Algoritmos de Ordenação:

Inserção e Ordenação

Introdução a Programação | 2024.1

Prof Jacson Barbosa

Discente: Verônica Ribeiro Oliveira Palmeira

Ordenação por Inserção

1 Percorrer o Array

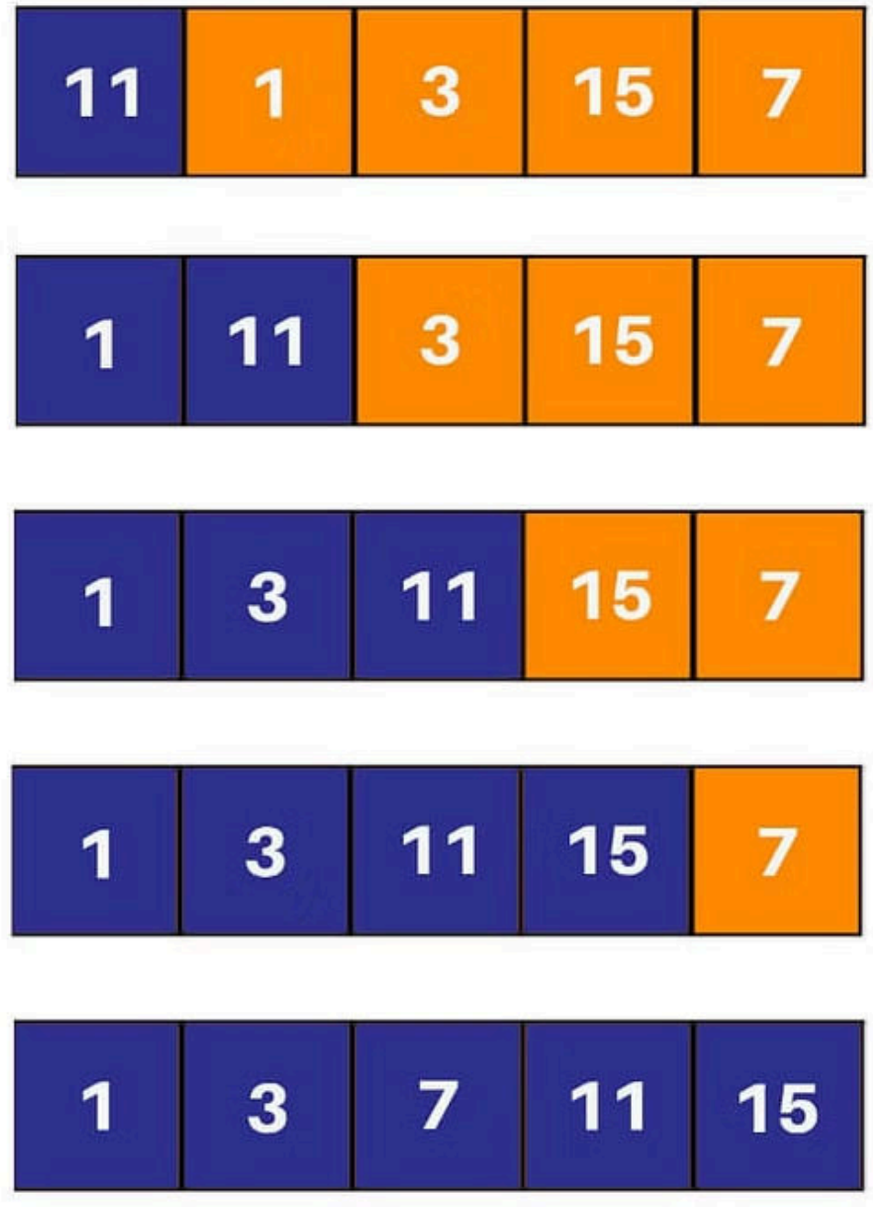
O algoritmo de ordenação por inserção percorre o array a partir do segundo elemento, comparando-o com os elementos anteriores e inserindo-o na posição correta, mantendo os elementos anteriores já ordenados.

2 Comparar e Inserir

Para cada elemento, o algoritmo compara seu valor com os elementos anteriores e o insere na posição correta, deslocando os demais elementos para a próxima posição.

3 Construir o Array Ordenado

Ao final de cada iteração, o subarray anterior fica ordenado, e o algoritmo continua o processo até que todo o array esteja ordenado.



