UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE CRATEÚS

CRT0009 – ESTRUTURA DE DADOS

TRABALHO 03 – MÓDULO 03: ALGORITMOS DE ORDENAÇÃO

DISCENTES:

JOÃO LUCAS BARRETO LACERDA

DOCENTE: SIMONE DE OLIVEIRA SANTOS

FEVEREIRO DE 2022

Algoritmo: *Bubble Sort*

Também conhecido como ordenação por flutuação (daí vem o nome de bolha), Bubble sort também trata-se de um algoritmo de ordenação simples que tem como fundamento comparar o valor atual do vetor com o valor posterior. Caso o posterior seja menor, a troca é feita entre ambosos índices. Esse processo é repetido até que o vetor esteja completamente ordenado.

Sua vantagem é ser um algoritmo simples e fácil de se implementar, porém sua complexidade no pior caso é ruim, o que o torna um algoritmo muito lento para vetores de tamanho grande.

Sua complexidade temporal: O(n^2), no melhor caso é (n), no caso médio é (n^2) e no pior caso é (n^2), sendo assim quadrática.

Sua complexidade espacial geralmente é O(1), porém no nosso exemplo é O(n), pois é criado um vetor e não uma variável. Sendo portanto linear.

```
5  void bubbleSort(int TAM, int *vetor){
6
7      int i, segue, f = TAM, aux;
8
9      do{
10
11         segue = 0;
12
13         for(i=0; i<f-1; i++){
14             if(vetor[i] > vetor[i+1]){
15                 aux = vetor[i];
16                 vetor[i] = vetor[i+1];
17                 vetor[i+1] = aux;
18                 segue = i;
19             }
20         }
21
22         f--;
23
24     }while(segue != 0);
25
26 }
```

Mais uma vez, temos um aqui um código de implementação simples. São criadas duasvariáveis de controle **segue** e **f**. Segue é responsável por terminar a execução do **do while**. Caso o vetor esteja ordenado, **segue** não receberá o valor de **i** e a função será finalizada. Quanto a variável **f**, a qual se responsabiliza pelo termino do **for** interno. Como já citado, a ideia aqui permutar o valor atual do vetor como próximo, isso caso o próximo seja menor. A permutação é feita por meio do **aux**. A cada rodada do **while**, **f** é decrementado, o que faz o **for** ter menos execuções.

O vetor é ordenado de forma crescente de acordocom o **valor** dos produtos.

Algoritmo*: Selection Sort*

Selection sort ou ordenação por seleção, trata-se um algoritmo simples que baseia-se em buscar sempre o menor valor contido no vetor e passá-lo para a primeira posição. Logo depois é descarta tal posição e o novo menor elemento é alocado na nova primeira posição do vetor. Esse processo é repetido até que se chegue até o último índice do vetor.

Seus prós são: é um algoritmo simples e fácil de se implementar e trata-se de um dos algoritmos mais velozes na ordenação de vetores de pequenos tamanhos.

Seus contras são: sua complexidade no pior caso é ruim, o que o torna um algoritmo muito lento para vetores de tamanho grande; não é um algoritmo estável e ocorrem sempre comparações independente do vetor já estar ordenado ou não.

Sua complexidade temporal: $O(n^2)$, no melhor caso é (n^2), no caso médio é (n^2) e no pior caso é (n^2), sendo assim quadrática.

Sua complexidade espacial geralmente é O(1), porém no nosso exemplo é O(n), pois é criado um vetor e não uma variável. Sendo portanto linear.

