INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Campos do Jordão

ESTRUTURA DE DADOS

Algoritmos de Ordenação por Distribuição e Intercalação + Mais Algoritmos.

Merge Sort, Radix Sort, Quick Sort, Heap Sort, Counting Sort.

Professor Mestre Igor de Moraes Sampaio
igor.sampaio@ifsp.edu.br

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo - **Campus Campos do Jordão**

Algoritmos de Ordenação por Intercalação

Definição Geral

- Os **algoritmos de ordenação por intercalação** (ou merge sort algorithms) seguem a ideia de **dividir o problema em partes menores, ordenar cada uma e depois intercalar (mesclar) os resultados.**

Eles se baseiam no princípio Dividir para Conquistar (Divide and Conquer).

Um Novo Paradigma de Algoritmos

O que é Divisão e Conquista?

- Uma estratégia algorítmica que resolve problemas complexos dividindo-os em subproblemas menores e mais simples, resolvendo cada subproblema independentemente e depois combinando as soluções para obter a solução do problema original.

1 Dividir

Dividir o problema original em subproblemas menores, geralmente de tamanho igual ou similar.

Exemplo: Dividir um array em duas metades.

2 Conquistar

Resolver os subproblemas recursivamente. Se os subproblemas forem pequenos o suficiente, resolvê-los diretamente.

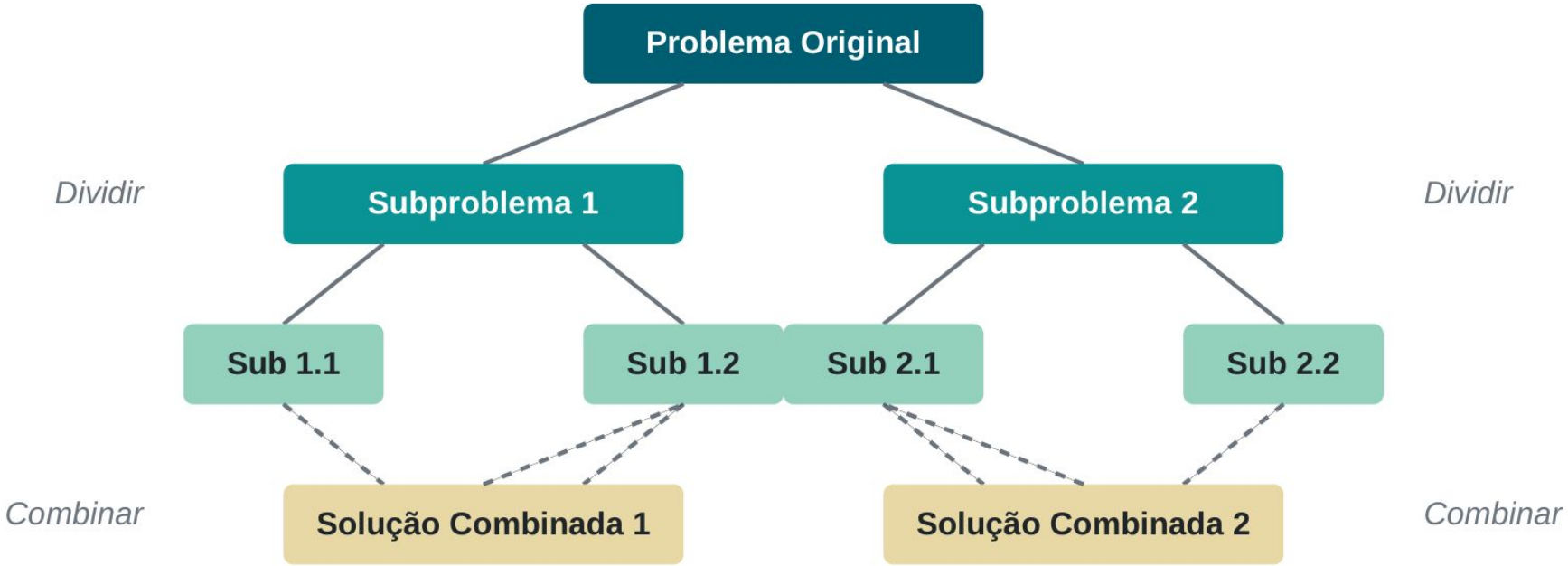
Exemplo: Ordenar cada metade do array.

