Algoritmos de Ordenação: InsertionSort e QuickSort em C

Bem-vindos! Nesta aula, vamos explorar os algoritmos de ordenação InsertionSort e QuickSort, entendendo como eles funcionam, comparando seus desempenhos e aplicando-os na linguagem C.

PU por Prof. Me. William P. S. Júnior

Objetivos da aula e resultados esperados

Objetivos

- Compreender os princípios básicos do InsertionSort e QuickSort.
- Implementar ambos os algoritmos em C.
- Comparar o desempenho dos algoritmos em diferentes cenários.

Resultados Esperados

- Capacidade de escolher o algoritmo de ordenação ideal para um problema específico.
- Domínio da implementação dos algoritmos em C.
- Compreensão das vantagens e desvantagens de cada algoritmo.

Conceitos fundamentais de ordenação de dados

1 Ordenação

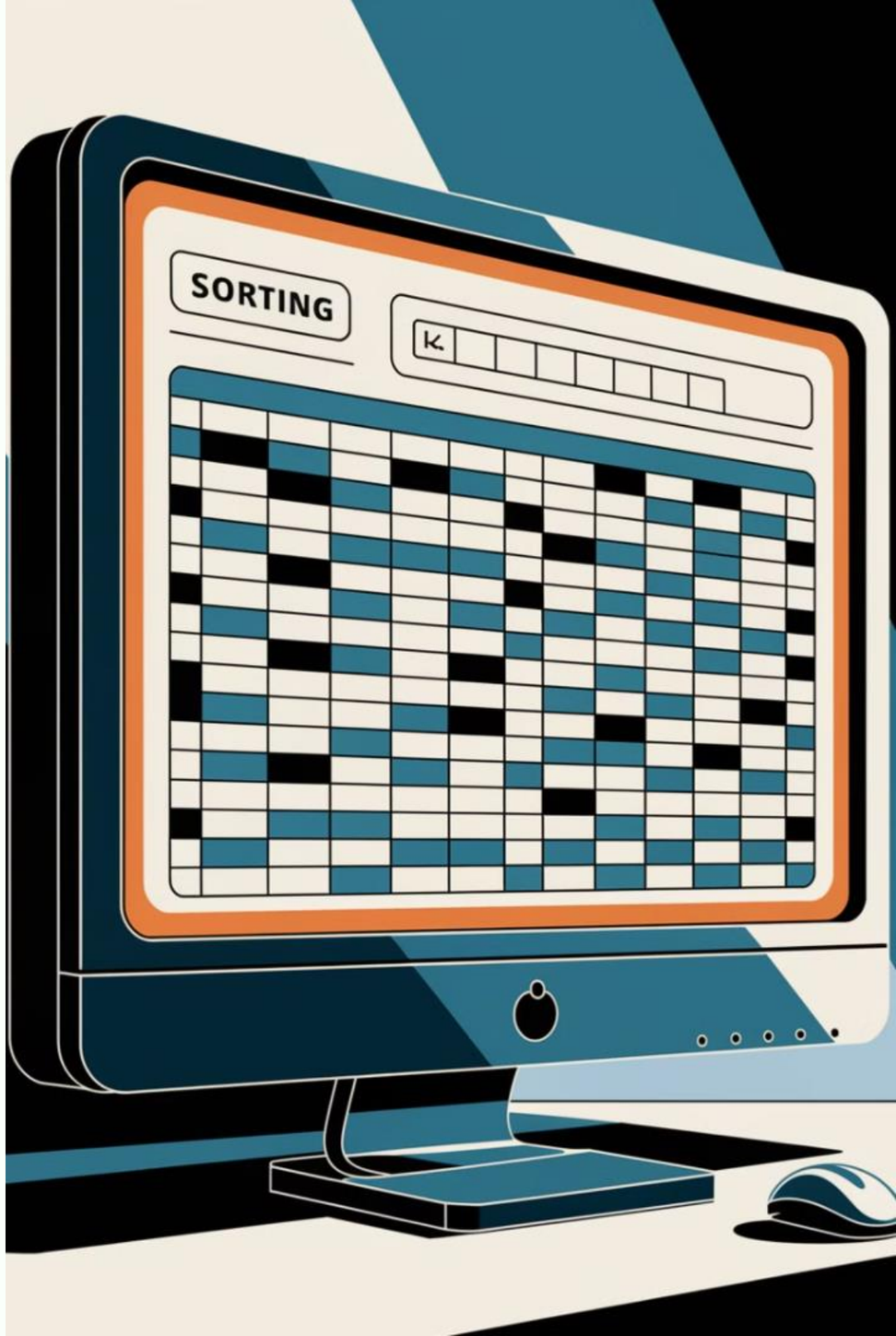
O processo de organizar dados em uma sequência específica, como ordem crescente ou decrescente.

2 Algoritmos de Ordenação

Procedimentos passo a passo para ordenar dados, com diferentes eficiências e complexidades.

3 Complexidade

Medida de quantas operações são necessárias para ordenar um conjunto de dados.



Analogia prática: organizando cartas de baralho

Início

Comece com um monte de cartas em ordem aleatória.

Primeiro passo

Pegue a segunda carta e compare-a com a primeira, inserindo-a na posição correta.

Próximos passos

Pegue cada carta restante e insira-a na posição correta dentro do conjunto de cartas já ordenadas.

Visão geral do InsertionSort: princípios básicos

Princípio

Insere cada elemento em sua posição correta em uma sublista já ordenada.

Funcionamento

Começa com o segundo elemento, comparando-o com o anterior e movendo-o para a posição correta, repetindo o processo para os elementos restantes.

Demonstração do código InsertionSort passo a passo

Laço externo

Percorre o array, inserindo cada elemento em sua posição correta.

Laço interno

Compara o elemento atual com os elementos anteriores, movendo-os para a direita para abrir espaço.

Inserção

Insere o elemento atual na posição correta na sublista ordenada.

