais baseado nas contagens.

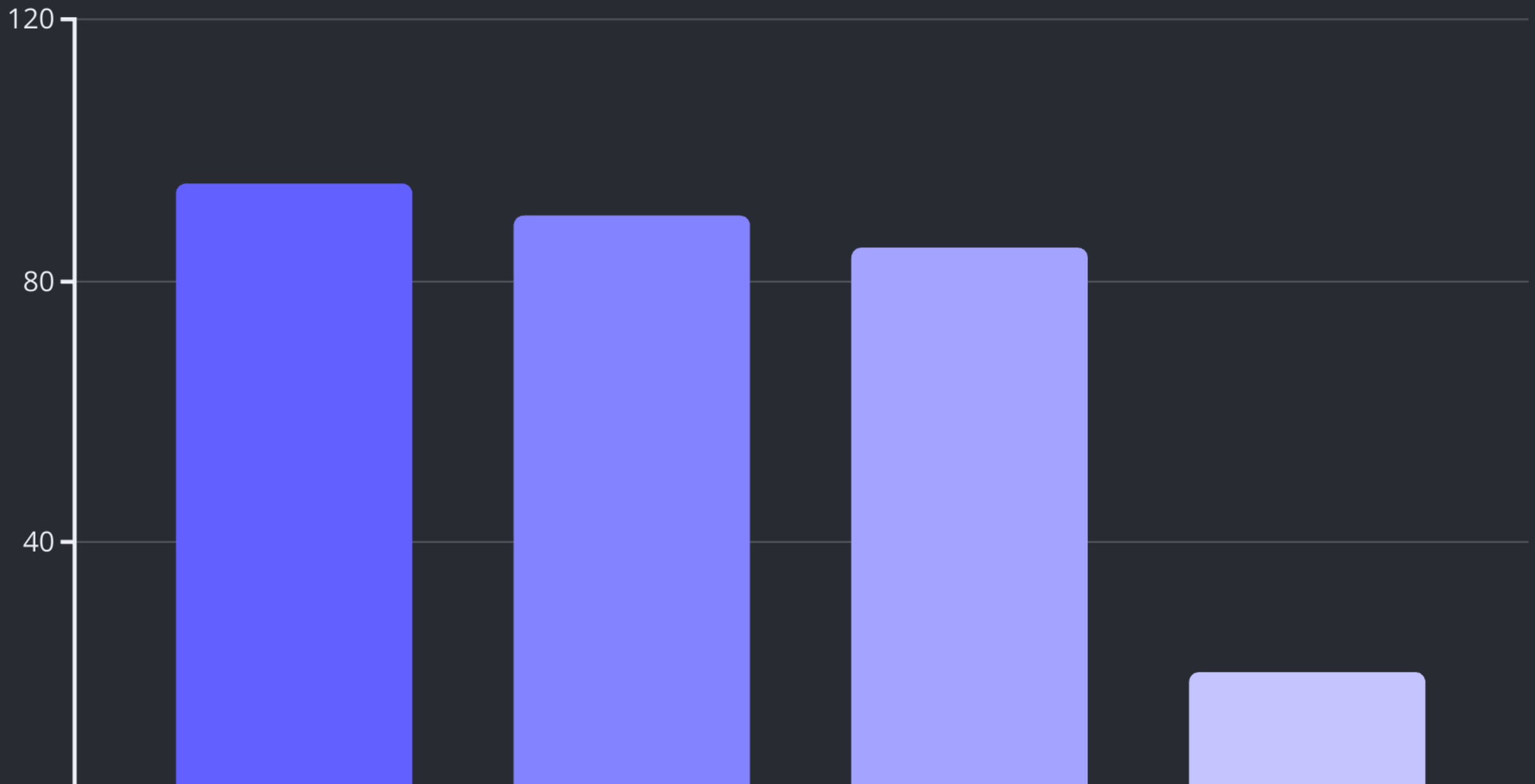
Copiar

Transfere resultado para array original.

Construir

Monta lista ordenada usando informações calculadas.

Counting Sort: Quando Aplicar



Bucket Sort: Estratégia Geral

Distribuir

Separa elementos em baldes baseado em critérios.

Ordenar Baldes

Ordena individualmente cada balde usando algoritmo apropriado.

Concatenar

Junta baldes ordenados para formar resultado final.

Bucket Sort: Funcionamento em Etapas

1

Distribuição

Elementos espalhados em baldes segundo função hash.