3 Combinar

Combinar as soluções dos subproblemas para obter a solução do problema original.

Exemplo: Mesclar as duas metades ordenadas.

Visualização do Paradigma



Merge Sort

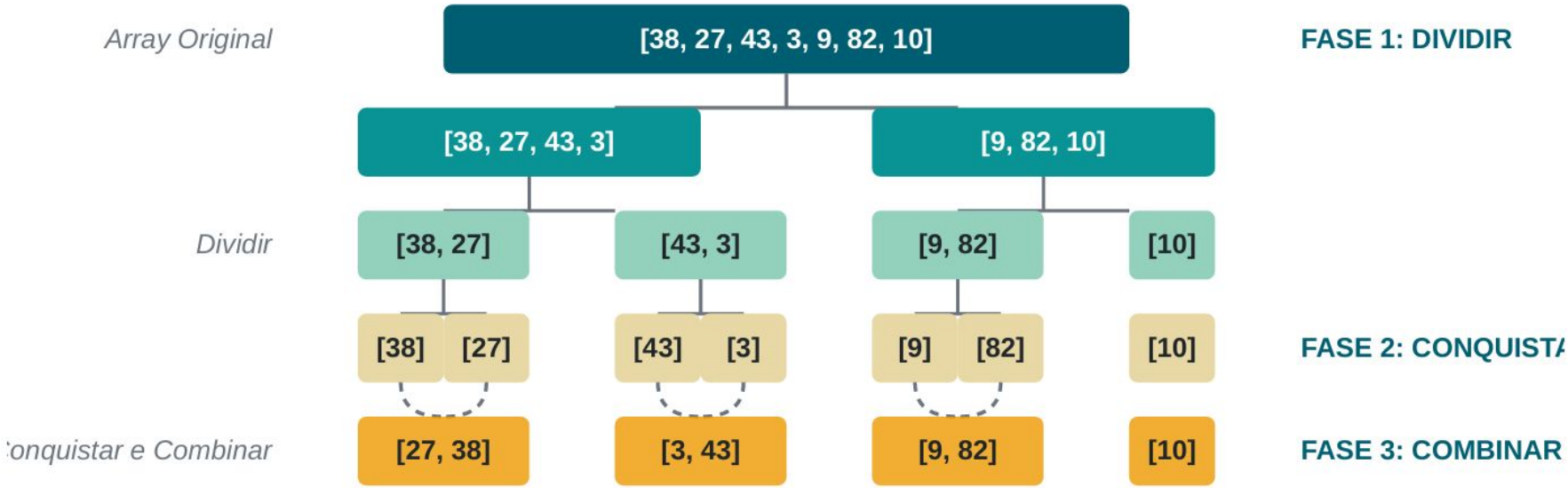
Merge Sort

- O Merge Sort é um algoritmo de ordenação eficiente baseado no paradigma de Divisão e Conquista:

- Ideia Fundamental

    - Dividir o array em metades menores, ordenar cada metade recursivamente e, em seguida, mesclar (merge) as metades ordenadas para produzir o array final ordenado.

Visão Geral do Processo



Como Funciona? (Passo a Passo)

Exemplo: Ordenando [38, 27, 43, 3, 9, 82, 10]

Estado Inicial:

Nível 1: Array completo

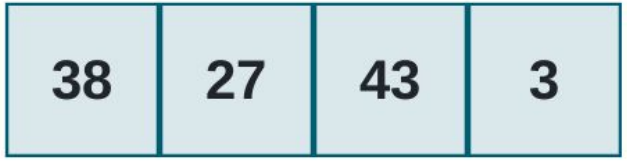
38	27	43	3	9	82	10

Como Funciona? (Passo a Passo)

