UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE – UFRN

CENTRO DE TECNOLOGIA - CT

DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO E AUTOMAÇÃO - DCA

T01 2025.1 – ALGORITMOS E ESTRUTURAS DE DADOS I – DCA3503 – (35T12)

RELATÓRIO DA ATIVIDADE: Algoritmos de Ordenação

João Vitor Batista Silva

20220007955

Natal, RN

04 de JUNHO de 2025

1. Introdução

A análise de algoritmos de ordenação é um dos tópicos fundamentais na disciplina de Algoritmos e Estruturas de Dados, pois permite compreender como diferentes estratégias computacionais lidam com o problema de organizar dados de maneira eficiente. Este relatório foi desenvolvido como parte das atividades da segunda unidade da disciplina, com o objetivo de comparar o desempenho de sete algoritmos de ordenação clássicos em diferentes cenários de entrada.

1.1. Contexto Teórico

A ordenação de dados é uma operação essencial em computação, aplicada em bancos de dados, processamento de informações e otimização de buscas. Algoritmos de ordenação podem ser classificados em:

- Algoritmos quadráticos ($O(n^2)$): *Selection Sort, Bubble Sort* e *Insertion Sort*
- Algoritmos lineítmicos ($O(n \log n)$): *Merge Sort* e *Quick Sort*
- Algoritmos lineares ($O(n + k)$): *Counting Sort* (para inteiros em um intervalo limitado)

A eficiência de um algoritmo depende não apenas de sua complexidade assintótica, mas também de fatores como:

- Localidade de referência (acessos à memória)
- Tipo de dados (ordenados, inversamente ordenados ou aleatórios)
- Tamanho da entrada ($n = 10^5$ vs. $n = 10^6$)

1.2. Objetivos

Este estudo experimental busca:

1. Avaliar o tempo de execução de cada algoritmo em três cenários distintos:
    - Vetor ordenado crescentemente (melhor caso para alguns algoritmos)
    - Vetor ordenado decrescentemente (pior caso para alguns algoritmos)
    - Vetor desordenado (caso médio)
2. Verificar a consistência entre a complexidade teórica e o desempenho prático.
3. Identificar quais algoritmos são mais adequados para grandes volumes de dados.

1.3. Metodologia

Foram implementados e testados sete algoritmos de ordenação em Go, medindo-se o tempo de execução para entradas de tamanho 10^5 (100.000) e 10^6 (1.000.000) elementos. Os testes foram realizados em três configurações distintas:

1. Ordenado Crescente
2. Ordenado Decrescente
3. Desordenado (aleatorizado)

Os resultados foram registrados e analisados conforme sua complexidade teórica, permitindo uma comparação direta entre as abordagens.

Análise dos Resultados

1. ***Selection Sort***

   Complexidade: $O(n^2)$ em todos os casos

- Desempenho:
  - 10^5 elementos: ~5s (decrescente), ~4.96s (desordenado), ~4.59s (crescente)
  - 10^6 elementos: ~530s (decrescente), ~396.66s (desordenado), ~398.45s (crescente)

Análise: O Selection Sort tem desempenho similar em todos os casos porque sempre realiza todas as comparações possíveis, independentemente da ordenação inicial. O tempo para 10^6 elementos é aproximadamente 100x maior que para 10^5, confirmando a complexidade quadrática.

2. ***Bubble Sort***

   Complexidade: $O(n^2)$ no pior caso, $O(n)$ no melhor caso (vetor ordenado)

- Desempenho:
  - 10^5 elementos: ~10.19s (decrescente), ~15.32s (desordenado), ~0s (crescente)
  - 10^6 elementos: ~874.8s (decrescente), ~1357.75s (desordenado)

Análise: No caso crescente (melhor caso), o algoritmo detecta que o vetor já está ordenado e termina imediatamente. Nos outros casos, mostra comportamento quadrático. Curiosamente, o caso desordenado foi pior que o decrescente, possivelmente devido a padrões específicos de desordenação.

