



# CCF 211 - Algoritmos e Estruturas de Dados I Trabalho Prático 03 - Ordenação

Vinícius Mendes - 3881, João Roberto - 3883, Artur Papa - 3886

11 de maio de 2021

## 1 Introdução

O trabalho em questão tem como objetivo abordar os conteúdos trabalhados na disciplina de Algoritmos e Estrutura de Dados I, em específico, como visto nas aulas teóricas e práticas, os diferentes métodos de ordenação, sejam eles simples ou sofisticados. A priori, as técnicas de ordenação permitem apresentar um conjunto amplo de algoritmos distintos para resolver uma mesma tarefa, sendo que alguns métodos podem ser melhores que outros dependendo da aplicação [3].

Além disso, cabe ressaltar que existem dois aspectos diferentes de organização, sendo a ordenação interna e externa. Nota-se que a primeira acontece quando todos os elementos a serem ordenados cabem na memória principal qualquer registro pode ser imediatamente acessado. Outrossim, é válido dizer que a segunda acontece quando todos os elementos a serem ordenados não cabem na memória principal e os registros são acessados sequencialmente ou em grandes blocos. [2]

## 2 Métodos de Ordenação

### 2.1 Bubble Sort

A implementação desse método de ordenação tem como ideia principal percorrer o vetor/lista diversas vezes, buscando passar o maior valor para a posição final da cadeia de caracteres.

```
void bubbleSort(long long int *arr, int n, int flag, int *comparacoes, int *movimentacoes, double* tempo, int cenario, int *exec1)
{
    inicio = clock();
    int i, j;
    int bubble_comp = 0;
    int bubble_swap = 0;
    for (i = 0; i < n - 1; i++)
    {
        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
        {
            bubble_comp++;
            if (arr[j] > arr[j + 1])
            {
                swap(&arr[j], &arr[j + 1]);
                bubble_swap++;
            }
        }
    }
}
```

Figura 1: Ordenação por Bolha

O Método Bolha pode ser considerado uma das piores formas de se ordenar um array, visto que possui complexidade de tempo  $O(n^2)$  em seu pior caso, e no melhor caso  $O(n)$ . Fato este

que, até o ex-presidente dos Estados Unidos, Barack Obama, tem conhecimento(dito em sua participação na "Business of Software 2013: A/B Testing - If data can help win elections, what can it help you do for your business?").

Na computação, sua utilização comumente se dá com o objetivo de identificar um pequeno erro, logo, em grandes quantidades de dados torna-se ineficiente.

Em nossa implementação, utilizamos uma função chamada "swap"que é responsável por realizar a troca de posição dos itens.

```
void swap(long long int *xp, long long int *yp)
{
    long long int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
```

Figura 2: Ordenação por Bolha.

## 2.2 Selection Sort

Por sua vez, o Selection Sort tem como objetivo o contrário do bubble, passar o menor valor para o início da cadeia de caracteres. Tem custo linear no tamanho da entrada para o número de movimentos.

O método de ordenação por Seleção tem uma característica importante, seu melhor e pior caso possuem a mesma complexidade de tempo, visto que o algoritmo realizará sempre todas as comparações possíveis , logo  $O(n^2)$ , além de que o mesmo é considerado um algoritmo não estável.

```
// One by one move boundary of unsorted subarray
for (i = 0; i < n - 1; i++)
{
    // Find the minimum element in unsorted array
    min_index = i;
    for (j = i + 1; j < n; j++)
    {
        selection_comp++;
        if (arr[j] < arr[min_index])
        {
            min_index = j;
            selection_swap++;
        }
    }
    // Swap the found minimum element with the first element
    swap(&arr[min_index], &arr[i]);
    selection_swap++;
}
```

Figura 3: Ordenação por Seleção.

## 2.3 Insertion Sort

O método do Insertion Sort é similar ao que as pessoas usam para ordenar um baralho. A lista é dividida em duas sub listas, uma ordenada e outra não ordenada, conforme a lista é percorrida cada elemento é transferido para a sub lista de elementos não ordenados, e através de uma série de comparações, ele já é inserido na posição correta.