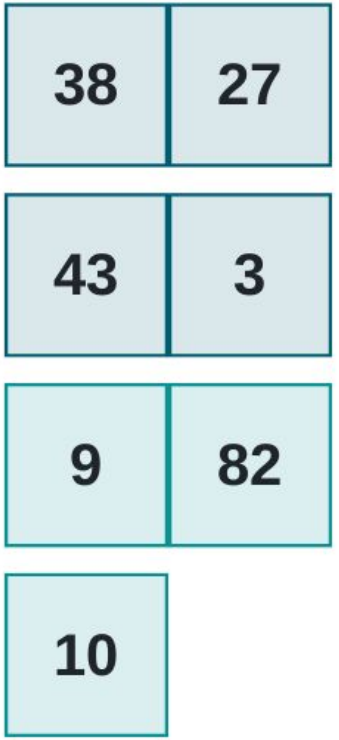
Fase 1: Divisão Recursiva

Nível 1: Array completo

Nível 2: Dividido em duas metades



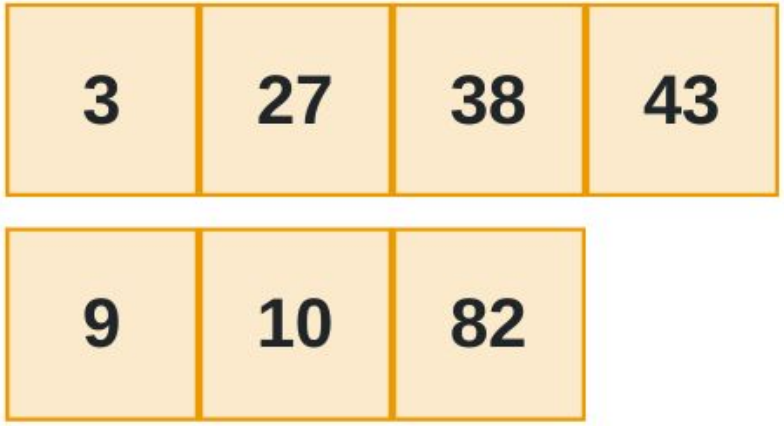
Nível 3: Subdivisões menores



Como Funciona? (Passo a Passo)

Como Funciona? (Passo a Passo)

Nível 2: Mesclagem de quartetos



Nível 1: Mesclagem final

Como Funciona a Mescla? (Passo a Passo)

Vamos mesclar duas sublistas ordenadas: [3, 27, 38, 43] e [9, 10, 82]

Passo 1: Inicialização

Comparamos os primeiros elementos: 3 < 9, então colocamos 3 no array resultado.

Como Funciona a Mescla? (Passo a Passo)

Passo 2: Próxima comparação

Comparamos os próximos elementos: 27 > 9, então colocamos 9 no array resultado.

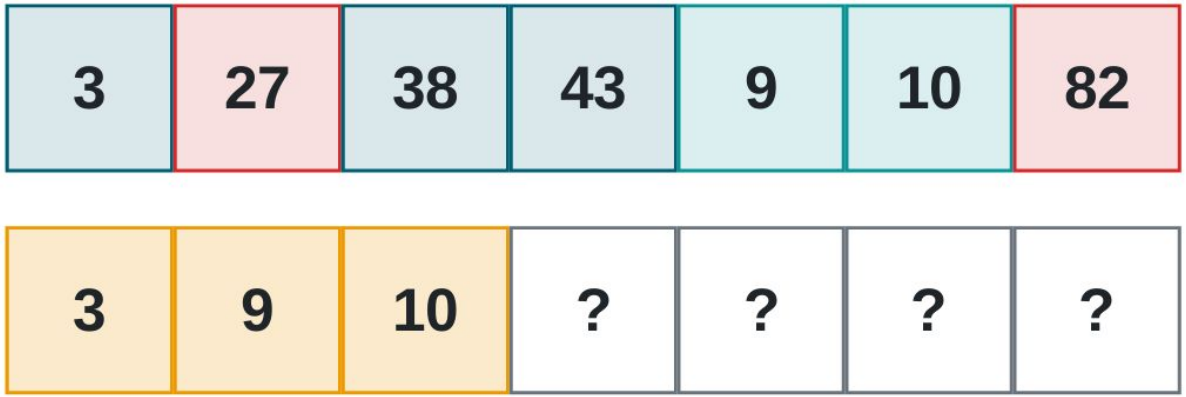
Como Funciona a Mescla? (Passo a Passo)

Passo 3: Continuando

Comparamos os próximos elementos: 27 > 10, então colocamos 10 no array resultado.