Temos aqui uma implementação bem simples. Há um **for** externo que percorre todo o vetor e um **for** interno que procura o menor elemento do vetor
(aqui trata-se do índice do vetor que possui o menor **valor** de produto). Já o **if** é responsável pela troca entreo produto atual do vetor o respectivo produto armazenado em **aux**.

```
120 void selectionSort(int TAM, int *vetor){
121
122     int i, j, min, aux;
123
124     for(i=0; i<TAM-1; i++){
125
126         min = i;
127
128         for(j=i+1; j<TAM; j++){
129             if(vetor[j] < vetor[min]){
130                 min = j;
131             }
132         }
133
134         if(i != min){
135             aux = vetor[i];
136             vetor[i] = vetor[min];
137             vetor[min] = aux;
138         }
139
140     }
141
142 }
143
```

A variável **min** recebe a cada rodada do **for** o valor de **i**, o que impede que o **for** interno percorra um índice já ordenado do vetor. Posteriormente **min** recebe**j**, o que permite comparar **i** a **min**. Caso **i** seja diferentede **min,** é necessário realocar o produto em questão.
Caso contrário não há necessidade de realocar.

É interessante notar o que o **for** externo percorreo vetor apenas até o índice anterior ao último. Isso
ocorre pois quando se chega no penúltimo índice, o último já deve naturalmente ser o maior **valor**.

Questão 4: Merge Sort

Criado por Von Neumann em 1945, Merge sort, também conhecido como ordenação por mistura é um algoritmo bem mais sofisticado que os citados anteriormente. Aqui é utilizado princípio de dividir para conquistar.

O conceito aqui é subdividir o programa e ordenar as pequenas parcelas divididas para posteriormente reagrupá-las.

Suas vantagens são: complexidade temporal extremamente eficiente e veloz.

Suas desvantagens são: depende de um alto consumo de memória por conta de sua natureza recursiva e a criação de um vetor.

```
void mergeSort(int TAM, int *vetor, int ini, int fim){

    int mei;

    if(ini<fim){
        mei = ((ini+fim)/2);
        mergeSort(TAM, vetor, ini, mei);
        mergeSort(TAM, vetor, mei+1, fim);
        mergeSortaux(TAM, vetor, ini, mei, fim);
    }

}

void mergeSortaux(int TAM, int *vetor, int ini, int mei, int fim){

    int p1, p2, tam, i, j, k, fim1 = 0, fim2 = 0;

    int *aux = (int*) malloc(sizeof(int)*tam-1);

    tam = (fim - ini) + 1;
    p1 = ini;
    p2 = mei + 1;

    if(aux != NULL){

        for(i=0; i<tam; i++){
            if(!fim1 && !fim2){
                if(vetor[p1] < vetor[p2]){
                    aux[i] = vetor[p1++];
                }
                else{
                    aux[i] = vetor[p2++];
                }

                if(p1 > mei){
                    fim1 = 1;
                }
                if(p2 > fim){
                    fim2 = 1;
                }
            }
            else{
                if(!fim1){
                    aux[i] = vetor[p1++];
                }
                else{
                    aux[i] = vetor[p2++];
                }
            }
        }

    }
```

Sua complexidade temporal: O(n log n), no melhor caso é (n log n), no caso médio é (n logn) e no pior caso é (n log n), sendo assim logarítmica linear.

Sua complexidade espacial geralmente é O(n), porém no nosso exemplo é O(n²), pois é criado um ponteiro de ponteiro e não um vetor. Sendo portanto quadrático.

*Inicialmente é criada a variável **mei** que recebe exatamente o centro do vetor. Ademais, a função **mergeSort** é chamada recursivamente duas vezes com o intuito de dividir o vetor em partes. Após a etapa recursiva de **mergeSort**, **mergeSortaux** é chamada com o dever de combinaras duas metades de forma ordenada. Dentro da função **mergeSortaux**, as variáveis **p1** e **p2** recebem os valores iniciais correspondentes a cada um dos vetores divididos em **mergeSort**.Além disso é criado um ponteiro para ponteiro **aux** o qual tem como função comparar os valores dos **produtos** em ambos os vetores subdivididos.*

Dentro do for, o primeiro if e else é chamado caso o vetor não estiver completamente varrido, ou seja f1 e f2 não sejam maiores que mei e fim respectivamente. Dentro desse if e else é feita a comparação dos valores dos produtos nos índices p1 e p2. Caso no índice p1 o valor do produto seja menor que no índice p2, aux receberá no índice i o vetor de dados no índice p1+1. Caso contrário aux receberá p2+1.