Em termos de complexidade, o melhor caso para o Insertion Sort ocorre quando a lista já está ordenada, nesse caso a complexidade é de  $O(n)$ , aonde  $n$  é o tamanho da lista. O pior caso é quando a lista está em ordem decrescente (ou crescente dependendo da ordem em que se deseja ordenar a lista), nesse caso, e no caso médio também, a complexidade é de  $O(n^2)$ .

Esse método de ordenação é estável e in-place.

```
for (i = 1; i < n; i++)
{
    j = i;
    insertion_comp++;
    while ((j > 0) && (vet[j - 1].registros.chave > vet[j].registros.chave))
    {
        if (vet[j - 1].registros.chave > vet[j].registros.chave)
        {
            insertion_comp++;
        }
        temp = vet[j - 1].registros.chave;
        vet[j - 1].registros.chave = vet[j].registros.chave;
        vet[j].registros.chave = temp;
        j--;
        insertion_swap++;
    }
}
```

Figura 4: Ordenação por Inserção.

## 2.4 Shell Sort

O Shell Sort é uma versão otimizada do Insertion Sort, sua proposta é separar a lista principal em listas menores, com intervalos na comparação que, decrescerão até chegar ao intervalo 1.

A análise de complexidade do Shell Sort contém problemas matemáticos muito difíceis e ninguém ainda foi capaz de analisa-lo por completo.

O Shell Sort é um método que não é estável e é in-place.

```
for (gap = n / 2; gap > 0; gap /= 2)
{
    for (i = gap; i < n; i++)
    {
        temp = vet[i].registros.chave;
        for (j = i; j >= gap && vet[j - gap].registros.chave > temp; j -= gap)
        {
            vet[j].registros.chave = vet[j - gap].registros.chave;
            shell_comp++;
            shell_swap++;
        }
        vet[j].registros.chave = temp;
        shell_comp++;
    }
}
```

Figura 5: Shell Sort.

## 2.5 Quick Sort

A proposta do Quick Sort é escolher um pivô na lista, e então dividir a lista em duas sublistas uma com elementos maiores e outras com elementos menores que o vetor. Esse processo é repetido diversas vezes até a lista estar ordenada.

A complexidade do Quick Sort também depende da lógica por trás da escolha do pivô, No pior caso, quando o pivô escolhido for o menor ou o maior elemento da lista várias vezes a complexidade vai ser de  $O(n^2)$ , no entanto o pior caso pode ser evitado fazendo com alterações na implementação. O melhor caso para este método tem complexidade de  $O(n \log n)$  e ele ocorre quando os pivôs escolhidos são os elementos do meio.

A implementação default do Quick Sort não é estável, e ele é in-place.

```
particao2(left, right, &i, &j, vet, comparacoes, movimentacoes, mov, comp, cenario);
comp++;
if (left < j)
    ordena2(left, j, vet, comparacoes, movimentacoes, mov, comp, cenario);
comp++;
if (i < right)
    ordena2(i, right, vet, comparacoes, movimentacoes, mov, comp, cenario);
```

Figura 6: Função Ordena.

```
int pivot = vet[(i + j) / 2].registros.chave; /* obtém o pivo x */
int aux;
do
{
    comp++;
    while (pivot > vet[*i].registros.chave)
    {
        (*i)++;
        comp++;
    }

    while (pivot < vet[*j].registros.chave)
    {
        (*j)--;
        comp++;
    }
    comp++;
    if (*i <= *j)
    {
        mov++;
        aux = vet[*i].registros.chave;
        vet[*i].registros.chave = vet[*j].registros.chave;
        vet[*j].registros.chave = aux;
        (*i)++;
        (*j)--;
    }
} while (*i <= *j);
```

Figura 7: Função Particiona.

## 2.6 Merge Sort

O Merge Sort funciona dividindo a lista ao meio diversas vezes e depois juntando essas sublistas de maneira ordenada.