Visão geral do InsertionSort: princípios básicos

Funcionamento

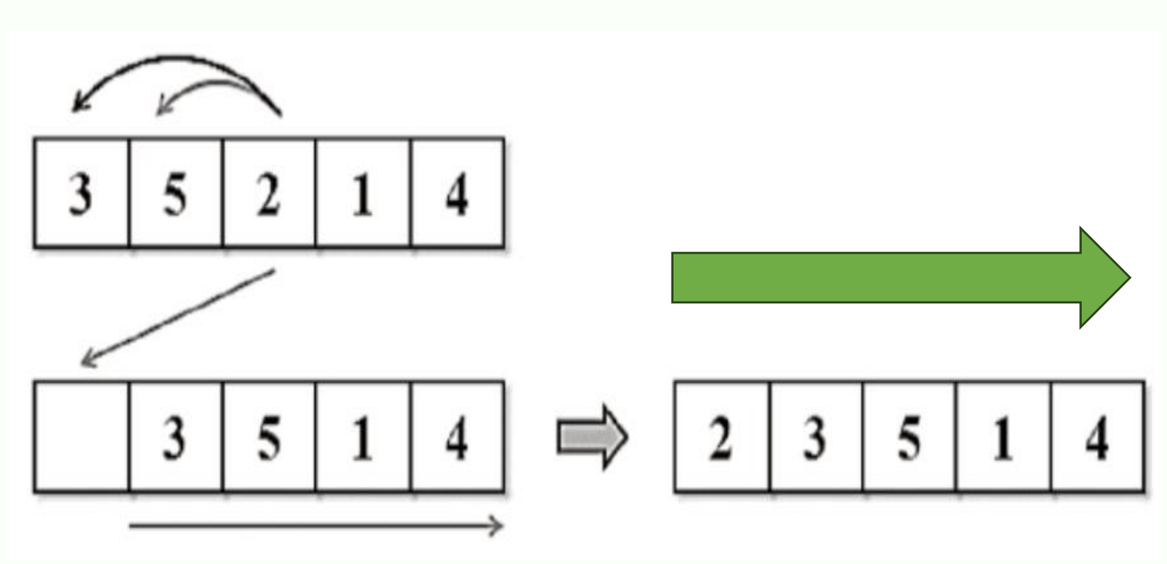
Começa com o segundo elemento, comparando-o com o anterior e movendo-o para a posição correta, repetindo o processo para os elementos restantes.

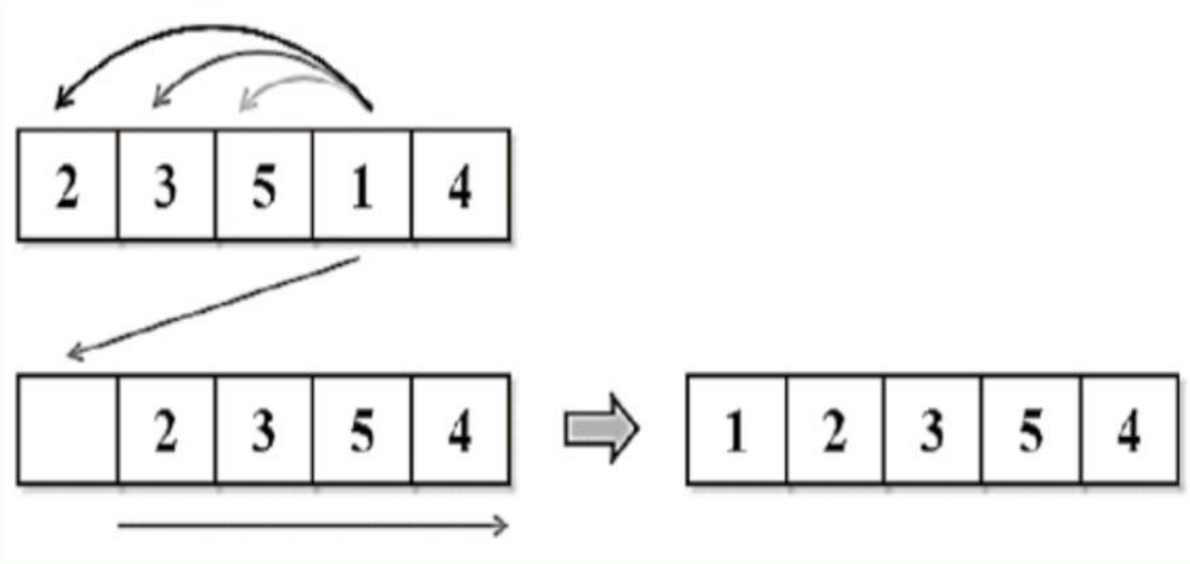

```
Numero => 5 (posição 1)
vetor [1] x vetor[0] (5 < 3 ? não faz nada)
```

Visão geral do InsertionSort: princípios básicos

Funcionamento

Começa com o segundo elemento, comparando-o com o anterior e movendo-o para a posição correta, repetindo o processo para os elementos restantes.

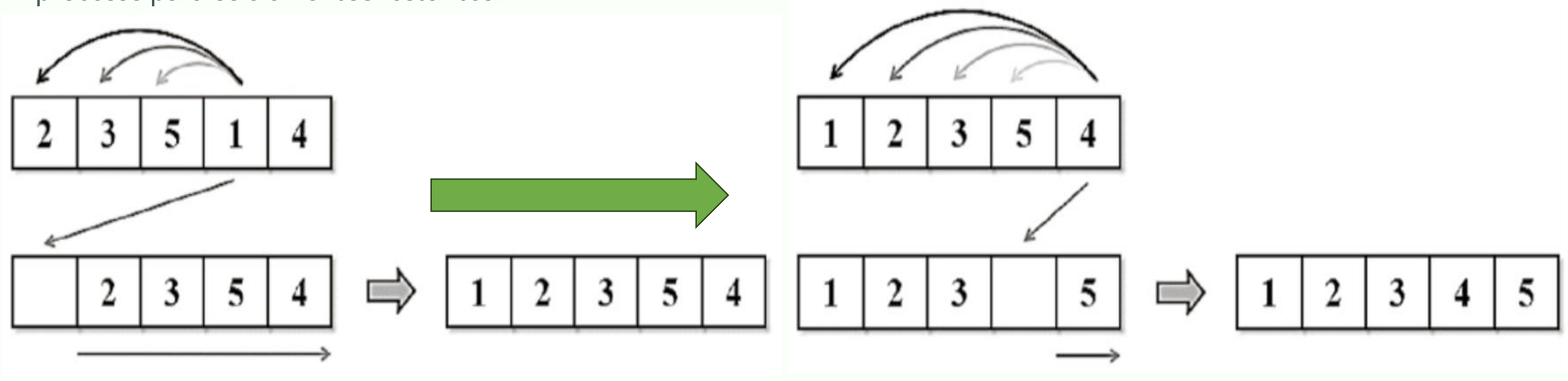




Visão geral do InsertionSort: princípios básicos

Funcionamento

Começa com o segundo elemento, comparando-o com o anterior e movendo-o para a posição correta, repetindo o processo para os elementos restantes.