Implementação em Go: Ordenação por Inserção

UFG

Função insertionSort

```
package main

import "fmt"

// Função para ordenar o array usando o algoritmo de ordenação por inserção
func insertionSort(array []int) {
    // Percorre o array a partir do segundo elemento
    for i := 1; i < len(array); i++ {
        valor := array[i]
        j := i - 1

        // Move os elementos maiores que a chave uma posição à frente
        for j >= 0 && array[j] > valor {
            array[j+1] = array[j]
            j = j - 1
        }
        array[j+1] = valor
    }
}

func main() {
    array := []int{5, 2, 8, 1, 9}
    fmt.Println("Array original:", array)
    insertionSort(array)
    fmt.Println("Array ordenado:", array)
}
```

Implementação em Go: Ordenação por Inserção

UFG

Função insertionSort

A função insertionSort() implementa o algoritmo de ordenação por inserção em Go. Ela percorre o array, compara o elemento atual com os anteriores e o insere na posição correta.

Explicação do Código

O código inclui loops for para percorrer o array, comparar os elementos e realizar as trocas necessárias. Ele utiliza uma variável temporária para armazenar o elemento atual durante o processo de inserção.

Eficiência e Uso

A ordenação por inserção é especialmente eficiente para pequenos conjuntos de dados e pode ser útil quando é necessário manter um array parcialmente ordenado atualizado.

Testando o Algoritmo

O código de exemplo inclui um array de entrada e a chamada da função insertionSort() para ordenar o array. Os resultados são impressos antes e depois da ordenação.