A complexidade do Merge Sort sempre será  $O(n \log n)$ , então não faz sentido em discutir melhores e piores casos.

Esse método é estável e não é in-place, pois ele precisa de memória adicional para armazenar as sub listas temporárias.

```
if (l < r)
{
    int m = l + (r - l) / 2;
    merge_Sort(vet, l, m, 0, comparacoes, movimentacoes, tempo, cenario, exec1);
    merge_Sort(vet, m + 1, r, 0, comparacoes, movimentacoes, tempo, cenario, exec1);
    merge2(vet, l, m, r, comp, mov, 0, comparacoes, movimentacoes, cenario, exec1);
}
```

Figura 8: Merge Sort.

## 2.7 Counting Sort

O Counting Sort é um método de ordenação que diferente dos outros que vimos até agora, não usa comparações, ao invés disso ele conta quantas vezes cada número aparece na lista, e através de algumas manipulações matemáticas nesses contadores ele determina a posição aonde cada item tem que assumir para que a lista seja ordenada.

O Counting Sort não pode ser usado para ordenar listas com números de ponto flutuante por causa de sua implementação, outra observação importante sobre este método é que ele funciona melhor quando o intervalo entre o menor elemento e o maior elemento já é conhecido antes da ordenação. O Counting Sort é frequentemente usado como um subprograma para outros métodos de ordenação como por exemplo o Radix Sort.

A complexidade do Counting Sort é  $O(n + k)$ , aonde  $n$  é o tamanho da lista e  $k$  é o intervalo entre os elementos da lista.

A versão deste método que implementamos no TP, é estável, e não é in-place pois vai precisar de memória adicional para guardar a lista temporária de contadores, no entanto há implementações desse método que não precisam de memória extra, no entanto, essas implementações não são estáveis.

```
for (i = 0; i < n; i++)
    count[((vet[i].registros.chave) / exp) % 10]++;

for (i = 1; i < 10; i++)
    count[i] += count[i - 1];

for (i = n - 1; i >= 0; i--)
{
    output[count[((vet[i].registros.chave) / exp) % 10] - 1] = (vet[i].registros.chave);
    count[((vet[i].registros.chave) / exp) % 10]--;
}

for (i = 0; i < n; i++)
{
    if ((vet[i].registros.chave) != output[i])
    {
        count_swap++;
    }
}

for (i = 0; i < n; i++)
    (vet[i].registros.chave) = output[i];
```

Figura 9: Counting Sort.

## 2.8 Radix Sort

O Counting Sort é um método de ordenação cuja a estratégia baseia-se em usar o valor de cada algarismo do elemento a ser ordenado para ordenar a lista, usando o Counting Sort. Ou seja, dada uma lista, o algoritmo, através do Counting Sort, irá usar o valor do algarismo na casa das unidades como chave de ordenação, após isso irá usar o valor do algarismo na casa das dezenas, e depois das centenas e assim por diante, ao final desse processo a lista estará ordenada. É importante mencionar que a versão do Counting Sort implementada deve ser estável para que o Radix Sort funcione corretamente.

A complexidade do Radix Sort é  $O(d(n + b))$  aonde é  $n$  o tamanho do vetor,  $b$  é a base dos números utilizados na lista e  $d$  é a quantidade de números que podem ser usados para representa-los.

Esse método é estável, e não é in-place pois ele usa o Counting Sort, método esse que não é in-site, como um subprograma.

```
long long int m = getMax2(vet, n, radix_comp);
for (int exp = 1; m / exp > 0; exp *= 10)
{
    count_Sort(vet, n, exp, 1);
    radix_swap = radix_swap + count_swap;
}
```

Figura 10: Radix Sort.

## 3 Gráficos

Após a execução dos algoritmos de ordenação, tivemos que fazer a coleta dos dados para a criação dos gráficos de médias das comparações, movimentações e do tempo de cada método. A priori, nota-se que para gerar os arquivos usamos a função *arquivoX* (nome baseado na série dos anos 90), responsável por coletar os valores, organizá-los para cada cenário e para cada algoritmo de ordenação.