Implementação do InsertionSort em C: estrutura básica

```
void insertionSort(int arr[], int n) {
  int i, key, j;
  for (i = 1; i < n; i++) {
    key = arr[i];
    j = i - 1;
    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j = j - 1;
    }
    arr[j + 1] = key;
  }
}
```

Análise de complexidade do InsertionSort

Melhor caso

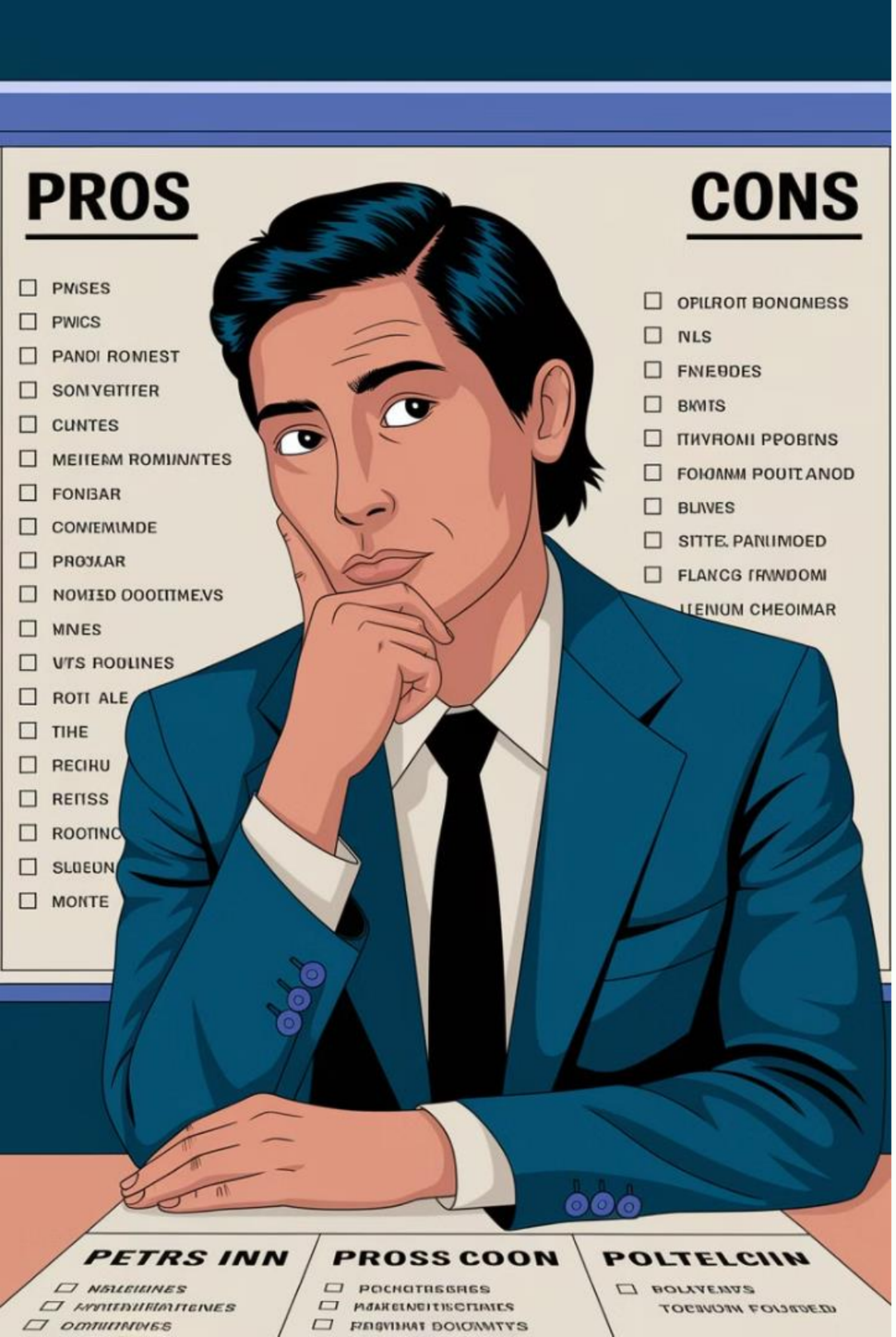
O(n): quando o array já está ordenado.

Pior caso

O(n^2): quando o array está em ordem inversa.

Caso médio

O(n^2): similar ao pior caso, mas com ligeira melhora.



Quando usar InsertionSort: vantagens e desvantagens

Vantagens

- Eficiente para conjuntos de dados pequenos.
- Estável: mantém a ordem relativa dos elementos iguais.
- Em-place: não requer espaço extra.

Desvantagens

- Ineficiente para conjuntos de dados grandes.
- Complexidade quadrática no pior caso.

Introdução ao QuickSort: conceito de particionamento

Particionamento

Dividir o array em duas sublistas: elementos menores que o pivô e maiores que o pivô.

Recursividade

Repetir o particionamento nas sublistas até que o array seja totalmente ordenado.