2

Ordenação Individual

Cada balde ordenado separadamente com algoritmo eficiente.

3

Reunião Final

Baldes concatenados na ordem para resultado ordenado.

Bucket Sort: Utilidade

O(n)

Melhor Caso

Performance linear com distribuição ideal

10

Baldes Típicos

Número comum de baldes utilizados

0.1

Fator Carga

Elementos por balde para eficiência ótima

Radix Sort: Ordenação por Dígitos

Unidades

Primeira passagem ordena por dígito das unidades.

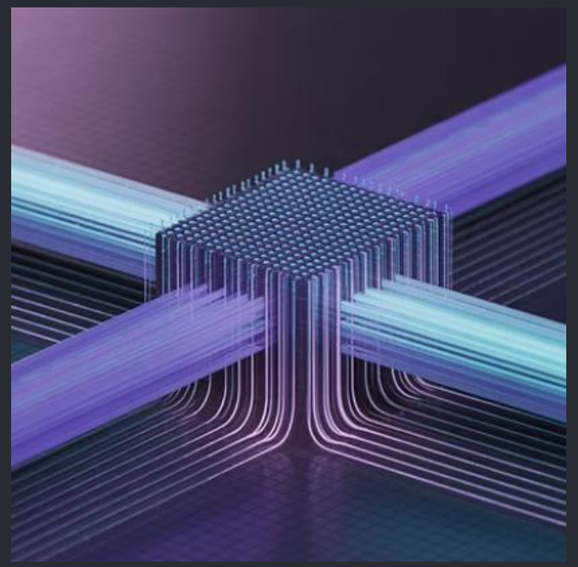
Dezenas

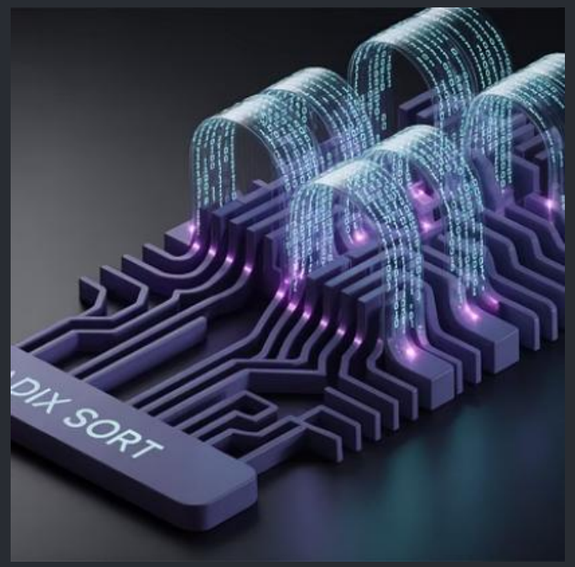
Segunda passagem considera dígito das dezenas.

Centenas

Continua até processar todos os dígitos significativos.

Radix Sort: Passos do Algoritmo





Ordena por dígitos menos significativos para mais importantes. Usa algoritmo estável internamente.

Radix Sort: Exemplo Prático

Dígito	Após Ordenação
Unidades	[170, 45, 75, 90]
Dezenas	[170, 45, 75, 90]
Centenas	[45, 75, 90, 170]

Lista [170, 45, 75, 90] processada dígito por dígito. Resultado final completamente ordenado.

Comparação e Escolha dos Algoritmos

Tamanho

Volume dos dados influencia escolha do método.

Tipo

Estrutura e natureza dos dados determinam algoritmo.

Recursos

Limitações de memória e processamento são cruciais.

Distribuição

Padrão de distribuição afeta performance significativamente.