Além disso, cabe dizer que os dados foram utilizados no *excel* para gerar os gráficos, sendo que esta tabela esta localizada na pasta *doc*.

```
void arquivoX(long long int *comparacoes, long long int *movimentacoes, double* tempo, int valor)
{
    FILE *pFile;
    switch (valor)
    {
        case 1:
            pFile = fopen("../out/BubbleSort_Cenario1.txt", "w+");
            fprintf(pFile, "Bubble Sort -> Cenario 1\n");
            fprintf(pFile, "Media das comparacoes: %lld\nMedia das movimentacoes: %lld\nMedia do tempo: %lf segundos\n",
            fclose(pFile);
            break;
    }
```

Figura 11: Função "ArquivoX".

### 3.1 Cenário 1

Em primeiro lugar, serão mostrados os gráficos para o primeiro cenário, tendo em conta que eles foram feitos baseados nos métodos de ordenação e tamanho da entrada.

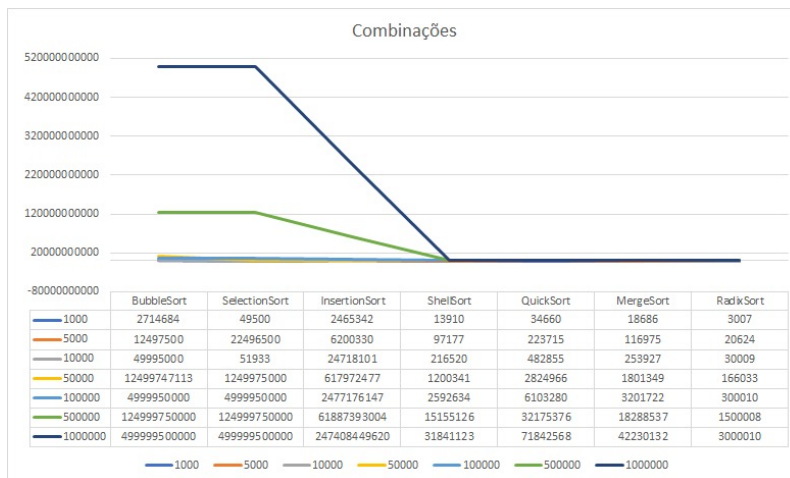


Figura 12: Gráfico das médias de Combinações.

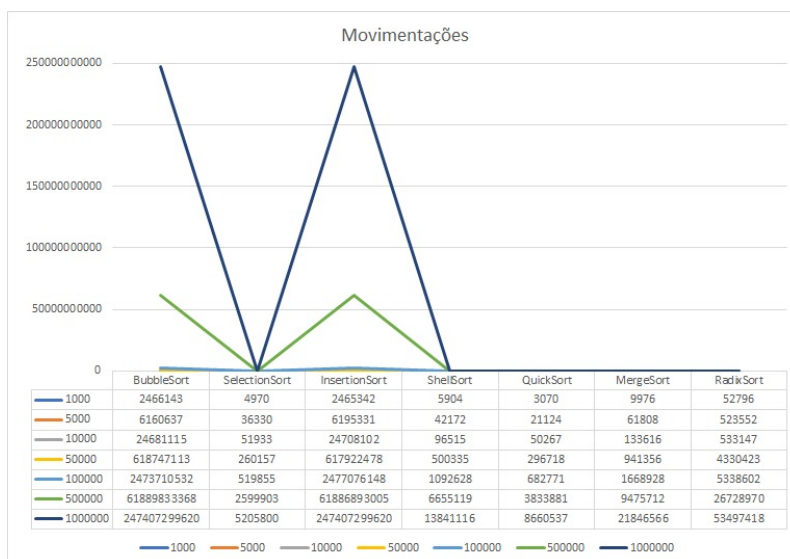


Figura 13: Gráfico das médias de Movimentações.

