



TRABALHO I

Pesquisa e Ordenação de Dados

João Luis Almeida Santos

SUMÁRIO

- 1. Introdução**
Instruções e objetivos do trabalho
- 2. Desafios e Evolução do Código**
Metodologia, descrição da lógica, sumário de decisões
- 3. Implementação e Solução Principal**
Metodologia, descrição da lógica, sumário de decisões
- 4. Análise de Complexidade**
Análise do Algoritmo e definição matemática do tempo de execução
- 5. Conclusão**
Conclusão final a partir dos resultados e processo de execução do trabalho
- 6. Referências e Código-Fonte**
Referências utilizadas no relatório final e código-fonte. (Também em anexo.)

O projeto implementado também pode ser acessado pelo repositório do GITHUB, [através deste link](#).

I. INTRODUÇÃO

Este relatório apresenta o desenvolvimento e a análise de uma solução voltada para a ordenação externa de números inteiros positivos, conforme proposto no Trabalho T1 da disciplina. O objetivo central é aplicar técnicas de manipulação de arquivos e otimização de uso de memória principal, simulando um cenário em que os dados excedem a capacidade de armazenamento interno e precisam ser processados em blocos.

Apesar dos comentários estarem em português, a prática para nomear as funções e variáveis foram baseadas no inglês. O documento explora a lógica aplicada e discute as decisões de design algorítmico, identifica desafios enfrentados durante o desenvolvimento e realiza uma análise da complexidade do código.

O código clona o arquivo original para um arquivo que podemos ler e alterar sem alterar o arquivo principal, INPUT. Após, há a ordenação dos dados, e salvamento dos resultados no arquivo SAIDA.TXT. Este relatório estuda as decisões críticas tomadas durante a realização do trabalho, as estratégias de manipulação de arquivos temporários, e análise da complexidade.

Aqui estão as bibliotecas utilizadas e as variáveis globais usadas no código:

```
1  #ifdef _WIN32
2  // Configurado para o meu computador, alterar se for usar no Windows
3  #define INPUT "C:\\Users\\neoka\\CLionProjects\\Trabalho---POD\\dados.txt"
4  #define PROCESS "C:\\Users\\neoka\\CLionProjects\\Trabalho---POD\\processo.txt"
5  #define OUTPUT "C:\\Users\\neoka\\CLionProjects\\Trabalho---POD\\saida.txt"
6  #else
7  #define INPUT "dados.txt"
```

```

8  #define PROCESS "processo.txt"
9  #define OUTPUT "saida.txt"
10 #endif
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <string.h>

```

II. DESAFIOS E EVOLUÇÃO DO CÓDIGO

ENTENDIMENTO INICIAL ERRADO

Na primeira versão do código, desenvolvi uma solução baseada no carregamento completo dos dados em memória principal usando um array, seguido da aplicação do *Insertion Sort*. Essa abordagem, disponível no GitHub¹, partia da premissa equivocada de que a ordenação poderia ser feita somente lendo os arquivos e colocando os números em um array.

```

// Abordagem inicial (removida)
int* loadFullFile(int* total) {
    FILE *f = fopen(INPUT, "r");
    int *numbers = malloc(500 * sizeof(int)); // Alocação fixa
    // ... leitura completa do arquivo
    return numbers;
}

```

Esta implementação apresentava dois problemas críticos:

- Violação do requisito de ordenação externa
- Incompatibilidade com arquivos maiores que a memória principal

TRANSIÇÃO PARA ORDENAÇÃO EXTERNA

¹<https://github.com/lysz/Trabalho—POD/blob/main/antigo.c>

A necessidade de operar diretamente nos arquivos tornou evidente a inadequação da primeira abordagem. Muitas tentativas de implementação foram feitas, incluindo tentar fazer o bubble sort diretamente dentro do arquivo usando **fputc** sem criar um arquivo exterior. Além disso, a ideia de ler o arquivo, colocar em um array e só escrever por cima da saída.txt não condizia com a comanda do trabalho.

A versão final, embora tecnicamente correta, ainda apresenta limitações que seriam críticas em cenários reais de big data - particularmente o merge $\mathcal{O}(n^2)$ e a ausência de tratamento para duplicatas - mas serve como prova de conceito eficaz para os objetivos do trabalho.

III. IMPLEMENTAÇÃO DA SOLUÇÃO PRINCIPAL

MANIPULAÇÃO DE ARQUIVOS

A clonagem de arquivos é realizada pela função **cloneFile**. Todo o processo é feito no arquivo indicado pela variável global **PROCESS**, a partir do código a seguir:

```
1 void cloneFile(const char* origin, const char* destination) {
2     FILE *f = fopen(origin, "r");
3     FILE *clone = fopen(destination, "w");
4     if (f == NULL || clone == NULL) {
5         printf("[ERRO] Não foi possível abrir os arquivos para clonagem.\n");
6         if (f) fclose(f);
7         if (clone) fclose(clone);
8         return;
9     }
10    char content[1024];
11    while (fgets(content, sizeof(content), f)) {
12        fputs(content, clone);
13    }
14    fclose(f);
15    fclose(clone);
```

16 }

O processo de ordenação externa utiliza três arquivos:

- **INPUT:** Arquivo original (somente leitura)
- **PROCESS:** Cópia trabalhável dos dados
- **temp.txt:** Armazena chunks ordenados temporários

Apesar dos arquivos serem definidos aqui a partir destes nomes, eles foram, assim como a comanda pediu, nomeados como "dados.txt" e "saida.txt".