Pesquisa Linear: Introdução

Busca Sequencial

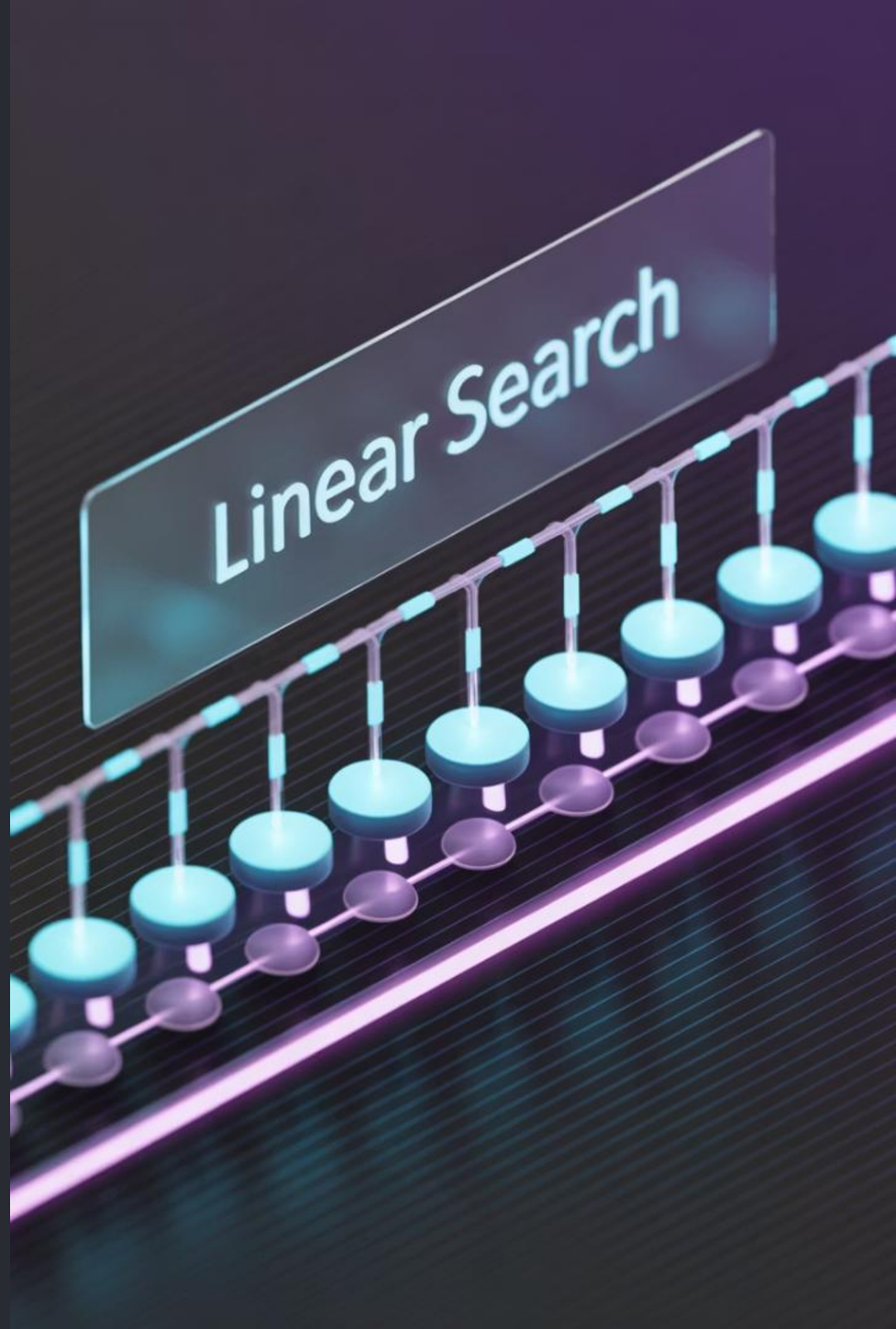
Examina cada elemento da lista sequencialmente até encontrar.

Simplicidade

Método mais direto e simples de implementar.

Universalidade

Funciona em qualquer tipo de lista ou estrutura.