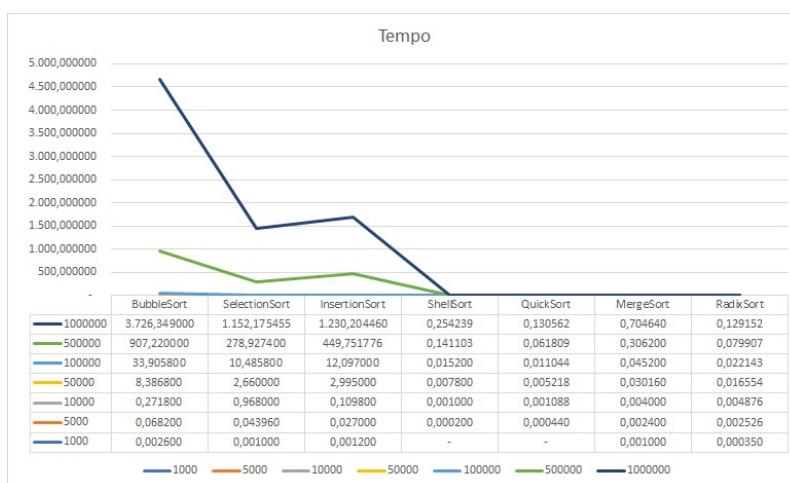


Figura 14: Gráfico das médias de Tempo.



## 3.2 Cenário 2

Em segundo lugar, serão mostrados os gráficos para o segundo cenário, seguindo a mesma ordem e requisitos mostrados no primeiro caso.

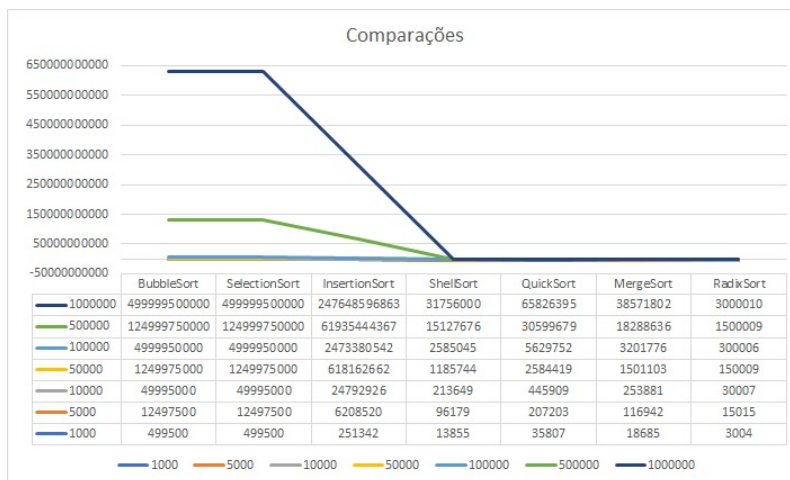


Figura 15: Gráfico das médias de Combinações.

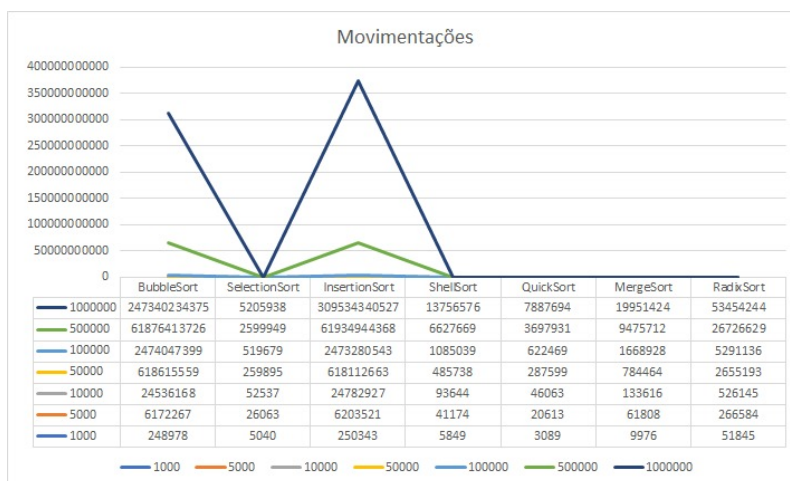


Figura 16: Gráfico das médias de Movimentações.