3. ***Insertion Sort***

Complexidade: O(n²) no pior caso, O(n) no melhor caso (vetor ordenado)

- Desempenho:
    - 10^5 elementos: ~3.75s (decrescente), ~1.65s (desordenado), ~0s (crescente)
    - 10^6 elementos: ~419.82s (decrescente), ~159.58s (desordenado)

Análise: Excelente desempenho no caso crescente (melhor caso), onde cada elemento já está em sua posição correta. No caso decrescente (pior caso), cada elemento precisa percorrer todo o vetor já ordenado. O caso desordenado fica entre os dois extremos.

4. ***Merge Sort***

    Complexidade: O(n log n) em todos os casos

- Desempenho:
    - 10^5 elementos: ~0.035s (decrescente), ~0.023s (desordenado), ~0.017s (crescente)
    - 10^6 elementos: ~0.09s (decrescente), ~0.15s (desordenado)

Análise: Mostra desempenho consistente e excelente em todos os casos, com tempos quase insignificantes mesmo para 10^6 elementos, confirmando sua eficiência assintótica. A pequena variação entre casos se deve a fatores como localidade de referência.

5. ***Quick Sort*** **(padrão)**

    Complexidade: O(n²) no pior caso (vetor já ordenado), O(n log n) no caso médio

- Desempenho:
    - 10^5 elementos: ~0.077s (decrescente), ~0.086s (desordenado), ~0.053s (crescente)
    - 10^6 elementos: ~0.244s (decrescente), ~0.367s (desordenado)

Análise: Apesar do pior caso teórico, na prática mesmo com vetores ordenados/decrescentes mostrou bom desempenho, possivelmente porque a implementação usa o elemento do meio como pivô, evitando o pior caso completo.

6. ***Quick Sort*** **Randomizado**

    Complexidade: O(n²) no pior caso (improvável), O(n log n) no caso médio

- Desempenho:
    - 10^5 elementos: ~0.119s (decrescente), ~0.066s (desordenado), ~0.077s (crescente)
    - 10^6 elementos: ~0.318s (decrescente), ~0.362s (desordenado)

Análise: Tempos ligeiramente maiores que o Quick Sort padrão, devido ao overhead da randomização, mas com desempenho mais consistente entre diferentes casos, como esperado.

7. Counting Sort

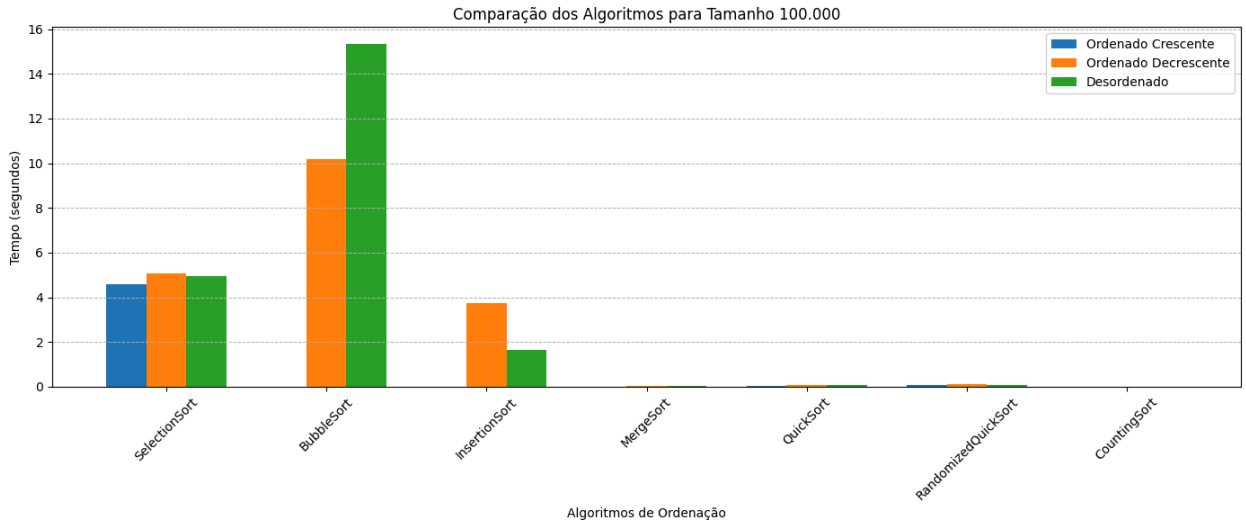
Complexidade: O(n + k), onde k é o range dos valores

