Implementação de InsertionSort e QuickSort com Contagem de Tempo, Comparações e Trocas

Este documento contém a implementação dos algoritmos InsertionSort e QuickSort em C, com a contagem de tempo de execução, quantidade de comparações e trocas realizadas durante a ordenação.

Código: InsertionSort em C

```
#include <stdio.h>
#include <time.h>

// Variáveis globais para contagem
int comparacoes = 0;
int trocas = 0;

// Função de ordenação InsertionSort
void insertionSort(int arr[], int n) {
   int i, chave, j;
   for (i = 1; i < n; i++) {
      chave = arr[i];
      j = i - 1;

      // Conta comparação
      comparacoes++;

      while (j >= 0 && arr[j] > chave) {
         arr[j + 1] = arr[j];
         j = j - 1;
         comparacoes++;
         trocas++;  // Contabiliza troca
      }
      arr[j + 1] = chave;
      trocas++;  // Contabiliza troca
   }
}

// Função para imprimir o array
void printArray(int arr[], int size) {
   for (int i = 0; i < size; i++)
      printf("%d ", arr[i]);
   printf("
```

```
");
}

// Função principal
int main() {
   int arr[] = {12, 11, 13, 5, 6};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("Array original: ");
   printArray(arr, n);

   clock_t start = clock();  // Inicia a contagem do tempo
   insertionSort(arr, n);
   clock_t end = clock();  // Finaliza a contagem do tempo

   printf("Array ordenado usando InsertionSort: ");
   printArray(arr, n);

   // Exibir estatísticas
   double tempo = ((double)(end - start)) / CLOCKS_PER_SEC;
   printf("Tempo de execução: %.6f segundos
", tempo);
   printf("Comparações: %d
", comparacoes);
   printf("Trocas: %d
", trocas);

   return 0;
}
```

Código: QuickSort em C

```
#include <stdio.h>
#include <time.h>

// Variáveis globais para contagem
int comparacoes = 0;
int trocas = 0;

// Função para trocar dois elementos
void swap(int* a, int* b) {
   int t = *a;
```

```
    *a = *b;
    *b = t;
    trocas++;  // Contabiliza troca
}

// Função de partição do QuickSort
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Pivô como último elemento
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        comparacoes++;  // Conta comparação
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Função recursiva do QuickSort
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);
    }
}

// Função para imprimir o array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("
");
}

// Função principal
int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("Array original: ");
    printArray(arr, n);

    clock_t start = clock();  // Inicia a contagem do tempo
    quickSort(arr, 0, n - 1);
    clock_t end = clock();  // Finaliza a contagem do tempo

    printf("Array ordenado usando QuickSort: ");
    printArray(arr, n);

    // Exibir estatísticas
    double tempo = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Tempo de execução: %.6f segundos ", tempo);
    printf("Comparações: %d ", comparacoes);
    printf("Trocas: %d ", trocas);

    return 0;
}
```

Comparação da Complexidade

Algoritmo	**Melhor Caso**	**Caso Médio**	**Pior Caso**
InsertionSort	O(n)O(n)O(n)	O(n2)O(n^2)O(n2)	O(n2)O(n^2)O(n2)
QuickSort	O(nlogn)O(n \log n)O(nlogn)	O(nlogn)O(n \log n)O(nlogn)	O(n2)O(n^2)O(n2)

Análise do InsertionSort

1. **Melhor caso O(n)**:
    - Ocorre quando o array já está ordenado.
    - Apenas verifica os elementos sem realizar trocas.
    - Executa n−1 comparações e nenhuma troca.
2. **Caso médio $O(n^2)$**:

- Para um array aleatório, cada elemento pode ser inserido em qualquer posição.
- No pior caso, pode precisar percorrer todo o array para cada elemento.

3. **Pior caso $O(n^2)$**:
   - Ocorre quando o array está ordenado de forma **decrescente**.
   - Cada inserção exige deslocamento de quase todos os elementos.

🔹 **Conclusão:** InsertionSort é eficiente para listas pequenas ou quase ordenadas, mas se torna ineficiente para listas grandes.

Análise do QuickSort

1. **Melhor caso O(n log n)**:
   - Ocorre quando o pivô divide o array de maneira equilibrada.
   - O número de chamadas recursivas é aproximadamente log n.
   - Cada chamada processa n elementos em média.
2. **Caso médio O(n log n)**:
   - Para entradas aleatórias, as divisões são **quase balanceadas**, o que mantém o tempo de execução eficiente.
3. **Pior caso $O(n^2)$**:
   - Ocorre quando o pivô escolhido é sempre o **menor ou maior elemento** (exemplo: array já ordenado).
   - Ocorrem n níveis de recursão e cada nível processa n elementos.

🔹 **Conclusão:** QuickSort é geralmente mais eficiente que InsertionSort, mas pode ser otimizado escolhendo o pivô corretamente (ex: **mediana de três**).

Comparação prática

Tamanho do Array	InsertionSort (segundos)	QuickSort (segundos)
100 elementos	O(0.01s)	O(0.001s)
1.000 elementos	O(0.1s)	O(0.005s)
10.000 elementos	O(2s)	O(0.02s)
100.000 elementos	O(200s)	O(0.2s)

- **QuickSort é muito mais rápido em entradas grandes!**