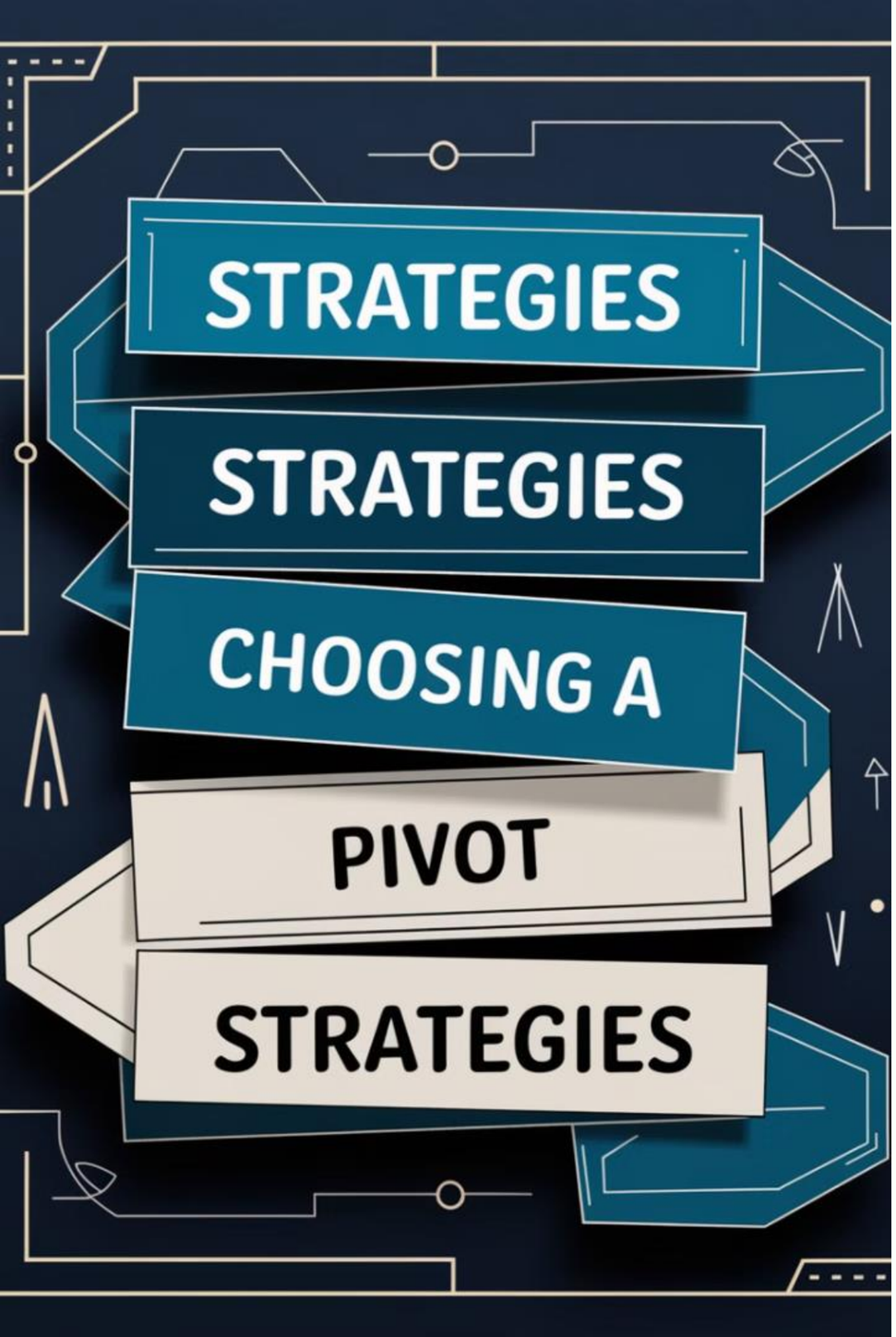
O papel do pivô no QuickSort

Ponto de referência

O pivô é um elemento usado para dividir o array em sublistas.

Determinante da ordem

Os elementos menores que o pivô ficam à esquerda, e os maiores à direita.



Estratégias de escolha do pivô

1 ***Primeiro elemento***

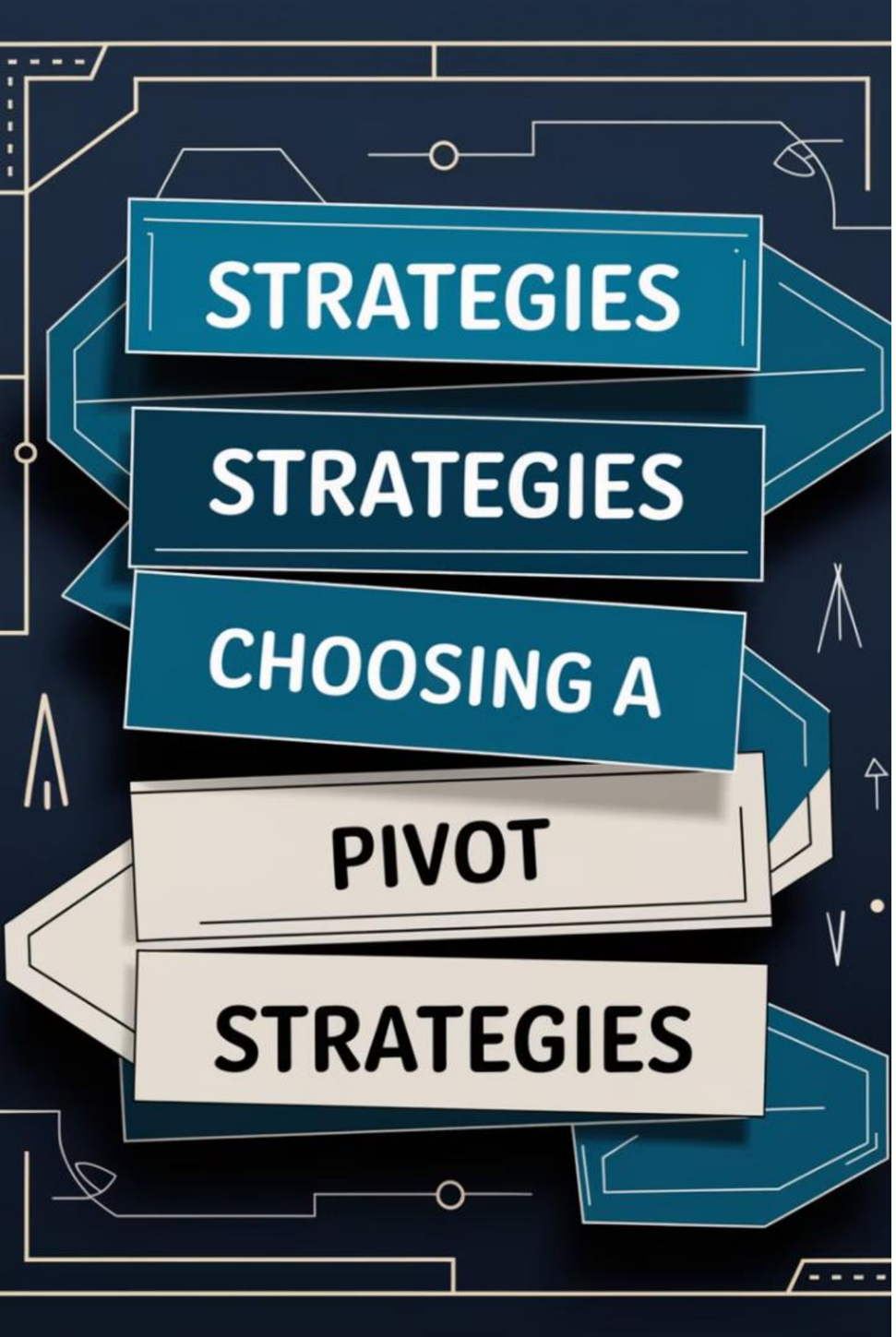
Escolher o primeiro elemento do array como pivô.