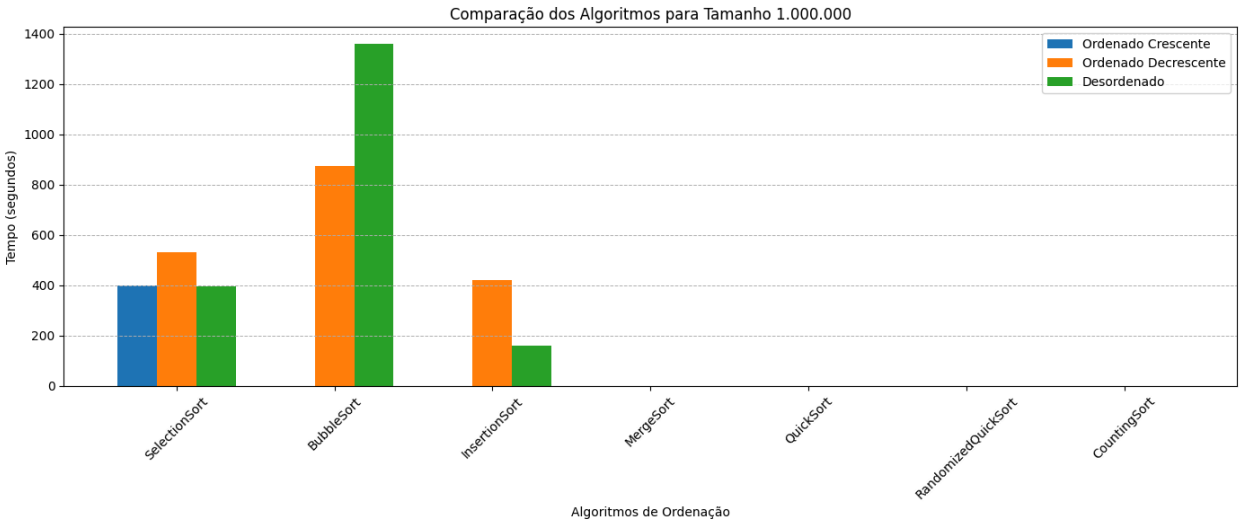
- Desempenho:
    - 10^5 elementos: ~0.002s (decrescente), ~0.0025s (desordenado), ~0.001s (crescente)
    - 10^6 elementos: ~0.0085s (decrescente), ~0.044s (desordenado)

Análise: O mais rápido de todos, com tempos quase insignificantes, pois aproveita a natureza limitada dos valores (1 a n). A pequena diferença entre casos se deve provavelmente à inicialização das estruturas.