Como Funciona a Mescla? (Passo a Passo)

Passo 4: Continuando



Comparamos os próximos elementos: 27 < 82, então colocamos 27 no array resultado.

Como Funciona a Mescla? (Passo a Passo)

Resultado Final

3	9	10	27	38	43	82

Continuamos o processo até que todos os elementos sejam colocados no array resultado.

Análise

Análise de Complexidade

Complexidade de Tempo

Análise do Algoritmo:

- **Divisão:** Dividir o array ao meio leva tempo constante O(1)
- **Recursão:** Duas chamadas recursivas para metades do tamanho n/2
- **Mesclagem:** Mesclar n elementos leva tempo O(n)

Complexidade:

- **Melhor caso:** O(n log n)
- **Caso médio:** O(n log n)
- **Pior caso:** O(n log n)

Equação de Recorrência:

$$T(n) = 2T(n/2) + O(n)$$

Pelo Teorema Mestre, isso resolve para O(n log n)

Análise de Complexidade

Complexidade de Espaço:

- **O(n)** - Requer espaço auxiliar proporcional ao tamanho do array
- Arrays temporários são criados durante a mesclagem
- Pilha de recursão também consome espaço O(log n)

Características

- **Estabilidade:**
  - **Estável** - Mantém a ordem relativa de elementos com valores iguais.
  - Durante a mesclagem, elementos iguais do primeiro subarray são colocados antes dos elementos do segundo subarray.
- **Adaptabilidade:**
  - **Não adaptativo** - O desempenho não melhora para arrays parcialmente ordenados.
  - Sempre executa o mesmo número de operações, independentemente da ordem inicial dos elementos.

Características

- **Espaço Extra:** A mesclagem requer espaço auxiliar proporcional ao tamanho dos arrays sendo mesclados (O(n)).

- **Previsibilidade:** O processo de mesclagem sempre realiza no máximo (n-1) comparações para mesclar n elementos.

- **Eficiência:** A mesclagem é um processo linear O(n), mas o Merge Sort completo é O(n log n) devido às divisões recursivas.

- **Dica:** O processo de mesclagem é o que torna o Merge Sort eficiente, pois combina duas listas ordenadas em tempo linear.

Vantagens e Desvantagens

- Vantagens
  - Complexidade garantida O(n log n) em todos os casos
  - Estável - mantém a ordem relativa de elementos iguais
  - Eficiente para grandes conjuntos de dados
  - Bom para ordenação externa (dados que não cabem na memória)
  - Paralelizável - diferentes partes podem ser processadas simultaneamente
- Desvantagens
  - Requer espaço auxiliar O(n) - não é in-place
  - Overhead para arrays pequenos
  - Não adaptativo - não aproveita ordenação parcial
  - Mais complexo de implementar que algoritmos simples
  - Alocação e desalocação de memória podem ser custosas

Quando Usar?

- Melhor escolha quando:
  - O conjunto de dados é grande (> 1000 elementos)
  - Você precisa de desempenho garantido O (n log n)
  - A estabilidade é importante
  - Você está trabalhando com ordenação externa (dados que não cabem na memória)
  - Você pode paralelizar o processamento

Desafio

Desafio

- Implemente o Merge Sort.

```
// Função principal do Merge Sort
void mergeSort(int arr[], int inicio, int fim) {
    if (inicio < fim) {
        // Encontra o ponto médio
        int meio = inicio + (fim - inicio) / 2;
        // Ordena a primeira e a segunda metade
        mergeSort(arr, inicio, meio);
        mergeSort(arr, meio + 1, fim);
        // Mescla as metades ordenadas
        merge(arr, inicio, meio, fim);
    }
}
```

Sugestão de Exercícios

Sugestão de Exercícios

- **Bottom-up Merge Sort:** Implementação iterativa (não recursiva) do Merge Sort que começa mesclando pares de elementos, depois grupos de 4, 8, etc. Vantagens: Elimina a sobrecarga da recursão e o uso da pilha.