Pesquisa Linear: Exemplo

1. **Posição 0**

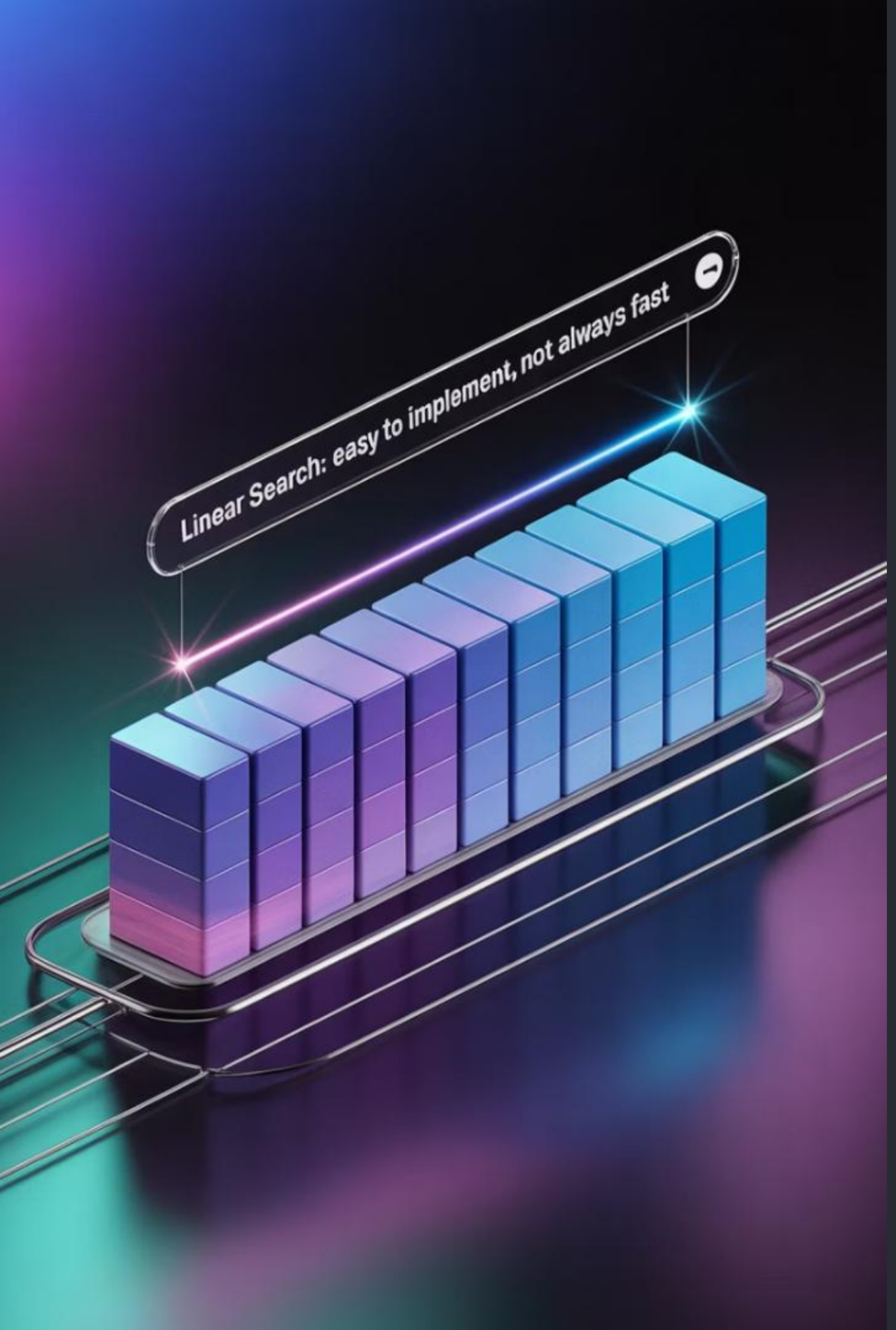
   Verifica 7 ≠ 5, continua busca.

2. **Posição 1**

   Verifica 10 ≠ 5, continua busca.

3. **Posição 2**

   Verifica 5 = 5, elemento encontrado!



Pesquisa Linear: Características

Implementação

Extremamente fácil de implementar e compreender. Código direto e intuitivo.

Aplicação Ideal

Usada para listas pequenas ou não ordenadas. Não requer pré-processamento.

Performance

Eficiência limitada em grandes conjuntos. Examina todos elementos no pior caso.

Pesquisa Binária: Introdução

Pré-requisito

Aplica-se exclusivamente em listas previamente ordenadas.

Estratégia

Divide lista ao meio sucessivamente eliminando metades.

Eficiência

Reduz drasticamente número de comparações necessárias.

Pesquisa Binária: Funcionamento

Meio

Calcula elemento central da faixa atual.

Comparar

Compara elemento buscado com o central.

Repetir

Repete processo na metade restante.

Eliminar

Descarta metade que não contém elemento.

Pesquisa Binária: Exemplo

Início

Lista [2,4,6,8,10], busca pelo 6. Meio = 6.

Comparação

Elemento central 6 = elemento buscado 6.

Sucesso

Elemento encontrado na primeira tentativa!

Pesquisa Binária: Vantagens

O(log n)

Complexidade

Crescimento logarítmico com tamanho da lista

1M

Capacidade

Busca em milhão de elementos com ~20 comparações

50%

Redução

Elimina metade dos candidatos a cada step

Muito rápida para listas grandes ordenadas. Revoluciona eficiência de busca em grandes volumes.