**RESPOSTAS SEMANA 6**

1 -

Conseguimos verificar da seguinte forma, se a modificação é capaz de identificar um subgrupo já ordenado, não tem mais a necessidade de realizar o mergesort nesse subgrupo, diante disso, ocorre uma melhoria na performance do mergesort. O mergesort é um algoritmo que requer mais computacionalmente no processo de intercalação dos elementos. Se um vetor de números já está ordenado, em consequência o Mergesort só irá empenhar-se na realização do merge desse vetor com um outro vetor, fundamentando-se que o outro vetor foi ordenado previamente pela recursividade do algoritmo.

No entanto, tratando-se das comparações que esse subgrupo irá processar com um outro subgrupo para compor um grupo, não ocorrerá diferença, porque a intercalação irá dar-se entre todos os elementos.

Em conclusão, se a modificação identifica o ordenamento de um subgrupo pai, não ocorrerá a necessidade de subgrupos filhos, e logo não ocorrerá intercalação entre os subgrupos filhos, entretanto se o ponto de vista for a intercalação de comparações do subgrupo com outro subgrupo, não haverá diferenças nas comparações de intercalação.

---

2 - Resposta **a** e **b** no main abaixo

```c
#include <stdio.h>
#include <stdlib.h>
#include "simpio.h"
#include "random.h"

#define TAM_INI 10000
#define TAM_MAX 100000
#define INC 200

// simpio.h e random.h veem da biblioteca de
// Eric Roberts, acessível por exemplo em
// https://www.ime.usp.br/~pf/Roberts/C-library/standard/

typedef int *veti; // Apelida de "veti" ("vetor de inteiros") o tipo
"int *".

veti criaVeti(int);
int busca (int, int, veti);
int maximo (int, veti);
int twosum (int, int, veti);
```

```
void insertionSort(int, veti);
int buscaBinaria(int, int, veti);

// Merge sort padrão
void mergeSort(int, veti);
void topDownMergeSort(int, int, veti);

// Merge insertion sort
void mergeInsertionSort(int, veti);
void topDownMergeInsertionSort(int, int, veti);

int NOP;

int main() {

    int tamanho, chave, i, r;
    veti vetor;

    Randomize();

    FILE *saida;

    // Mude o nome do arquivo de saida
    saida = fopen ("dadosMergeSort.dat", "w");
    //saida = fopen ("dadosMergeInsertionSort15.dat", "w");


    for(tamanho = TAM_INI; tamanho <= TAM_MAX; tamanho += INC) {
        vetor = criaVeti(tamanho);

        for(i = 0; i < tamanho; i++)
            vetor[i] = RandomInteger(0, TAM_MAX);

        // Sorteia uma chave existente no vetor
        chave = vetor[RandomInteger(0, tamanho)];

        // r = maximo(tamanho, vetor);
        // r = twosum(chave, tamanho, vetor);

        // Busca linear
        //r = busca(chave, tamanho, vetor);

        // Habilite as duas linhas para busca binaria:
```

```
        // insertionSort(tamanho, vetor);
        // r = buscaBinaria(chave, tamanho, vetor);

        mergeSort(tamanho, vetor);
        //mergeInsertionSort(tamanho, vetor);

        fprintf(saida, "%d %d\n", tamanho, NOP);

        free(vetor);
    }

    fclose(saida);

    return 0;
}

int busca (int chave, int tam, int vetor[]) {
    int i;
    i = tam - 1;
    NOP = 0;

    while (i >= 0 && vetor[i] != chave) {
        i -= 1;
        NOP++;
    }

    return i;
}

int maximo (int tam, int vetor[]) {
    int i, m = vetor[0];
    NOP = 0;
    for(i = 1; i < tam; i++) {
        NOP++;
        if(vetor[i] > m)
            m = vetor[i];
    }
    return m;
}

int twosum (int chave, int tam, int vetor[]) {

    int i, j, r;
```

```
    NOP = 0;
    r = 0;

    for(i = 0; i < tam; i++)
        for(j = 0; j < tam; j++) {
            NOP++;

            if(vetor[i] + vetor[j] == chave)
                r = 1;
        }

    return r;
}

void insertionSort(int tam, veti vetor){
    int i, j, chave;

    NOP = 0;

    for(i = 1; i < tam; ++i){
        chave = vetor[i];

        for(j = i -1; j >= 0 && vetor[j] > chave; --j){
            vetor[j+1] = vetor[j];
            NOP++;
        }

        vetor[j+1] = chave;
    }
}

int buscaBinaria(int chave, int tam, veti vetor){
    int ini = 0, fim = tam - 1, meio;

    NOP = 0;

    while(ini <= fim){
        meio = (ini + fim)/2;

        NOP++;

        if(chave == vetor[meio])
```

```
            return meio;
        if(chave < vetor[meio])
            fim = meio - 1;
        else
            ini = meio + 1;
    }

    return -1;
}

// Merge Sort padrão
void mergeSort(int tamanho, veti vetor){
    NOP = 0;

    topDownMergeSort(0, tamanho-1, vetor);
}

void topDownMergeSort(int ini, int fim, veti vetor){
    if (fim <= ini)
        return;

    // Encontra o meio do lista
    int meio = (ini + fim)/2;

    // Sepera na metade e usa o mergeSort em cada sublista
    topDownMergeSort(vetor, ini, meio);
    topDownMergeSort(vetor, meio+1, fim);

    int i, j, k;
    veti aux = criaVeti(fim+1);

    // Inicia mesclagem

    // Copia lista [ini : fim] para lista auxiliar
    for(i = ini; i <= fim; i++){
        aux[i] = vetor[i];
        NOP++;
    }

    // Inicializa indices
    i = ini, j = meio + 1, k = ini;

    // Começa mesclagem:
```

```
    while(i <= meio && j <= fim){
        NOP++;
        if(aux[i] < aux[j]){
            vetor[k++] = aux[i++];
            NOP++;
        }
        else{
            vetor[k++] = aux[j++];
            NOP++;
        }
    }

    // Copia os elementos restantes:
    while(i <= meio){
        vetor[k++] = aux[i++];
        NOP++;
    }

    while(j <= fim){
        vetor[k++] = aux[j++];
        NOP++;
    }
}

// Merge Sort padrão
void mergeInsertionSort(int tamanho, veti vetor){
    NOP = 0;

    topDownMergeInsertionSort(0, tamanho-1, vetor);
}

void topDownMergeInsertionSort(int ini, int fim, veti vetor){
    int i, j, chave, k = 15;

    // Se o tamanho do sublista for menor que k
    if (fim - ini + 1 < k){
        // Ordena o sublista por insertionSort

        for(i = ini; i <= fim; ++i){
            chave = vetor[i];
            NOP++; // Copia de valor

            for(j = i-1; j >= ini && vetor[j] > chave; --j){
```

```
                vetor[j+1] = vetor[j];
                NOP++; // Copia de valor
            }

            vetor[j+1] = chave;
            NOP++; // Copia de valor
        }
        return;
    }


    if (fim <= ini)
        return;

    // Encontra o meio do lista
    int meio = (ini + fim)/2;

    // Sepera na metade e usa o mergeSort em cada sublista
    topDownMergeSort(vetor, ini, meio);
    topDownMergeSort(vetor, meio+1, fim);

    veti aux = criaVeti(fim+1);

    // Inicia mesclagem

    // Copia lista [ini : fim] para lista auxiliar
    for(i = ini; i <= fim; i++){
        aux[i] = vetor[i];
        NOP++;
    }

    // Inicializa indices
    i = ini, j = meio + 1, k = ini;

    // Começa mesclagem:
    while(i <= meio && j <= fim){
        NOP++;
        if(aux[i] < aux[j]){
            vetor[k++] = aux[i++];
            NOP++;
        } else{
            vetor[k++] = aux[j++];
            NOP++;
```

```
        }
    }

    // Copia os elementos restantes:
    while(i <= meio){
        vetor[k++] = aux[i++];
    }

    while(j <= fim){
        vetor[k++] = aux[j++];
    }
}

// Aloca um vetor de inteiros de tamanho n.
// Devolve o ponteiro para esse vetor.
veti criaVeti(int tam) {
   return (int *) malloc(tam * sizeof(int));
}
```