2 ***Último elemento***

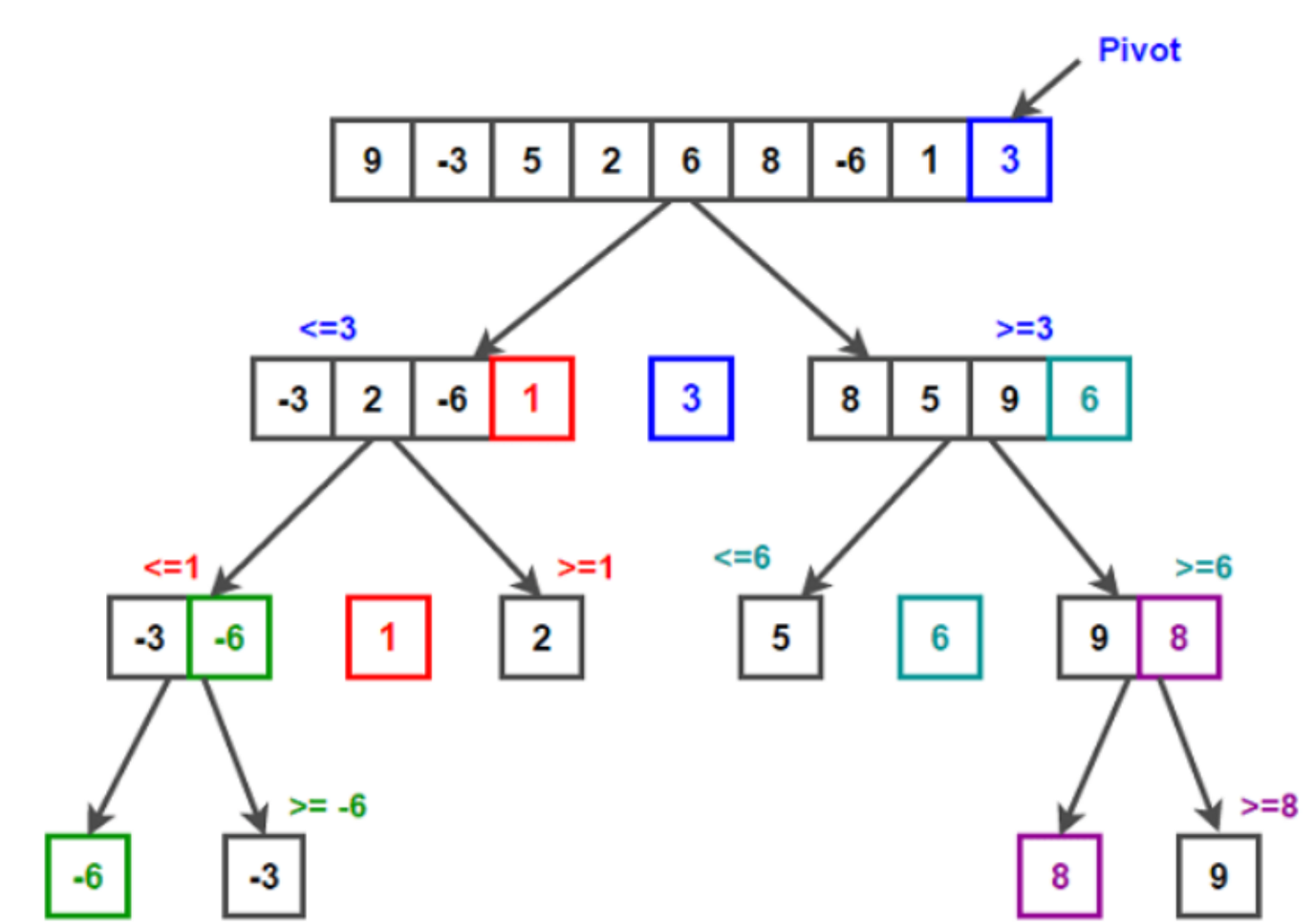
Escolher o último elemento do array como pivô.

3 ***Elemento aleatório***

Escolher um elemento aleatório do array como pivô.