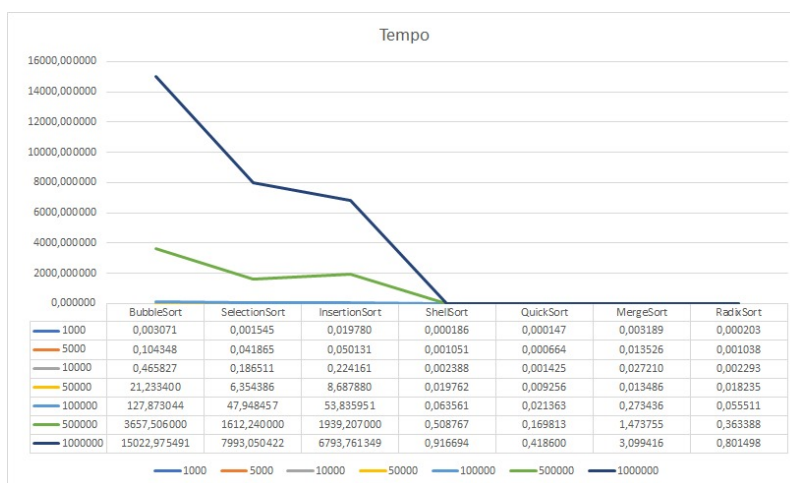


Figura 17: Gráfico das médias de Tempo.

Desta maneira, observa-se nos gráficos que algumas linhas estão sobrepostas, isto se deve ao fato da diferença de valores na escala para o *eixo Y*, já que são valores muito grandes, por este

motivo há a impressão de algumas curvas estarem tocando o *eixo X*.

## 4 Estrutura do Projeto

Com o intuito de manter o projeto organizado, decidimos estruturar o trabalho em pastas diferentes, dessa forma seria mais claro para entender o código e não confundir no momento da execução e ordenação dos inteiros ou registros.

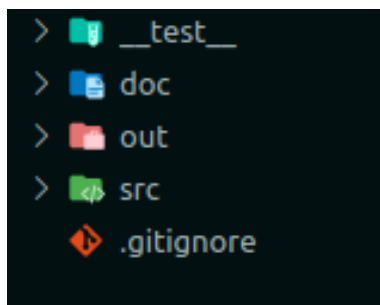


Figura 18: Distribuição de pastas.

### 4.1 Testes

Primeiramente, criamos a pasta *test* com o intuito de não poluirmos o nosso ambiente de programação e para não nos atrapalharmos quando fossemos fazer a execução do programa. Vale ressaltar que esta pasta foi de grande utilidade pois tiramos dela a implementação do vetor para o segundo cenário.

```
printf("Chave \n");
printf("%d\n", vet[i].registros.chave);

printf("Matriz de char \n");
for (l = 0; l < 10; l++)
{
    for (k = 0; k < 200; k++)
    {
        printf("%c", vet[i].registros.string[l][k]);
    }
}printf("\n");

printf("Vetor de float \n");
for (j = 0; j < 4; j++)
{
    printf("%f\n", vet[i].registros.real_value[j]);
}printf("\n\n");
```

Figura 19: Função Imprime.

### 4.2 Documentação

Por conseguinte, criamos a pasta *doc* para salvarmos a especificação do trabalho, quaisquer observações futuras e, inclusive, esta mesma documentação a ser entregue.

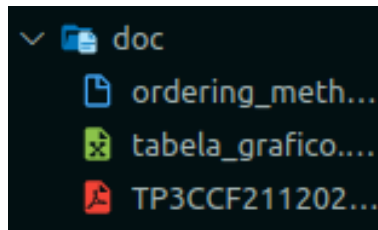


Figura 20: Pasta "Doc".

### 4.3 Arquivos

Ademais, temos a pasta *out*, onde são gerados os arquivos *.txt* com a média das comparações, movimentações e do tempo.