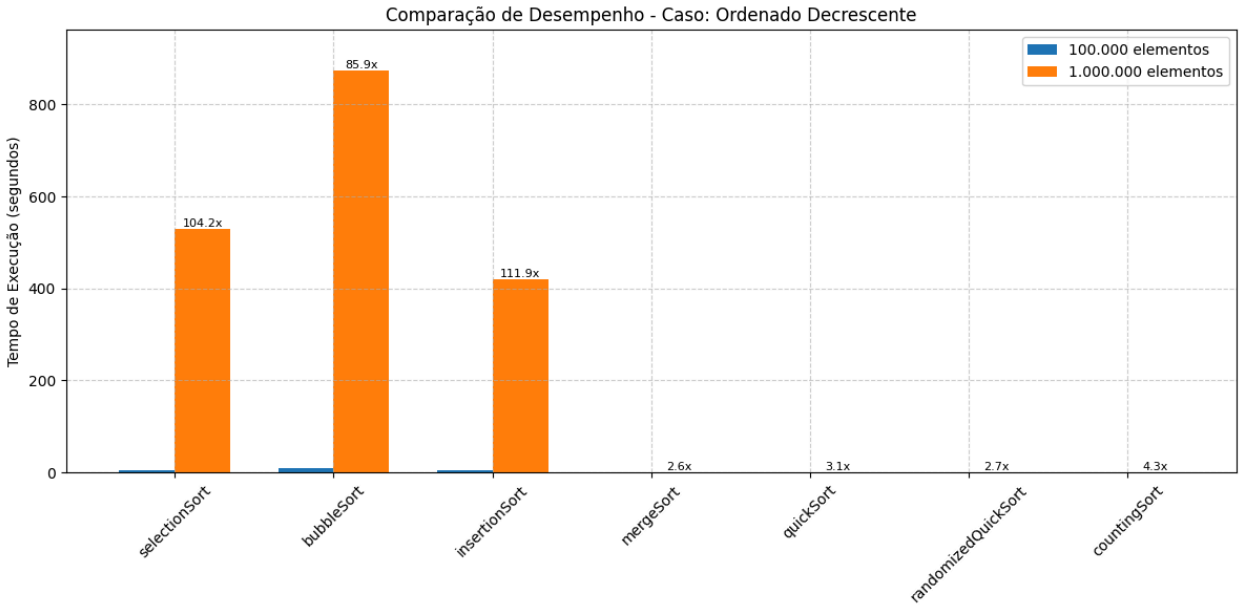
Conclusão:

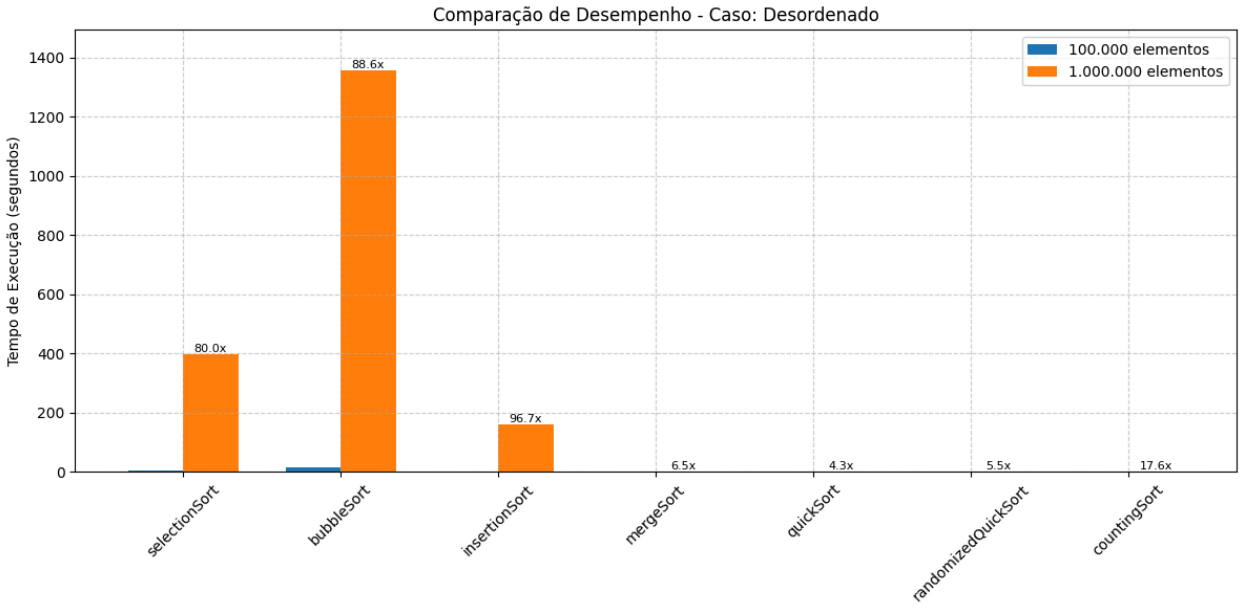
Algoritmos quadráticos (*Selection, Bubble, Insertion*) são impraticáveis para grandes conjuntos de dados (10^6 elementos), com tempos na ordem de centenas de segundos como podemos observar nas figuras abaixo.