Caso f1 ou f2 forem iguais a 1, um else é rodado. Caso apenas f2 seja igual a 1, aux receberá no índice o vetor de dados no índice p1+1.Caso contrário aux receberá p2+1.

Ao fim de tais operações, o vetor de dados é rearranjado de forma a ordená-lo até que o índice tam seja chamado. Tam sempre recebe o valor de fim-ini+1. Por meio de um for o vetor de dados recebe aux.

Por fim, aux é liberado.

O que acontece na prática é que a função mergeSort quebra o vetor de forma binária até que restem apenas unidades individuais. Enquanto a mergeSortaux age em reagrupar cada elemento individual de forma ordenada.

Algoritmo: *Quick Sort*

Em 1960, Charles Antony Richard Hoare visitou a Universidade de Moscovo na época de estudante. Naquela época, Hoare trabalhava em um projeto de tradução de máquina para o National Physhical Laboratory. Ao tentar traduzir um dicionário de inglês para russo, ordenandoas palavras, seu objetivo era reduzir o problema original em subproblemas que pudessem ser resolvidos de forma mais fácil e rápida. Foi então que em 1962, após uma série de refinamentos, Hoare publicou o Quick Sort.

Também conhecido como ordenação por partição, o Quick sort segue a ideia de dividir para conquistar. Sua estratégia consiste em rearranjar as chaves de modo que as chaves menores precedam as chaves maiores. Em seguida há a ordenação dos sub vetores de chaves menores e maiores de recursiva até que o vetor seja completamente ordenado.

Suas vantagens são: Extremamente veloz e eficiente para grandes entradas. Suas desvantagens são: trata-se de um algoritmo de comparação não estável.

Sua complexidade temporal: O(n^2), no melhor caso é (n log n), no caso médio é (n log n) eno pior caso é (n^2), sendo assim logarítmica linear na maioria das vezes, porém, quadrático em casos raros.

Sua complexidade espacial geralmente é O(log n), Sendo portanto logarítmica.

```c
void quickSort(int TAM, int *vetor, int ini, int fim){

    int i, j, pivo, aux;

    i = ini;
    j = fim-1;
    pivo = vetor[(ini + fim)/2];

    while(i <= j){

        while((vetor[i] < pivo) && (i < fim)){
            i++;
        }
        while((vetor[j] > pivo) && (j > ini)){
            j--;
        }

        if(i <= j){
            aux = vetor[i];
            vetor[i] = vetor[j];
            vetor[j] = aux;
            i++;
            j--;
        }
    }

    if(j > ini){
        quickSort(TAM, vetor, ini, j+1);
    }
    if(i < fim){
        quickSort(TAM, vetor, i, fim);
    }

}
```

Os vetor **pivo** recebe o vetor de dados no índice central do vetor. Enquanto **aux** é responsável pela troca no vetor de dados, **pivo** é responsável pela comparação do valor do **produto** central com o valor atual do vetor de dados. Ambas essas operações são realizadas dentro de **while** externo que rodará enquanto o valor de **i** não chegar no valor de **j**.

Se no fim desse while externo j for maior que ini, quickSort é chamada novamente de forma recursiva com os parâmetros: o vetor de dados, ini e j+1. Caso i seja menor que fim, quickSort será chamada com os parâmetros: o vetor de dados, i e fim.

Basicamente aqui são seguidos 3 passos principais:

Primeiro: o pivo recebe o vetor de dados na posição central dos índices.

Segundo: rearranja a lista de forma que todos os elementos anteriores ao pivo sejam menores que o mesmo. Ao fim desse processo o pivo estará em sua posição final e haverá dois sub vetores não ordenados. Esse processo chama-se partição.