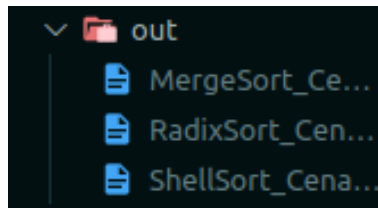


Figura 21: Pasta "Out".

### 4.4 Source

Por fim, temos a pasta *src*, sendo esta a principal para o nosso projeto, haja vista, que é nela em que encontramos os códigos de ordenação, execução, declaração dos registros e coleta dos dados.

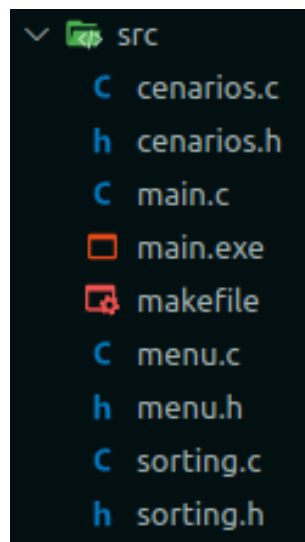
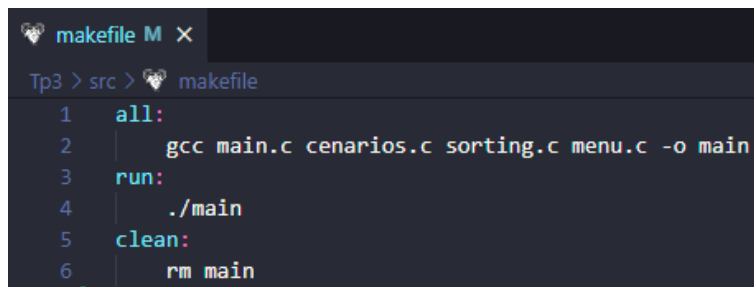


Figura 22: Pasta "Src".

### 4.5 Execução do Trabalho

Apesar de não ser uma pasta, vale ressaltar que optamos por usar um arquivo *makefile* para a execução do projeto:

A screenshot of a code editor window titled 'makefile M'. The editor shows a Makefile with the following content:

```
1 all:
2     gcc main.c cenarios.c sorting.c menu.c -o main
3 run:
4     ./main
5 clean:
6     rm main
```

The editor's breadcrumb shows the path 'Tp3 > src > makefile'.

Figura 23: MakeFile.

Assim, para compilar basta executar no terminal o comando *make*, para executar usamos o comando *./main* (vale ressaltar que deve-se instalar o *Make* antes de executar o algoritmo) e para deletar o arquivo de saída basta digitar *rm main*.

## 5 Conclusão

Posto isso, conclui-se que o trabalho em questão foi desenvolvido conforme o esperado, atingindo todas as especificações requeridas na descrição do mesmo em implementar os algoritmos de ordenação postos em sala, bem como dois extras.

Por conseguinte podemos ressaltar que o livro-texto da disciplina [3] foi de suma importância para o desenvolvimento do projeto, haja vista que as explicações dadas pelo autor nos ajudaram bastante a entender as etapas a serem executadas, a construção dos códigos e na disponibilização de alguns algoritmos de ordenação [1]. Além disso podemos ressaltar que vídeos como os disponibilizados na disciplina de Algoritmos e Estrutura de Dados I, foram de grande ajuda para o entendimento dos conceitos necessário para o desenvolvimento do projeto .

Em adição, é válido dizer que apesar das dificuldades na implementação do código, o trio foi capaz de superar e corrigir quaisquer erros no desenvolvimento do algoritmo. Por fim, verificou-se a assertiva para o objetivo do projeto em implementar os métodos de ordenação para inteiros e estruturas de dados.

## Referências

- [1] Implementation:<http://www2.dcc.ufmg.br/livros/algoritmos/implementacoes-04.php>.
- [2] Ordenacao:<https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao>.
- [3] Nivio Ziviani. *Projeto de Algoritmos com implementações em Pascal e C*, volume 3. 2010.