- **Natural Merge Sort:** Aproveita sequências já ordenadas no array original, identificando "corridas" naturais antes de mesclar. Vantagens: Adaptativo, mais eficiente para arrays parcialmente ordenados.

- **Merge Sort Híbrido:** Usa Insertion Sort para pequenos subarrays (tipicamente < 10-20 elementos) e Merge Sort para o resto. Vantagens: Reduz o overhead para pequenos arrays onde Insertion Sort é mais eficiente.

- **In-place Merge Sort:** Implementação que tenta minimizar o uso de memória auxiliar, realizando a mesclagem no próprio array. Vantagens: Reduz o uso de memória, mas geralmente com maior complexidade de tempo.

Sugestão de Exercícios

- **Evitar Cópias Desnecessárias:** Alternar entre dois arrays auxiliares em vez de copiar de volta para o array original a cada mesclagem.
- **Otimização de Cache:** Organizar o acesso à memória para maximizar o uso do cache, processando blocos de tamanho adequado.
- **Paralelização:** Dividir o trabalho entre múltiplos threads ou processos, especialmente para grandes conjuntos de dados.
- **Verificação de Arrays Já Ordenados:** Verificar se os subarrays já estão ordenados antes de mesclar, evitando operações desnecessárias.

Algoritmos de Ordenação por Distribuição

Definição Geral

- A ideia central é **organizar os elementos conforme sua representação (posição, dígito, faixa de valor, etc.)**, e não por comparações sucessivas.

- Esses algoritmos são geralmente muito eficientes quando o **intervalo de valores possível é conhecido e limitado,** ou quando os dados possuem estrutura que pode ser explorada (como números com dígitos).

Características Gerais

- Não baseada em comparações entre elementos
- Pode atingir complexidade linear O(n) em casos específicos
- Geralmente requer conhecimento sobre os dados (faixa de valores, etc.)
- Frequentemente usa mais memória auxiliar
- Exemplos: Counting Sort, Radix Sort, Bucket Sort

Vantagens e Desvantagens

Vantagens:

- Pode superar o limite teórico de O(n log n)
- Eficiente para conjuntos de dados específicos
- Não depende de comparações entre elementos
- Alguns algoritmos são estáveis por natureza

Desvantagens:

- Geralmente requer conhecimento prévio sobre os dados
- Pode consumir mais memória auxiliar
- Nem sempre aplicável a todos os tipos de dados
- Eficiência depende da distribuição dos dados

Counting Sort

Counting Sort

- O Counting Sort, um algoritmo de ordenação por distribuição que pode atingir complexidade linear O(n+k) quando aplicado a conjuntos de dados com características específicas.

- O Counting Sort é um algoritmo de ordenação baseado em contagem e distribuição de elementos:

- Ideia Fundamental

    - Contar a frequência de cada elemento no array original, usar essas contagens para determinar a posição correta de cada elemento no array ordenado, e então construir o array ordenado.

Pré-requisitos para Aplicação

- **Valores inteiros:** Os elementos devem ser inteiros (ou mapeáveis para inteiros)
- **Faixa conhecida:** Deve-se conhecer o intervalo de valores possíveis (min e max)
- **Faixa razoável:** O intervalo de valores não deve ser muito grande em relação ao tamanho do array

Exemplo de cenário ideal:

- Ordenar idades de pessoas (0-120)
- Ordenar notas de alunos (0-100)
- Ordenar códigos postais de uma região
- Ordenar frequências de palavras em um texto

Como Funciona

1. **Contagem**
   - Contar a frequência de cada valor no array original e armazenar em um array auxiliar.
2. **Acumulação**
   - Modificar o array de contagem para que cada elemento contenha a soma dos elementos anteriores.
3. **Posicionamento**
   - Colocar cada elemento do array original na posição correta no array de saída, usando o array de contagem acumulada.