Terceiro: ordena-se os sub vetores de tamanho maior e menor ao pivo. Essa é a parte recursiva do algoritmo.

Coclusão sobre os algoritmos de ordenação:

Analisando ambos os algoritmos tratados, temos a seguinte ordem crescente de eficiência:

***Bubble Sort** é o algoritmo menos eficiente visto aqui. Apesar de ser um algoritmo simples de ser implementado, sua complexidade (n^2) tanto em casos médios quanto no pior caso o torna deveras ineficiente e lento quando tratando-se de grandes entradas.*

***Selection Sort** é um algoritmo um pouco melhor que o Bubble Sort. Possui facilidade de ser implementado e possui grande velocidade quando trata-se de entradas pequenas, entretanto sua complexidade de (n^2) em qualquer que seja o caso faz dele um péssimo algoritmo quando trata-se de vetores grandes. Além de não ser um algoritmo estável.*

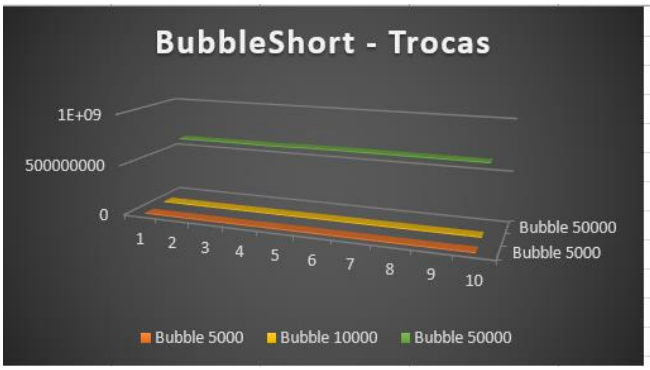
***Insertion Sort**, como já citado, é o melhor algoritmo de ordenação quadrática. Possui uma implementação simples e é estável. Todavia, sua complexidade de (n^2) em casos médios e em piores casos o torna um algoritmo lentíssimo para grandes entradas de dados.*

***Merge Sort** perde apenas para Quick Sort em eficiência. Sua complexidade temporal (n log n) independente do caso, o torna um algoritmo bastante eficiente para entradas grandes de dados.*

***Quick Sort** é o melhor algoritmo de ordenação. Apesar de no seu pior caso possuir complexidade temporal de (n^2), tal caso é muito raro. Na maioria das vezes, sua complexidade temporal é (n log n) e a complexidade espacial é (log n). Trata-se do algoritmo mais eficiente e veloz conhecido.*

Analise de execuções e metrícas de medidas

Buble Short:

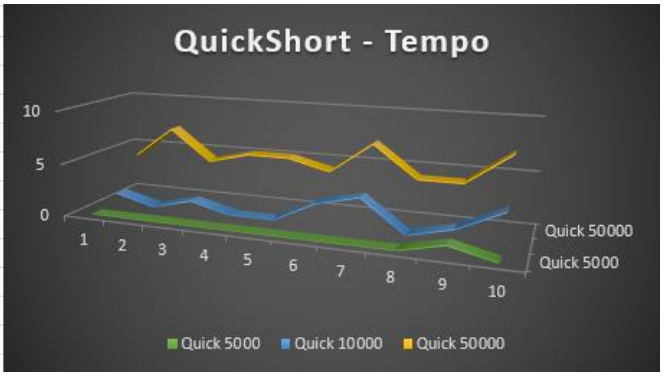


Analise Bubble Short na íntegra:

Bubble Short											
VETOR 5000	1	2	3	4	5	6	7	8	9	10	**TOTAL**
Troca	6256798	6135878	6245224	6192620	6262118	6284883	6314470	6228040	6229350	6310596	6245997,7
Tempo	48	59	51	70	68	68	69	68	70	69	64
VETOR 10000	1	2	3	4	5	6	7	8	9	10	**TOTAL**
Troca	25106742	24832324	24653118	25211903	24898099	25358346	24927127	24782937	24961298	24797624	24952951,8
Tempo	291	283	283	283	283	282	284	282	287	285	284,3
VETOR 50000	1	2	3	4	5	6	7	8	9	10	**TOTAL**
Troca	619425050	620165600	620165600	617212016	625345735	627115951	625163707	624034902	624489854	622082479	622520089,4
Tempo	7452	7155	7199	7187	7128	7187	7150	7168	7155	7165	7194,6

Estes foram os resultados obtidos durante as execuções de cada vetor, vale salientar que cada vetor de tamanho N, (previamente estipulado), deveria ser executado três vezes para que fosse admissível a coleta de dados. Logo após foi retirado uma média de cada parâmetro e finalizado a análise com a demonstração em gráficos, pra melhor visualização.

Quick Short:



Analise Quick Short na íntegra:

Quick Short											
VETOR 5000	1	2	3	4	5	6	7	8	9	10	**TOTAL**
Troca	8389711	7210056	1468	6751148	1351	10360865	8062063	1402	7603138	1147	4838234,9
Tempo	0	0	0	0	0	0	0	0	1	0	0,1
VETOR 10000	1	2	3	4	5	6	7	8	9	10	**TOTAL**
Troca	2741	2549	10628752	11078326	6762175	2782	10422645	8391055	2496	7145864	5443938,5
Tempo	1	0	1	0	0	2	3	0	1	3	1,1
VETOR 50000	1	2	3	4	5	6	7	8	9	10	**TOTAL**
Troca	17023	14137	11224305	13999	10372495	8268917	16869	13769	14206	12803	2996852,3
Tempo	4	7	4	5	5	4	7	4	4	7	5,1

Estes foram os resultados obtidos durante as execuções de cada vetor, vale salientar que cada vetor de tamanho N, (previamente estipulado), deveria ser executado três vezes para que fosse admissível a coleta de dados. Logo após foi retirado uma média de cada parâmetro e finalizado a análise com a demonstração em gráficos, pra melhor visualização.

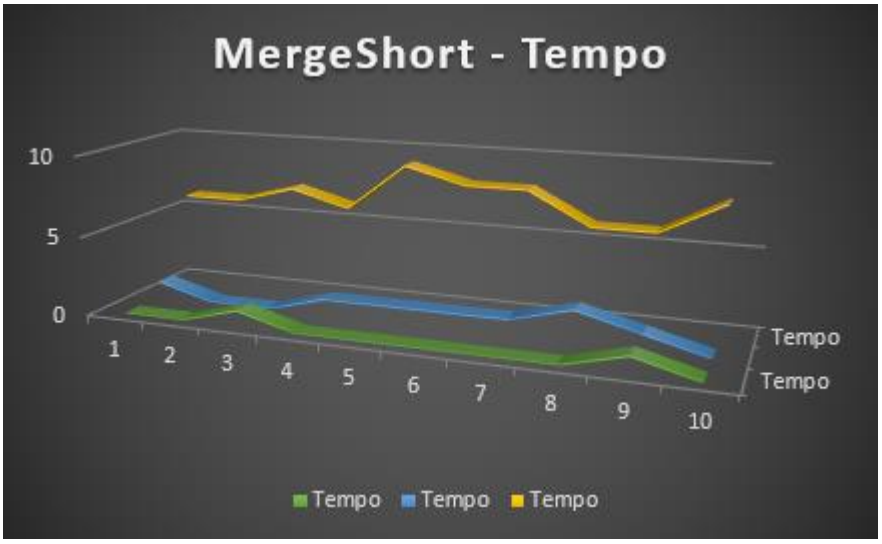
Selection Short:

Analise Selection Short na íntegra:

Selection Short											
VETOR 5000	1	2	3	4	5	6	7	8	9	10	**TOTAL**
Troca	4992	4987	4988	4986	4988	4988	4989	4984	4991	4988	4988,1
Tempo	23	25	25	24	24	25	26	25	24	25	24,6
VETOR 10000	1	2	3	4	5	6	7	8	9	10	**TOTAL**
Troca	9984	9988	9987	9983	9983	9987	9985	9979	9977	9984	9983,7
Tempo	102	108	103	101	99	101	102	102	102	102	102,2
VETOR 50000	1	2	3	4	5	6	7	8	9	10	**TOTAL**
Troca	49941	49948	49948	49935	49945	49936	49934	49942	49945	49945	49941,9
Tempo	2594	2586	2608	2579	2587	2572	2568	2582	2605	2576	2585,7

Estes foram os resultados obtidos durante as execuções de cada vetor, vale salientar que cada vetor de tamanho N, (previamente estipulado), deveria ser executado três vezes para que fosse admissível a coleta de dados. Logo após foi retirado uma média de cada parâmetro e finalizado a análise com a demonstração em gráficos, pra melhor visualização.

Merge Short:



Como solicitado, a análise do Algoritmo Merge Shot se deu apenas para o escopo temporal, sem a necessidade de apurar seus dados de troca.

Analise Selection Short na íntegra:

Merge Short											
VETOR 5000	1	2	3	4	5	6	7	8	9	10	TOTAL
Tempo	0	0	1	0	0	0	0	0	1	0	0,2
VETOR 10000	1	2	3	4	5	6	7	8	9	10	TOTAL
Tempo	1	0	0	1	1	1	1	2	1	0	0,8
VETOR 50000	1	2	3	4	5	6	7	8	9	10	TOTAL
Tempo	6	6	7	6	9	8	8	6	6	8	7

Estes foram os resultados obtidos durante as execuções de cada vetor, vale salientar que cada vetor de tamanho N, (previamente estipulado), deveria ser executado três vezes para que fosse admissível a coleta de dados. Logo após foi retirado uma média de cada parâmetro e finalizado a análise com a demonstração em gráficos, pra melhor visualização.

Conclusão

Temos em vista que os algoritmos tem tempos de resposta e eficácia diferente em cada um dos casos. Vemos que o melhor algoritmo é o Merge Short e o Quick short por causa do seus tempos totais em relação aos demais algoritmos, já que no pior caso ele teve a média de tempo de 7ms no Merge Short e o Quick Short no seu pior caso fez 5.1ms. Se mostrando os algoritmos mais performáticos em relação ao tempo.

Já os piores representados neste experimento foram o Bubble Short e o Selection Short, em respectivas ordens dos resultados dos seus piores casos, temos a média de 7194.6ms e 2585.7ms. Isto representa uma disparidade elevada entre esses dois algoritmos, mesmo o Selection Short mais performático, ainda não alcanlça os níveis de performance dos Merge e Quick Short.

Para fins de conhecimento, o código usado para o experimento está presente abaixo:

----------Main.cpp----------

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdint.h>
#include "Sorting_Algorithms.h"
#define TAM 50000

int* gerador(int tam){

	int *vetor = (int *)malloc(sizeof(int)*tam);

	if (vetor != NULL) {
		srand(time(NULL));
		for (int i = 0; i < tam; i++){
			vetor[i] = rand() % 1000;
		}
		return vetor;
	}
	else {
		printf("Problema de memoria");
		exit(1);
	}

}

int main(){

	int *vetor = gerador(TAM);

	clock_t Ticks[2];
	Ticks[0] = clock();

	printf("Vetor: \n[");
	for(int i=0; i<TAM; i++){
	 printf("%d ", vetor[i]);
    }
    printf("]");

	bubbleSort(TAM, vetor);
	quickSort(TAM, vetor, 0, TAM);
	mergeSort(TAM, vetor);
	selectionSort(TAM, vetor);


	printf("\n\nVetor Ordenado: \n[");
	for(int i=0; i<TAM; i++){
	 printf("%d ", vetor[i]);
    }
    printf("]");

	printf("\n\n");
	Ticks[1] = clock();
	double Tempo = (Ticks[1] - Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
	printf("\n  Tempo gasto: %g ms.\n", Tempo);

	return 0;