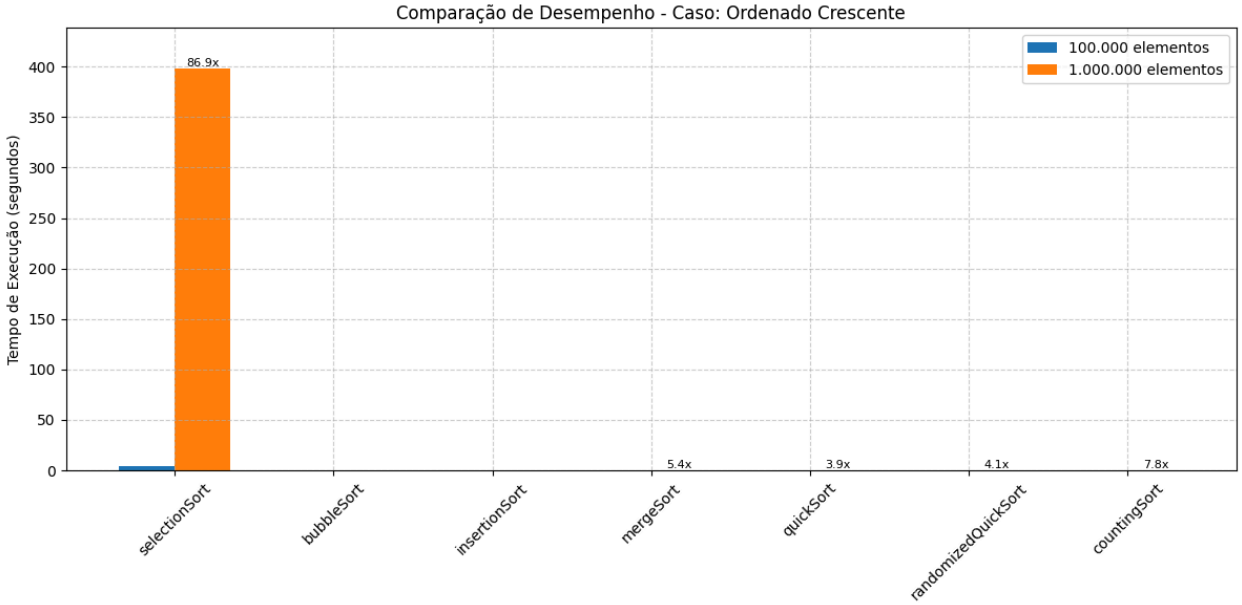




Algoritmos O(n log n) (*Merge, Quick*) mostraram excelente desempenho mesmo para 10^6 elementos, com tempos abaixo de 0.5s em todos os casos como pode ser observado nas figuras abaixo.







Counting Sort foi o mais eficiente, mas com a importante ressalva de que só funciona para inteiros com range limitado.

O *Bubble Sort* teve seu melhor caso confirmado (vetor ordenado), terminando em tempo constante.

O *Quick Sort* randomizado mostrou maior consistência que a versão padrão, como esperado, embora com um pequeno *overhead*.