Como Funciona? (Passo a Passo)

Exemplo: Ordenando [4, 2, 2, 8, 3, 3, 1]

Estado Inicial:

Como Funciona? (Passo a Passo)

Passo 1: Contagem de Frequências

Array Original

4	2	2	8	3	3	1

Array de Contagem (índices 0 a 8)

0	1	2	2	1	0	0	0	1

012345678

Como Funciona? (Passo a Passo)

Passo 2: Acumulação de Contagens

Array de Contagem Acumulada

0	1	3	5	6	6	6	6	7

012345678

Cada valor indica a posição final do último elemento com aquele valor.

Como Funciona? (Passo a Passo)

Passo 3: Posicionamento dos Elementos

Processando o array original de trás para frente:

4	2	2	8	3	3	1

Elemento: 1, Posição: count[1] = 1, Após decrementar: count[1] = 0

Array de saída após processar todos os elementos:

1	2	2	3	3	4	8

Desafio

Desafio

- Vamos implementar o Counting Sort por partes juntos.

Desafio

- Passo 1: Contagem
  - Contamos a frequência de cada elemento no array original e armazenamos no array de contagem.
  - O array de contagem possui o tamanho do intervalo de valores.

Desafio

- Passo 2: Acumulação
  - Modificamos o array de contagem para que cada posição contenha a soma das contagens anteriores, representando a posição final de cada elemento.

Desafio

- Passo 3: Posicionamento
  - Construímos o array ordenado colocando cada elemento na posição correta, usando o array de contagem como referência.

Desafio

- Observações Importantes
  - Percorremos o array original de trás para frente para garantir a estabilidade.
  - Decrementamos o contador após cada posicionamento para lidar com elementos repetidos.

Análise

Análise de Complexidade

Complexidade de Tempo

Encontrar o valor máximo

Percorrer o array uma vez: **O(n)**

Contar ocorrências

Percorrer o array uma vez: **O(n)**

Acumular contagens

Percorrer o array de contagem: **O(k)**, onde k é o tamanho do intervalo de valores

Construir array de saída

Percorrer o array uma vez: **O(n)**

Copiar para o array original

Percorrer o array uma vez: **O(n)**

Complexidade Total de Tempo

O(n + k)

Onde n é o tamanho do array e k é o tamanho do intervalo de valores (max - min + 1).

Análise de Complexidade

Análise de Casos

Melhor Caso

Complexidade: O(n + k)

O Counting Sort tem a mesma complexidade em todos os casos, independentemente da distribuição inicial dos dados.

Caso Médio

Complexidade: O(n + k)

A complexidade permanece constante, pois o algoritmo não depende de comparações entre elementos.

Pior Caso

Complexidade: O(n + k)

Mesmo no pior caso, a complexidade não muda, o que é uma vantagem em relação aos algoritmos baseados em comparação.

Análise de Complexidade

Complexidade de Espaço

Array de contagem

Tamanho proporcional ao intervalo de valores: **O(k)**

Array de saída

Tamanho proporcional ao array original: **O(n)**

Complexidade Total de Espaço

O(n + k)

O algoritmo requer espaço adicional proporcional ao tamanho do array e ao intervalo de valores.

Observação Importante

Se k >> n (o intervalo de valores é muito maior que o tamanho do array), o Counting Sort pode se tornar ineficiente tanto em tempo quanto em espaço.

Cuidados na Implementação

- **Alocação de memória:** Verificar se há memória suficiente para o array de contagem.
- **Intervalo muito grande:** Se o intervalo for muito maior que o tamanho do array, o algoritmo se torna ineficiente.
- **Índices negativos:** O algoritmo básico não lida com valores negativos diretamente.
- **Vazamento de memória:** Sempre liberar a memória alocada dinamicamente.

Características

- **Não comparativo:** Não compara elementos entre si
- **Estabilidade:** Mantém a ordem relativa de elementos iguais
- **Complexidade:** O(n+k), onde k é o tamanho do intervalo de valores
- **Memória:** Requer espaço auxiliar O(k) para o array de contagem
- **Eficiência:** Muito eficiente quando k é pequeno em relação a n

O Counting Sort é um exemplo de como, ao conhecermos características específicas dos dados, podemos desenvolver algoritmos que superam o limite teórico de O(n log n) dos algoritmos baseados em comparação.

Características

Estabilidade

O Counting Sort é um algoritmo **estável**, o que significa que ele preserva a ordem relativa de elementos iguais.