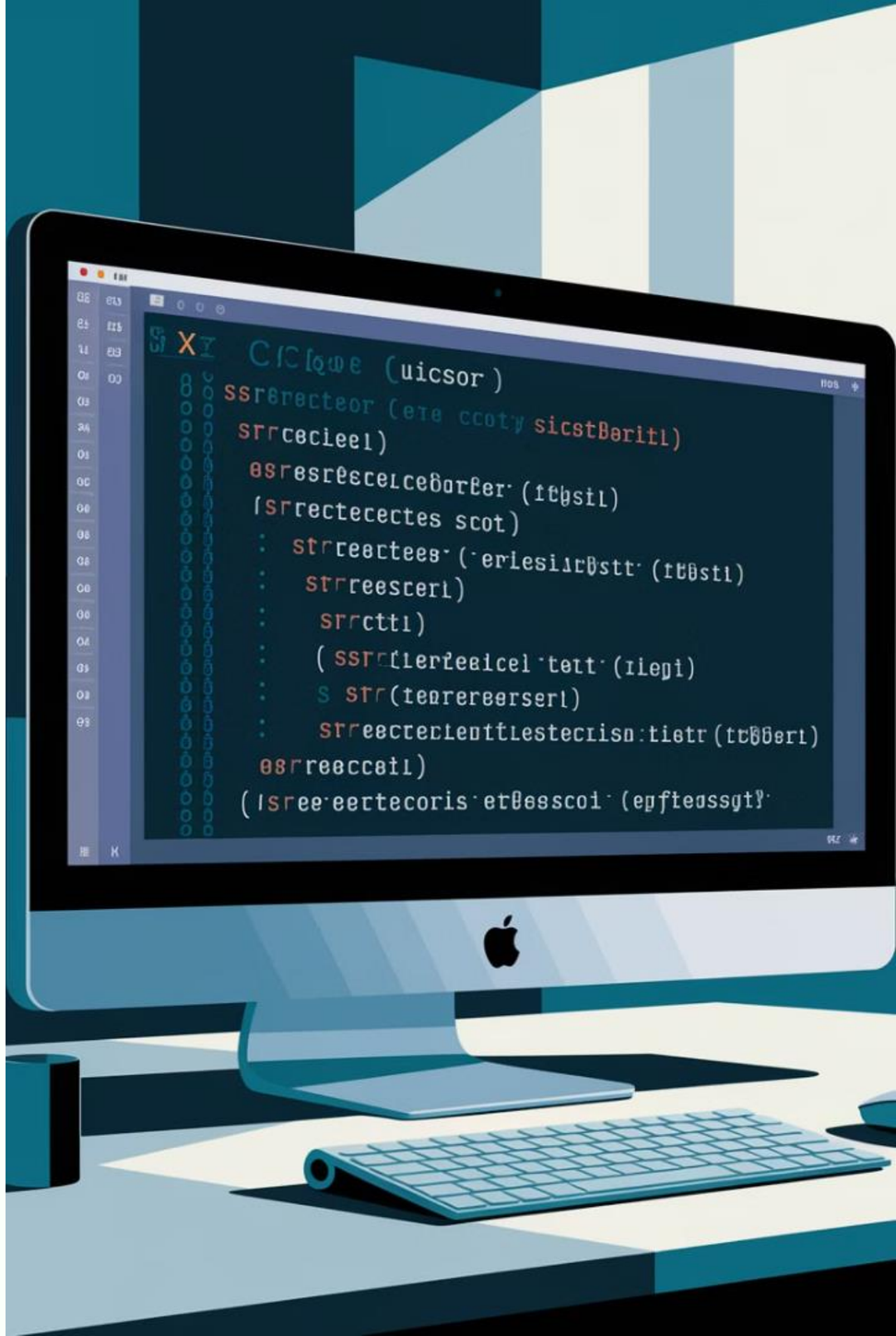


Estratégias de escolha do pivô



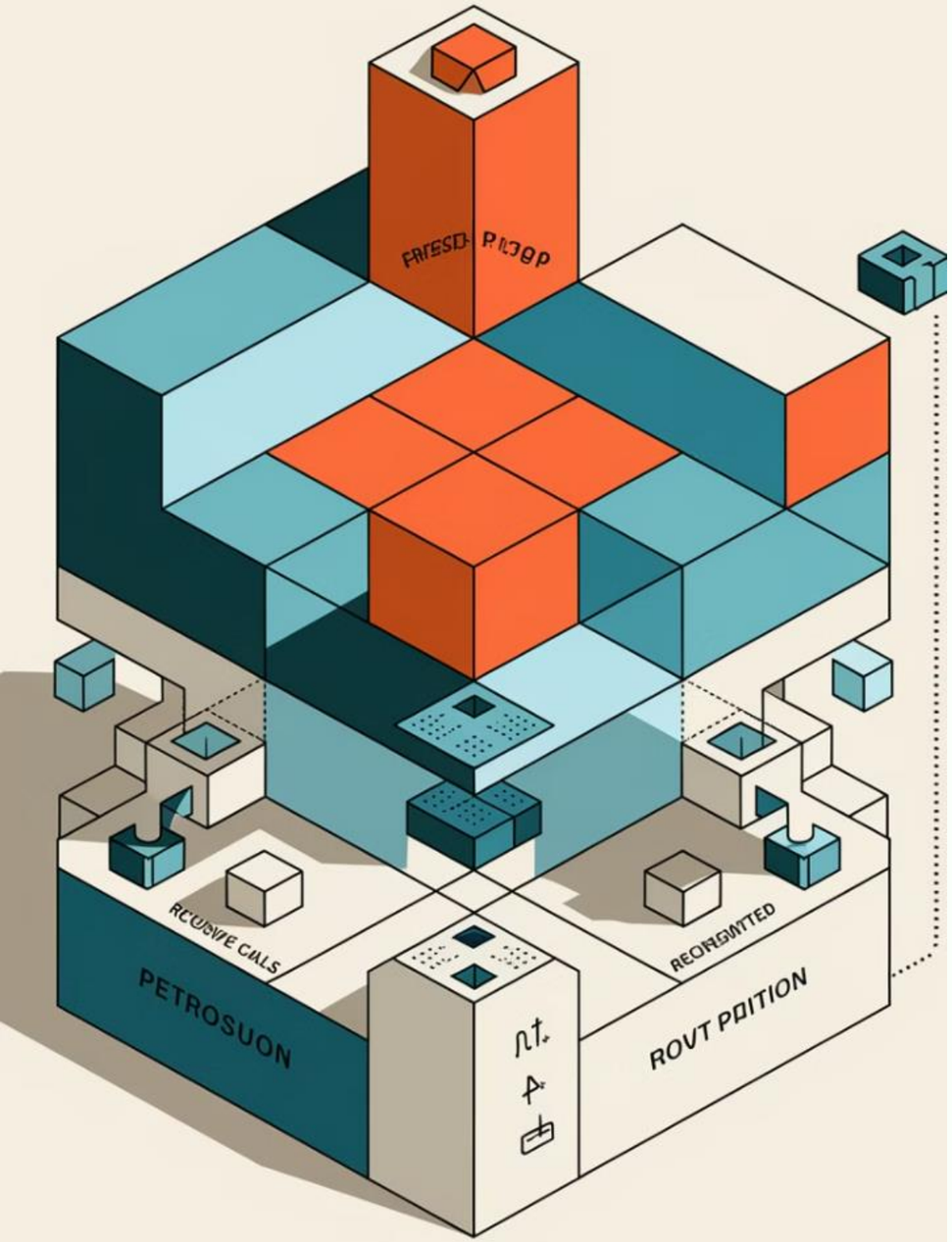
Implementação do QuickSort em C: função principal

```
void quickSort(int arr[], int low, int high) {
  if (low < high) {
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}
```



Implementação do particionamento em C

```
int partition(int arr[], int low, int high) {
  int pivot = arr[high];
  int i = (low - 1);
  for (int j = low; j <= high - 1; j++) {
    if (arr[j] < pivot) {
      i++;
      swap(arr, i, j);
    }
  }
  swap(arr, i + 1, high);
  return (i + 1);
}
```



Demonstração visual do QuickSort em ação

1

Passo 1

Escolher o pivô e particionar o array.

2

Passo 2

Ordenar recursivamente as sublistas à esquerda e à direita do pivô.

3

Passo 3

Combinar as sublistas ordenadas para obter o array completo ordenado.

Exemplo prático: executando QuickSort com array grande

Array de entrada

Um array com 1000 elementos aleatórios.

Saída após ordenação

O array ordenado com os 1000 elementos em ordem crescente.