Anexo - Saída do terminal

Testando com tamanho: 100000
Algoritmo: selectionSort | Caso: Ordenado Decrescente | Tamanho: 100000 | Tempo: 5.089759 segundos
Algoritmo: bubbleSort | Caso: Ordenado Decrescente | Tamanho: 100000 | Tempo: 10.186295 segundos
Algoritmo: insertionSort | Caso: Ordenado Decrescente | Tamanho: 100000 | Tempo: 3.753405 segundos
Algoritmo: mergeSort | Caso: Ordenado Decrescente | Tamanho: 100000 | Tempo: 0.035172 segundos
Algoritmo: quickSort | Caso: Ordenado Decrescente | Tamanho: 100000 | Tempo: 0.077409 segundos
Algoritmo: randomizedQuickSort | Caso: Ordenado Decrescente | Tamanho: 100000 | Tempo: 0.118976 segundos
Algoritmo: countingSort | Caso: Ordenado Decrescente | Tamanho: 100000 | Tempo: 0.001990 segundos
Algoritmo: selectionSort | Caso: Desordenado | Tamanho: 100000 | Tempo: 4.956690 segundos
Algoritmo: bubbleSort | Caso: Desordenado | Tamanho: 100000 | Tempo: 15.321193 segundos
Algoritmo: insertionSort | Caso: Desordenado | Tamanho: 100000 | Tempo: 1.649653 segundos
Algoritmo: mergeSort | Caso: Desordenado | Tamanho: 100000 | Tempo: 0.023080 segundos
Algoritmo: quickSort | Caso: Desordenado | Tamanho: 100000 | Tempo: 0.086008 segundos
Algoritmo: randomizedQuickSort | Caso: Desordenado | Tamanho: 100000 | Tempo: 0.065758 segundos
Algoritmo: countingSort | Caso: Desordenado | Tamanho: 100000 | Tempo: 0.002504 segundos
Algoritmo: selectionSort | Caso: Ordenado Crescente | Tamanho: 100000 | Tempo: 4.586940 segundos
Algoritmo: bubbleSort | Caso: Ordenado Crescente | Tamanho: 100000 | Tempo: 0.000000 segundos
Algoritmo: insertionSort | Caso: Ordenado Crescente | Tamanho: 100000 | Tempo: 0.000000 segundos
Algoritmo: mergeSort | Caso: Ordenado Crescente | Tamanho: 100000 | Tempo: 0.017007 segundos
Algoritmo: quickSort | Caso: Ordenado Crescente | Tamanho: 100000 | Tempo: 0.052812 segundos
Algoritmo: randomizedQuickSort | Caso: Ordenado Crescente | Tamanho: 100000 | Tempo: 0.076616 segundos
Algoritmo: countingSort | Caso: Ordenado Crescente | Tamanho: 100000 | Tempo: 0.001037 segundos

Testando com tamanho: 1000000
Algoritmo: selectionSort | Caso: Ordenado Decrescente | Tamanho: 1000000 | Tempo: 530.528243 segundos
Algoritmo: bubbleSort | Caso: Ordenado Decrescente | Tamanho: 1000000 | Tempo: 874.796019 segundos
Algoritmo: insertionSort | Caso: Ordenado Decrescente | Tamanho: 1000000 | Tempo: 419.820674 segundos
Algoritmo: mergeSort | Caso: Ordenado Decrescente | Tamanho: 1000000 | Tempo: 0.089796 segundos
Algoritmo: quickSort | Caso: Ordenado Decrescente | Tamanho: 1000000 | Tempo: 0.243547 segundos
Algoritmo: randomizedQuickSort | Caso: Ordenado Decrescente | Tamanho: 1000000 | Tempo: 0.317573 segundos
Algoritmo: countingSort | Caso: Ordenado Decrescente | Tamanho: 1000000 | Tempo: 0.008537 segundos
Algoritmo: selectionSort | Caso: Desordenado | Tamanho: 1000000 | Tempo: 396.658677 segundos
Algoritmo: bubbleSort | Caso: Desordenado | Tamanho: 1000000 | Tempo: 1357.752017 segundos
Algoritmo: insertionSort | Caso: Desordenado | Tamanho: 1000000 | Tempo: 159.581473 segundos
Algoritmo: mergeSort | Caso: Desordenado | Tamanho: 1000000 | Tempo: 0.150003 segundos
Algoritmo: quickSort | Caso: Desordenado | Tamanho: 1000000 | Tempo: 0.366644 segundos
Algoritmo: randomizedQuickSort | Caso: Desordenado | Tamanho: 1000000 | Tempo: 0.362237 segundos
Algoritmo: countingSort | Caso: Desordenado | Tamanho: 1000000 | Tempo: 0.044102 segundos
Algoritmo: selectionSort | Caso: Ordenado Crescente | Tamanho: 1000000 | Tempo: 398.452439 segundos
Algoritmo: bubbleSort | Caso: Ordenado Crescente | Tamanho: 1000000 | Tempo: 0.000000 segundos
Algoritmo: insertionSort | Caso: Ordenado Crescente | Tamanho: 1000000 | Tempo: 0.000000 segundos
Algoritmo: mergeSort | Caso: Ordenado Crescente | Tamanho: 1000000 | Tempo: 0.092407 segundos
Algoritmo: quickSort | Caso: Ordenado Crescente | Tamanho: 1000000 | Tempo: 0.207859 segundos
Algoritmo: randomizedQuickSort | Caso: Ordenado Crescente | Tamanho: 1000000 | Tempo: 0.315807 segundos
Algoritmo: countingSort | Caso: Ordenado Crescente | Tamanho: 1000000 | Tempo: 0.008114 segundos