Isso é garantido pelo processamento do array original de trás para frente durante a construção do array de saída.

In-Place vs. Not-In-Place

O Counting Sort **não é in-place**, pois requer espaço adicional proporcional ao tamanho do array e ao intervalo de valores.

Isso pode ser uma limitação em cenários com restrições de memória ou com grandes volumes de dados.

Vantagens e Desvantagens

Vantagens e Desvantagens

Vantagens

- Complexidade linear
- Estável por natureza
- Simples de implementar
- Eficiente para dados específicos

Desvantagens

- Requer conhecimento prévio dos dados
- Ineficiente para grandes intervalos
- Uso de memória extra
- Limitado a inteiros ou dados mapeáveis

Sugestão de Exercícios

Sugestão de Exercícios

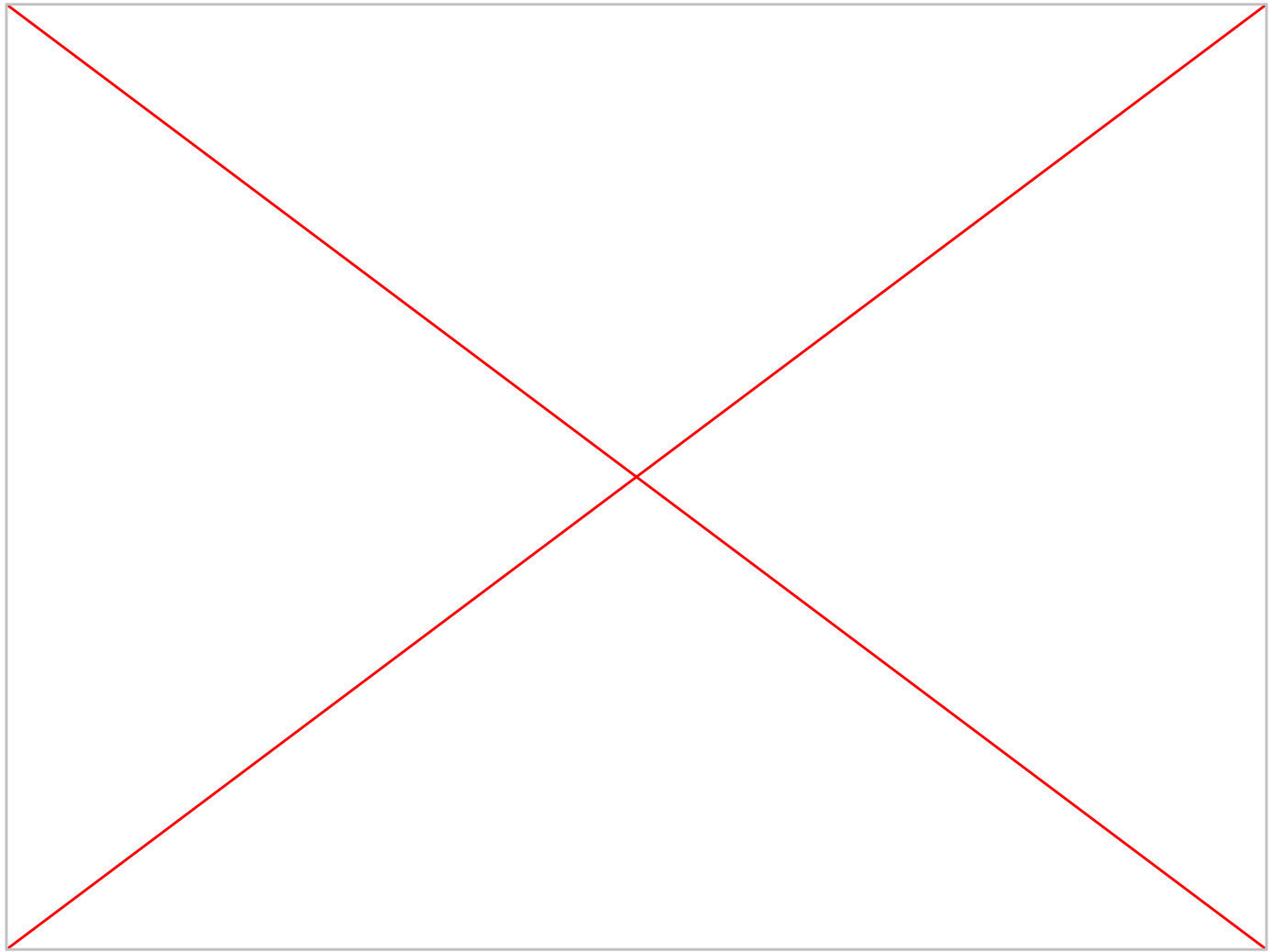
- **Intervalo desconhecido:** Se não soubermos o intervalo exato de valores, podemos buscar pelo máximo.