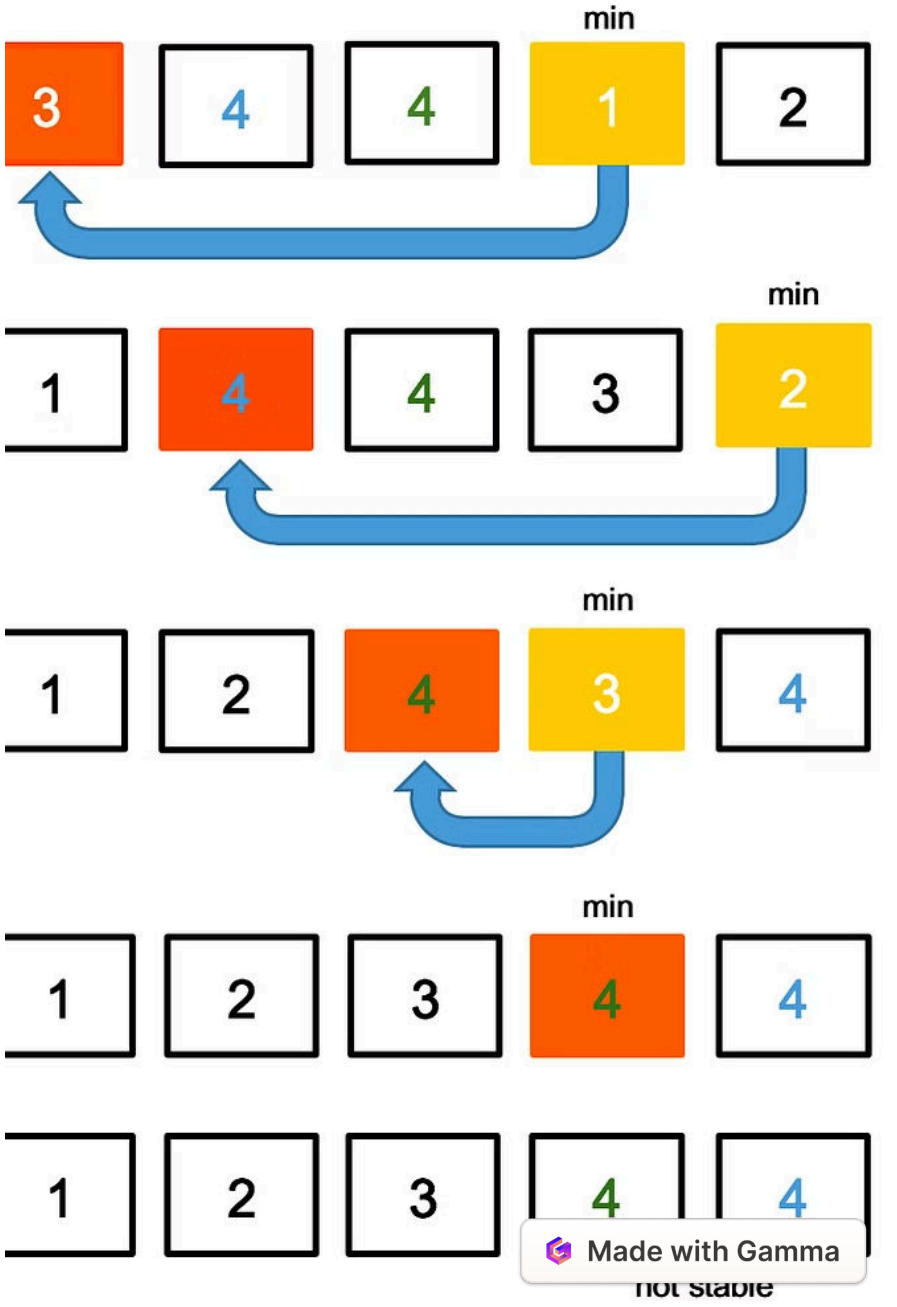
Ordenação por Seleção

Encontrar o Menor Elemento

Funciona selecionando o menor valor do conjunto e colocando-o na primeira posição. Em seguida, seleciona-se o próximo menor valor e o posiciona na segunda posição, e assim por diante até os últimos dois elementos.

Ordenar Iterativamente

O processo é repetido até que todo o array esteja ordenado, com o menor elemento na primeira posição, o segundo menor na segunda posição, e assim por diante.



Implementação em Go: Ordenação por Seleção

UFG

Função selectionSort

```
package main

import "fmt"

// Função para ordenar o array usando o algoritmo de ordenação por seleção
func selectionSort(array []int) {
    // Calcula o tamanho do array
    n := len(array)

    // Itera sobre todos os elementos, exceto os já ordenados
    for i := 0; i < n-1; i++ {
        // Encontra o índice do menor elemento na sublista não ordenada
        minIdx := i
        for j := i + 1; j < n; j++ {
            if array[j] < array[minIdx] {
                minIdx = j
            }
        }

        // Troca o menor elemento com o primeiro elemento da sublista não ordenada
        array[i], array[minIdx] = array[minIdx], array[i]
    }
}

func main() {
    array := []int{5, 2, 8, 1, 9}
    fmt.Println("Array original:", array)
    selectionSort(array)
    fmt.Println("Array ordenado:", array)
}
```

Implementação em Go: Ordenação por Seleção

UFG

Encontrar Menor

A função `selectionSort()` procura o menor elemento na sublista não ordenada.

Trocar Elementos

O menor elemento é então trocado com o primeiro da sublista não ordenada.

Explicação do Código

O código usa loops `for` para percorrer o array, encontrar o menor elemento e realizar a troca. Mantendo a variável temporária `minIdx` para armazenar o índice do menor elemento.

Complexidade $O(n^2)$

A ordenação por seleção possui complexidade de tempo quadrática.

Quando Usar Cada Algoritmo?

1

Dados Pequenos

A ordenação por inserção é a melhor escolha para conjuntos de dados pequenos, pois é simples de implementar e eficiente nesses casos.

2

Dados Parcialmente Ordenados

A ordenação por inserção também é ideal para manter atualizados arrays parcialmente ordenados, pois realiza um número mínimo de trocas.

3

Dados Grandes

Para conjuntos de dados maiores, a ordenação por seleção é geralmente mais eficiente, embora possua complexidade de tempo similar.

UFG

Considerações Finais

1 Ordenação por Inserção

Simples de implementar, eficiente para pequenos conjuntos de dados e excelente para listas parcialmente ordenadas.

2 Ordenação por Seleção

Requer menos trocas de elementos, é mais eficiente para grandes conjuntos de dados, mas geralmente mais lenta.

3 Escolha do Algoritmo

A escolha do algoritmo depende do tamanho e das características do conjunto de dados, bem como dos requisitos de desempenho do sistema.

4 Explorar Mais Algoritmos

Existem outros algoritmos de ordenação, como a ordenação por fusão, ordenação rápida e ordenação heap, cada um com suas próprias características e aplicações.