}
```

----------Shorting_Algorithms.h----------

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

void bubbleSort(int TAM, int *vetor);

void quickSort(int TAM, int *vetor, int ini, int fim);

void mergeSort(int TAM, int *vetor);
void sort(int *vetor, int *vetorAux, int i, int f);
void merge(int *vetor, int *vetorAux, int i, int m, int f);
void selectionSort(int TAM, int *vetor);
```

---------- Shorting_Algorithms.cpp ----------

```
#include <stdio.h>
#include <stdlib.h>
#include "Sorting_Algorithms.h"

void bubbleSort(int TAM, int *vetor){

	int i, segue, f = TAM, aux, troca = 0;

	do{

		segue = 0;

		for(i=0; i<f-1; i++){
			if(vetor[i] > vetor[i+1]){
				aux = vetor[i];
				vetor[i] = vetor[i+1];
				vetor[i+1] = aux;
				segue = i;
				troca++;
			}
		}

		f--;

	}while(segue != 0);
	printf("\nTrocas efetuadas: %d", troca);
}


void quickSort(int TAM, int *vetor, int ini, int fim){

	int i, j, pivo, aux, troca = 0, trocaAux;

	i = ini;
	j = fim-1;
	pivo = vetor[(ini + fim)/2];
	if((j == TAM - 1)&&(i == 0)){
		trocaAux = 0;
	}

	while(i <= j){

		while((vetor[i] < pivo) && (i < fim)){
			i++;
		}
		while((vetor[j] > pivo) && (j > ini)){
			j--;
		}

		if(i <= j){
			aux = vetor[i];
			vetor[i] = vetor[j];
			vetor[j] = aux;
			troca++;
			i++;
			j--;
		}
	}

	trocaAux += troca;

	if(j > ini){
		quickSort(TAM, vetor, ini, j+1);
	}
```

```
        if(i < fim){
                quickSort(TAM, vetor, i, fim);
        }
        if(i==TAM){
                printf("\nTrocas efetuadas: %d", trocaAux);
        }

}

void mergeSort(int TAM, int *vetor){
  int *vetorAux = (int *)malloc(sizeof(int)*TAM);
  sort(vetor, vetorAux, 0, TAM-1);
  free(vetorAux);
}

void sort(int *vetor, int *vetorAux, int i, int f){
  if (i >= f) return;

  int m = (i + f) / 2;

  sort(vetor, vetorAux, i, m);
  sort(vetor, vetorAux, m + 1, f);

  if (vetor[m] <= vetor[m + 1]) return;

  merge(vetor, vetorAux, i, m, f);
}

void merge(int *vetor, int *vetorAux, int i, int m, int f) {
  int z, ivetor = i, ivetorAux = m + 1;

  for (z = i; z <= f; z++) vetorAux[z] = vetor[z];

  z = i;

  while (ivetor <= m && ivetorAux <= f) {

    if (vetorAux[ivetor] <= vetorAux[ivetorAux]) vetor[z++] = vetorAux[ivetor++];
    else vetor[z++] = vetorAux[ivetorAux++];
  }

  while (ivetor <= m) vetor[z++] = vetorAux[ivetor++];

  while (ivetorAux <= f) vetor[z++] = vetorAux[ivetorAux++];
}

void selectionSort(int TAM, int *vetor){

        int i, j, min, aux, troca = 0;

        for(i=0; i<TAM-1; i++){

                min = i;

                for(j=i+1; j<TAM; j++){
                        if(vetor[j] < vetor[min]){
                                min = j;
                        }
                }

                if(i != min){
                        aux = vetor[i];
                        vetor[i] = vetor[min];
                        vetor[min] = aux;
                        troca++;
                }
```

```
        }
        printf("\nTrocas efetuadas: %d", troca);

}
```