Análise de complexidade do QuickSort

Melhor caso

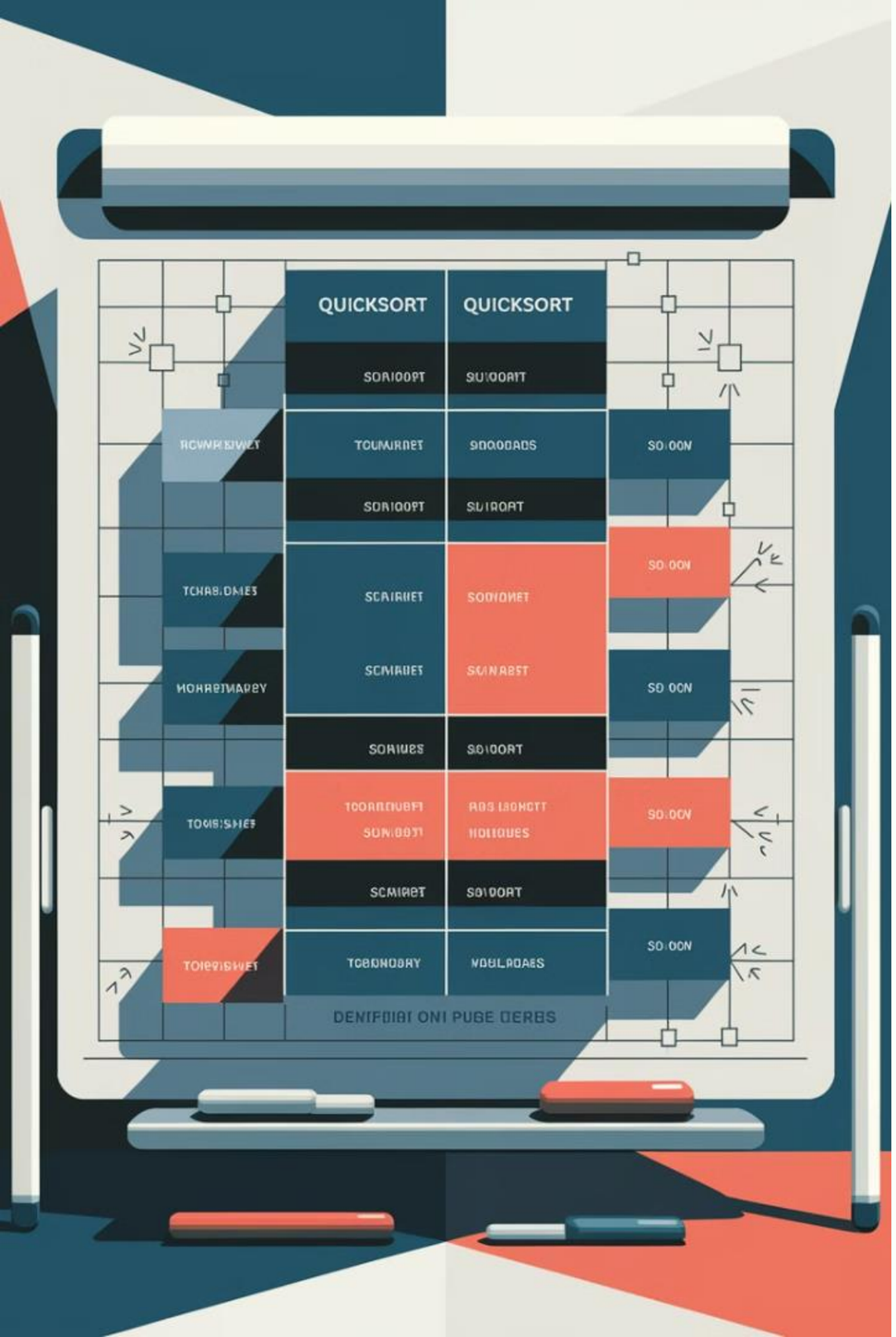
O(n log n): quando o pivô divide o array igualmente em cada etapa.

Pior caso

O(n^2): quando o pivô é sempre o menor ou maior elemento.

Caso médio

O(n log n): similar ao melhor caso, mas com variações.



Otimizações possíveis no QuickSort

Pivô aleatório

Escolher o pivô aleatoriamente para evitar o pior caso.

Quicksort híbrido

Usar InsertionSort para conjuntos de dados pequenos.

Tail recursion

Otimizar a recursividade para reduzir o uso de memória.

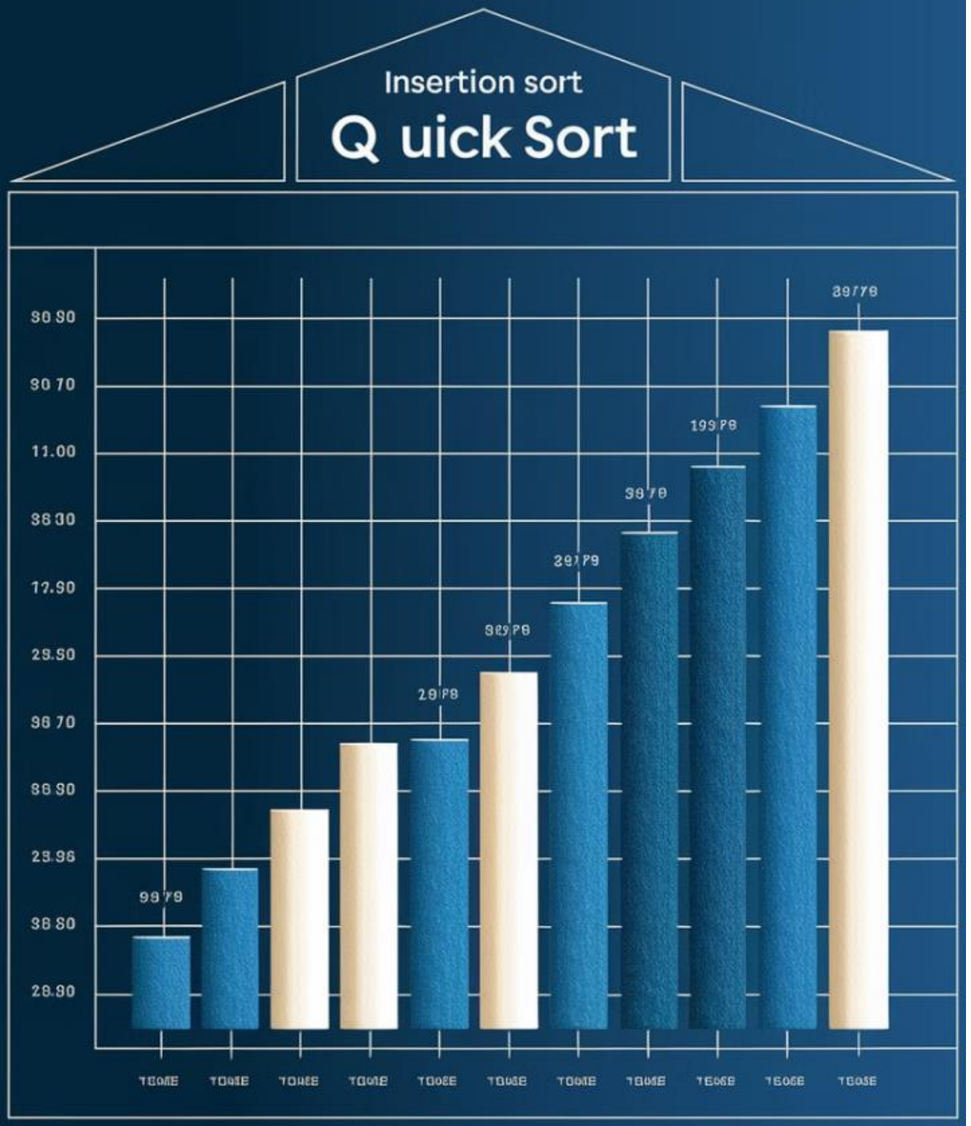
Comparação prática: InsertionSort vs QuickSort

InsertionSort

Eficiente para conjuntos de dados pequenos, mas lento para conjuntos grandes.