---

c.

Para escolhermos o K temos que analisar a performance do algoritmo insertionSort, uma vez que o algoritmo de Mergesort em todas as situações apresentará a mesma performance independentemente de qual caso seja. (melhor caso, pior caso ou caso médio)

Investigando um pouco mais o insertionSort ele possui um melhor caso e um pior caso. Sendo eles:

- quando o vetor está pouco desordenado, ele tem a sua melhor performance. O(n);
- quando o vetor está muito desordenado, ele tem  a sua pior performance. $O(n^2)$.

---

d.

O resultado apresentado para os dois algoritmos é facultado pelas entradas que foram arranjadas aleatoriamente. Conforme mostrado no gráfico abaixo, o MergeInsertionSort tende a performar melhor que o MergeSort, já que beneficia-se da utilização do InsertionSort para a ordenação de sub-listas.

O MergeInsertionSort é um algoritmo híbrido, assim tem a benesse das partes menores serem ordenadas pelo InsertionSort e posteriormente mescladas usando Mergesort.

à vista disso, ele divide o vetor em sub-vetores de um tamanho pré-determinado, ordena estes sub-vetores com o Insertionsort e, após a ordenação dos sub-vetores, eles são combinados via Mergesort em pares, aumentando o tamanho dos pares até o vetor inteiro ser combinado.