Instituto Federal de Educação, Ciência e Tecnologia – Monte Castelo
Algoritmo e Estrutura de Dados II
Franciele Alves da Silva (20231SI0012)

Relatório

Código:

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

void insertionSort(int arr[], int size) {
  int i, key, j;
  for (i = 1; i < size; i++) {
    key = arr[i];
    j = i - 1;

    while (j >= 0 && key < arr[j]) {
      arr[j + 1] = arr[j];
      j = j - 1;
    }
    arr[j + 1] = key;
  }
}

void merge(int arr[], int left, int mid, int right) {
  int i, j, k;
  int n1 = mid - left + 1;
  int n2 = right - mid;
  int L[n1], R[n2];

  for (i = 0; i < n1; i++)
    L[i] = arr[left + i];
  for (j = 0; j < n2; j++)
    R[j] = arr[mid + 1 + j];

  i = 0;
  j = 0;
  k = left;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = R[j];
      j++;
    }
    k++;
  }
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
  }
}

void mergeSort(int arr[], int left, int right) {
  if (left < right) {
    int mid = left + (right - left) / 2;

    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
  }
}
```

```
void shellSort(int arr[], int n) {
   int gap, i, j, temp;
   for (gap = n / 2; gap > 0; gap /= 2) {
      for (i = gap; i < n; i++) {
         temp = arr[i];
         for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
            arr[j] = arr[j - gap];
         }
         arr[j] = temp;
      }
   }
}

void gRandomArray(int arr[], int n) {
  int i;
  srand(time(NULL));
  for (i = 0; i < n; i++)
    arr[i] = rand() % 100;
}

void comparaAlgoritimo(int n) {
  int *arr_ins = (int *)malloc(n * sizeof(int));
  int *arr_merge = (int *)malloc(n * sizeof(int));
  int *arr_shell = (int *)malloc(n * sizeof(int));

  gRandomArray(arr_ins, n);
  for (int i = 0; i < n; i++) {
    arr_merge[i] = arr_ins[i];
    arr_shell[i] = arr_ins[i];
  }

  clock_t insertion_ini = clock();
  insertionSort(arr_ins, n);
  clock_t insertion_fim = clock();
  double insertion_tempo = ((double)(insertion_fim - insertion_ini)) / CLOCKS_PER_SEC;

  printf("Insertion Sort: %f segundos\n", insertion_tempo);


  clock_t merge_ini = clock();
  mergeSort(arr_merge, 0, n - 1);
  clock_t merge_fim = clock();
  double merge_tempo = ((double)(merge_fim - merge_ini)) / CLOCKS_PER_SEC;

  printf("Merge Sort: %f segundos\n", merge_tempo);

  clock_t shell_ini = clock();
  shellSort(arr_shell, n);
  clock_t shell_fim = clock();
  double shell_tempo = ((double)(shell_fim - shell_ini)) / CLOCKS_PER_SEC;

  printf("Shell Sort: %f segundos\n", shell_tempo);

  free(arr_ins);
  free(arr_merge);
  free(arr_shell);
}

int main() {
  int n = 500000;

  comparaAlgoritimo(n);

  return 0;
}
```

Merge Sort

Tempo gasto para n = 500.000: 0.079000 segundos

Insertion Sort

Tempo gasto para n = 500.000: 105.912000 segundos;

Shell Sort

Tempo gasto para n = 500.000: 0.094000 segundos;

Conclusão:

- **Eficiência dos algoritmos:** O tempo de execução do Insertion Sort é significativamente maior em comparação com os outros dois algoritmos, especialmente para um conjunto de dados tão grande. Isso sugere que o Insertion Sort tem uma complexidade temporal muito pior em relação ao número de elementos a serem ordenados.
- **Comparação entre Merge Sort e Shell Sort:** Mesmo que o Merge Sort e o Shell Sort tenham tempos de execução próximos, o Merge Sort é ligeiramente mais rápido neste caso específico. No entanto, a diferença entre eles é marginal e pode variar dependendo dos dados de entrada. Ambos os algoritmos têm uma eficiência significativamente melhor em comparação com o Insertion Sort para grandes conjuntos de dados.