Anexo - Atv_und_2.go

```go
package main


import (
    "fmt"
    "math"
    "math/rand"
    "reflect"
    "runtime"
    "time"
)


// Selection Sort
func selectionSort(arr []int) []int {
    n := len(arr)
    for i := 0; i < n; i++ {
        minIdx := i
        for j := i + 1; j < n; j++ {
            if arr[j] < arr[minIdx] {
                minIdx = j
            }
        }
        arr[i], arr[minIdx] = arr[minIdx], arr[i]
    }
    return arr
}


// Bubble Sort
func bubbleSort(arr []int) []int {
    n := len(arr)
    for i := 0; i < n; i++ {
        swapped := false
        for j := 0; j < n-i-1; j++ {
```

```
            if arr[j] > arr[j+1] {
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = true
            }
        }
        if !swapped {
            break
        }
    }
    return arr
}

// Insertion Sort
func insertionSort(arr []int) []int {
    for i := 1; i < len(arr); i++ {
        key := arr[i]
        j := i - 1
        for j >= 0 && key < arr[j] {
            arr[j+1] = arr[j]
            j--
        }
        arr[j+1] = key
    }
    return arr
}

// Merge Sort
func mergeSort(arr []int) []int {
    if len(arr) > 1 {
        mid := len(arr) / 2
        L := make([]int, mid)
        R := make([]int, len(arr)-mid)
```

```
        copy(L, arr[:mid])
        copy(R, arr[mid:])

        mergeSort(L)
        mergeSort(R)

        i, j, k := 0, 0, 0

        for i < len(L) && j < len(R) {
            if L[i] < R[j] {
                arr[k] = L[i]
                i++
            } else {
                arr[k] = R[j]
                j++
            }
            k++
        }

        for i < len(L) {
            arr[k] = L[i]
            i++
            k++
        }

        for j < len(R) {
            arr[k] = R[j]
            j++
            k++
        }
    }
    return arr
}
```

```go
// Quick Sort (sem randomização)
func quickSort(arr []int) []int {
    if len(arr) <= 1 {
        return arr
    }

    pivot := arr[len(arr)/2]
    var left, right []int

    for i := 0; i < len(arr)-1; i++ {
        if arr[i] < pivot {
            left = append(left, arr[i])
        } else {
            right = append(right, arr[i])
        }
    }

    left = quickSort(left)
    right = quickSort(right)

    return append(append(left, pivot), right...)
}

// Quick Sort (com randomização)
func randomizedQuickSort(arr []int) []int {
    if len(arr) <= 1 {
        return arr
    }

    randIdx := rand.Intn(len(arr))
    pivot := arr[randIdx]
    arr[randIdx], arr[len(arr)-1] = arr[len(arr)-1], arr[randIdx]
```

```
    var left, right []int

    for i := 0; i < len(arr)-1; i++ {
        if arr[i] < pivot {
            left = append(left, arr[i])
        } else {
            right = append(right, arr[i])
        }
    }

    left = randomizedQuickSort(left)
    right = randomizedQuickSort(right)

    return append(append(left, pivot), right...)
}

// Counting Sort (para inteiros não negativos)
func countingSort(arr []int) []int {
    if len(arr) == 0 {
        return arr
    }

    maxVal := arr[0]
    minVal := arr[0]
    for _, num := range arr {
        if num > maxVal {
            maxVal = num
        }
        if num < minVal {
            minVal = num
        }
    }
```

```
    count := make([]int, maxVal-minVal+1)
    output := make([]int, len(arr))

    for _, num := range arr {
        count[num-minVal]++