QuickSort

Eficiente para conjuntos de dados grandes, mas pode ser lento em casos específicos.



Análise de desempenho com diferentes tamanhos de entrada

Tamanho da Entrada	Tempo de execução (InsertionSort)	Tempo de execução (QuickSort)
10	0.001s	0.002s
100	0.01s	0.003s
1000	1s	0.01s
10000	100s	0.1s

Casos de uso ideais para cada algoritmo

1

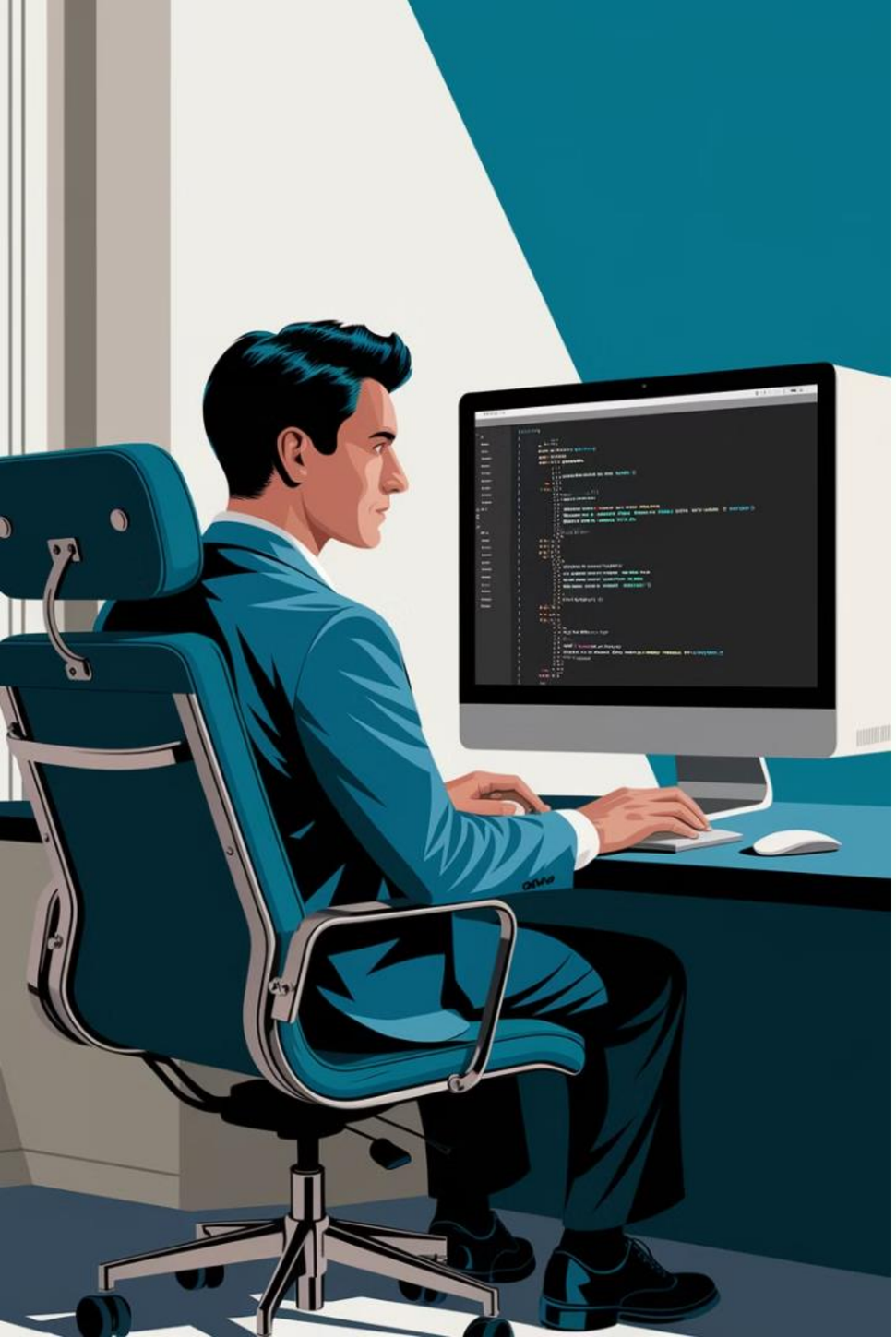
InsertionSort

Dados pequenos, já quase ordenados.

2

QuickSort

Dados grandes, aleatórios.



Exercício prático: implementação e teste

1

Passo 1

Implementar o código InsertionSort em C.

2

Passo 2

Criar um array de teste com diferentes elementos.

3

Passo 3

Executar o código e verificar a saída ordenada.

4

Passo 4

Repetir os passos 1-3 para o QuickSort.

Desafios comuns e como resolvê-los

Erros de lógica

Rever cuidadosamente a implementação do código e corrigir erros de lógica.

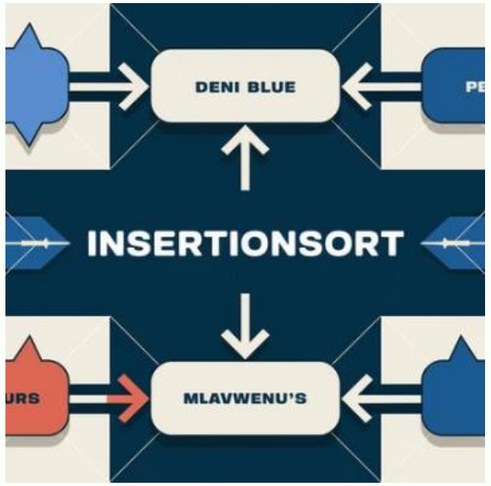
Segmentação de memória

Verificar se a memória alocada é suficiente e se o código está acessando memória fora dos limites.

Pivô ruim

Usar uma estratégia de escolha do pivô mais eficiente para evitar o pior caso.

Perguntas ?


DENI BLUE
INSERTIONSORT
MLAVWENU'S


Choose a pivot element
1
12
2
4
QUICKSORT
3
11
22
02
Recursively sort the sub-arrays