FASE 1: ORDENAÇÃO DE CHUNKS

A função **externalSort** divide o arquivo PROCESS em blocos de tamanho fixo, aplicando bubble sort em cada chunk:

```
1 void externalSort(int chunk_size) {
2     // Leitura do arquivo PROCESS
3     // Esse fscanf ignora os ;
4     while (fscanf(f, "%15[0-9];", num_str) == 1) {
5         numbers[count++] = atoi(num_str);
6         if (count == chunk_size) {
7             // Bubble sort no chunk atual
8             for (int i = 0; i < count - 1; i++) {
9                 for (int j = i + 1; j < count; j++) {
10                     if (numbers[i] > numbers[j]) swap();
11                 }
12             }
13             // Escrita no temp.txt
14         }
15     }
16 }
```

Para cada chunk de tamanho k , o custo é $\mathcal{O}(k^2)$. Considerando n elementos totais, teremos $\lceil n/k \rceil$ chunks, resultando em custo agregado $\mathcal{O}(nk)$.

FASE 2: MERGE

A consolidação via **mergeSortedChunks** carrega todos os chunks ordenados na memória e aplica uma ordenação final:

```
1 void mergeSortedChunks(FILE *output) {
2     // Leitura de todos os valores do temp.txt
3     int *values = /* ... */;
4
5     // Bubble sort completo dos valores
6     for (int i = 0; i < size - 1; i++) {
7         for (int j = i + 1; j < size; j++) {
8             if (values[i] > values[j]) swap();
9         }
10    }
11
12    // Escrita ordenada no OUTPUT
13 }
```

Esta fase adiciona custo $\mathcal{O}(n^2)$ ao processo total. Haveria a possibilidade de utilizar um algoritmo mais rápido, como o Insertion Sort, ou outros, mas preferi deixar da forma mais simples.

IV. ATENDIMENTO AOS REQUISITOS

A solução implementada atende aos requisitos principais do trabalho. A interface do usuário foi desenvolvida conforme as instruções, solicitando apenas a quantidade de elementos por vez e confirmando a criação do arquivo de saída. Entretanto, algumas limitações permanecem: o código não faz a separação por blocos na saída final, conforme pedido, e pode apresentar problemas com espaços múltiplos entre blocos.

V. ANÁLISE DE COMPLEXIDADE

```
1 int main() {
2     // Declaração de variável - O(1)
3     int quantidade_dados;
4
5     // Entrada do usuário - O(1)
6     printf("Quantidade de elementos por vez:\n");
7     scanf("%d", &quantidade_dados);
8
9     // Clonagem inicial - O(n)
10    cloneFile(INPUT, PROCESS);
11
12    // Ordenação externa - O(nk + n²)
13    externalSort(quantidade_dados);
14
15    // Geração do arquivo final - O(n)
16    cloneFile(PROCESS, OUTPUT);
17
18    // Limpeza do arquivo temporário - O(1)
19    remove(PROCESS);
20
21    // Confirmação de saída - O(1)
22    printf("Saída final disponível em [ SAIDA.TXT ]\n");
23
24    return 0; // Fim do programa - O(1)
25 }
```

$$\underbrace{\mathcal{O}(1)}_{\text{Entrada}} + \underbrace{2\mathcal{O}(n)}_{\text{Clone}} + \underbrace{\mathcal{O}(nk + n^2)}_{\text{Sorting}} = \mathcal{O}(nk + n^2)$$

Para grandes volumes de dados, o termo quadrático domina. Isso significa que esse algoritmo não é uma boa escolha caso haja um maior número de dados. Para esse caso, seriam necessárias mais alterações.

Para arquivos pequenos (até alguns milhares de números), o programa

funciona rápido. Porém, se o arquivo tiver muitos números (ex: 10.000+), o tempo de execução aumenta muito. Isso acontece porque usamos o método Bubble Sort duas vezes - primeiro nos blocos menores e depois juntando tudo.


VI. CONCLUSÃO

A partir do presente trabalho, é possível concluir que o algoritmo desenvolvido, apesar de suas limitações temporais para grandes volumes de dados, demonstrou com sucesso os princípios fundamentais da ordenação externa utilizando arquivos de texto. Conforme ilustrado na Figura ??, a execução do código segue o fluxo esperado.

O código, quando executado, se apresenta como demonstrado na figura 2. O código é performático mais rapidamente quando executado com mais chunks.