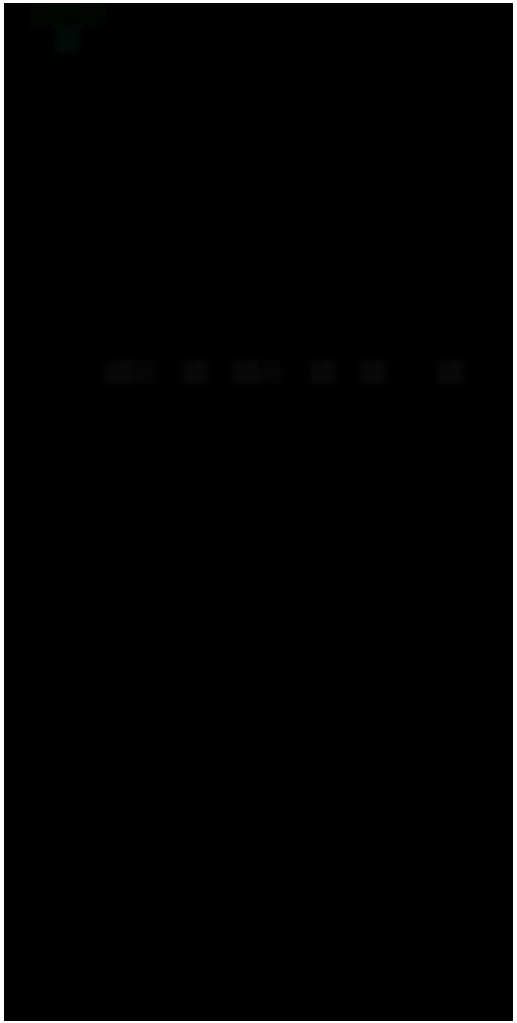
- **Valores negativos:** Podemos adaptar o algoritmo para lidar com valores negativos, deslocando o intervalo .

- **Valores não inteiros:** Para outros tipos de dados, podemos mapear os valores para índices inteiros (por exemplo: para números decimais podemos multiplicando o valor por uma constante e ao final dividir pela mesma).

- **Memória:** Para intervalos muito grandes, podemos usar técnicas de compressão ou outras estruturas de dados.

Outros Algoritmos






INGOSTR

Comparação dos Algoritmos

Comparação de Características

Algoritmo	Complexidade Tempo (Médio)	Complexidade Espaço	Estável	In-place	Adaptativo	Baseado em Comparação
Bubble Sort	$O(n^2)$	**O(1)**	**Sim**	**Sim**	**Sim**	Sim
Selection Sort	$O(n^2)$	**O(1)**	Não	**Sim**	Não	Sim
Insertion Sort	$O(n^2)$	**O(1)**	**Sim**	**Sim**	**Sim**	Sim
Merge Sort	O(n log n)	O(n)	**Sim**	Não	Não	Sim
Counting Sort	**O(n + k)**	O(n + k)	**Sim**	Não	Não	**Não**
Radix Sort	**O(d * n)**	O(n + k)	**Sim**	Não	Não	**Não**
Quick Sort	O(n log n)	**O(log n)**	Não	**Sim**	Não	Sim
Heap Sort	O(n log n)	**O(1)**	Não	**Sim**	Não	Sim

Nota: Nos algoritmos, d é o número de dígitos e k é o intervalo de valores.
Células destacadas em verde indicam vantagens relativas, enquanto células em vermelho claro indicam possíveis desvantagens.