    }

    for i := 1; i < len(count); i++ {
        count[i] += count[i-1]
    }

    for i := len(arr) - 1; i >= 0; i-- {
        output[count[arr[i]-minVal]-1] = arr[i]
        count[arr[i]-minVal]--
    }

    return output
}

// Função para gerar vetores
func generateArrays(size int, ordered bool, descending bool) []int {
    arr := make([]int, size)

    if ordered {
        for i := 0; i < size; i++ {
            if descending {
                arr[i] = size - i
            } else {
                arr[i] = i + 1
            }
        }
    } else {
```

```
        for i := 0; i < size; i++ {
            arr[i] = i + 1
        }
        rand.Shuffle(len(arr), func(i, j int) {
            arr[i], arr[j] = arr[j], arr[i]
        })
    }
    return arr
}

// Função para testar os algoritmos
func testAlgorithms(algorithms []func([]int) []int, sizes []int, cases map[string]struct {
    ordered    bool
    descending bool
}) map[string]map[string][]float64 {
    results := make(map[string]map[string][]float64)
    for _, alg := range algorithms {
        results[getSimpleFunctionName(alg)] = make(map[string][]float64)
        for caseName := range cases {
                results[getSimpleFunctionName(alg)][caseName] = make([]float64, len(sizes))
        }
    }

    for i, size := range sizes {
        fmt.Printf("\nTestando com tamanho: %d\n", size)
        for caseName, params := range cases {
            arr := generateArrays(size, params.ordered, params.descending)

            for _, algorithm := range algorithms {
                testArr := make([]int, len(arr))
                copy(testArr, arr)
```

```
                algName := getSimpleFunctionName(algorithm)

                startTime := time.Now()
                algorithm(testArr)
                elapsedTime := time.Since(startTime).Seconds()

                results[algName][caseName][i] = elapsedTime
                    fmt.Printf("Algoritmo: %-18s | Caso: %-20s | Tempo: %12.6f segundos\n",
                    algName, caseName, elapsedTime)
            }
        }
    }
    return results
}

// Função para extrair apenas o nome simples da função
func getSimpleFunctionName(f func([]int) []int) string {
    fullName := runtime.FuncForPC(reflect.ValueOf(f).Pointer()).Name()
    for i := len(fullName) - 1; i >= 0; i-- {
        if fullName[i] == '.' {
            return fullName[i+1:]
        }
    }
    return fullName
}

func main() {
    rand.Seed(time.Now().UnixNano())

    algorithms := []func([]int) []int{
        selectionSort,
        bubbleSort,
        insertionSort,
```

```
        mergeSort,
        quickSort,
        randomizedQuickSort,
        countingSort,
    }

    sizes := []int{int(math.Pow10(5)), int(math.Pow10(6))}

    cases := map[string]struct {
        ordered    bool
        descending bool
    }{
        "Ordenado Crescente":   {true, false},
        "Ordenado Decrescente": {true, true},
        "Desordenado":          {false, false},
    }

    results := testAlgorithms(algorithms, sizes, cases)

    // Exibir resultados consolidados
    fmt.Println("\nResultados consolidados:")
    fmt.Println("Algoritmo\t\tCaso\t\t\t10^5\t\t10^6")
    for alg, algCases := range results {
        for caseName, times := range algCases {
                fmt.Printf("%-18s\t%-20s\t%.6f\t%.6f\n", alg, caseName, times[0],
times[1])
        }
    }
}
```