VII. REFERÊNCIAS E CÓDIGO-FONTE

```
1
2  #ifdef _WIN32
3  // Configurado para o meu computador, alterar se for usar no Windows
4  #define INPUT "C:\\Users\\neoka\\CLionProjects\\Trabalho---POD\\dados.txt"
5  #define PROCESS "C:\\Users\\neoka\\CLionProjects\\Trabalho---POD\\processo.txt"
6  #define OUTPUT "C:\\Users\\neoka\\CLionProjects\\Trabalho---POD\\saida.txt"
7  #else
8  #define INPUT "dados.txt"
9  #define PROCESS "processo.txt"
10 #define OUTPUT "saida.txt"
11 #endif
12
13 #include <stdio.h>
14 #include <stdlib.h>
```



A ordenação a ser implementada deve ser realizada sobre números inteiros positivos vindos de um arquivo. Os números inteiros serão separados com o caractere `;` (ponto-e-vírgula) não havendo espaço entre os números e ponto-e-vírgula. Os blocos em cada arquivo deverão ser separados por um espaço em branco.

A única entrada de dados por parte do usuário é a quantidade de elementos a serem carregados por vez em memória (M). O arquivo contendo os números inteiros que serão ordenados deve se chamar obrigatoriamente dados.txt. e o arquivo resultante contendo os números já ordenados deve ter seu nome impresso em tela.

Figura 1: Instruções do Trabalho T1

```
Quantidade de elementos por vez:  
1  
Saída final disponível em [ SAIDA.TXT ]
```

Figura 2: Console na execução do código

```

15 #include <string.h>
16
17
18 void cloneFile(const char* origin, const char* destination) {
19     FILE *f = fopen(origin, "r");
20     FILE *clone = fopen(destination, "w");
21     if (f == NULL || clone == NULL) {
22         printf("[ERRO] Não foi possível abrir os arquivos para clonagem.\n");
23         if (f) fclose(f);
24         if (clone) fclose(clone);
25         return;
26     }
27     char content[1024];
28     while (fgets(content, sizeof(content), f)) {
29         fputs(content, clone);
30     }
31     fclose(f);
32     fclose(clone);
33 }
34
35 void mergeSortedChunks(FILE *output) {
36     // Feito no processo final
37     FILE *temp = fopen("temp.txt", "r");
38     if (!temp) {
39         printf("[ERRO] Arquivo temporário não encontrado.\n");
40         return;
41     }
42
43     int *values = NULL;
44     int size = 0, capacity = 0;
45     char num_str[16];
46
47     while (fscanf(temp, "%15[0-9];", num_str) == 1) {
48         if (size >= capacity) {
49             capacity = (capacity == 0) ? 1 : capacity * 2;
50             values = realloc(values, capacity * sizeof(int));
51         }
52         values[size++] = atoi(num_str);

```

```

53     }
54     fclose(temp);
55
56     for (int i = 0; i < size - 1; i++) {
57         for (int j = i + 1; j < size; j++) {
58             if (values[i] > values[j]) {
59                 int temp = values[i];
60                 values[i] = values[j];
61                 values[j] = temp;
62             }
63         }
64     }
65
66     // Escreve os arquivos de volta pra OUTPUT
67     for (int i = 0; i < size; i++) {
68         fprintf(output, "%d;", values[i]);
69     }
70
71     free(values);
72 }
73
74 void externalSort(int chunk_size) {
75     FILE *f = fopen(PROCESS, "r");
76     if (!f) {
77         printf("[ERRO] Arquivo não encontrado.\n");
78         return;
79     }
80
81     int *numbers = malloc(chunk_size * sizeof(int));
82     if (!numbers) {
83         printf("[ERRO] Falha na alocação de memória.\n");
84         fclose(f);
85         return;
86     }
87
88     FILE *temp = fopen("temp.txt", "w");
89     if (!temp) {
90         printf("[ERRO] Não foi possível criar temp.txt.\n");

```

```

91         fclose(f);
92         free(numbers);
93         return;
94     }
95     fclose(temp);
96
97     int count = 0;
98     char num_str[16];
99
100     while (fscanf(f, "%15[0-9];", num_str) == 1) {
101         numbers[count++] = atoi(num_str);
102         if (count == chunk_size) {
103             for (int i = 0; i < count - 1; i++) {
104                 for (int j = i + 1; j < count; j++) {
105                     if (numbers[i] > numbers[j]) {
106                         int tmp = numbers[i];
107                         numbers[i] = numbers[j];
108                         numbers[j] = tmp;
109                     }
110                 }
111             }
112             // Manda os dados ordenados para arquivo temp
113             temp = fopen("temp.txt", "a");
114             for (int i = 0; i < count; i++) {
115                 fprintf(temp, "%d;", numbers[i]);
116             }
117             fclose(temp);
118             count = 0;
119         }
120     }
121
122     // Números restantes
123     if (count > 0) {
124         for (int i = 0; i < count - 1; i++) {
125             for (int j = i + 1; j < count; j++) {
126                 if (numbers[i] > numbers[j]) {
127                     int tmp = numbers[i];
128                     numbers[i] = numbers[j];

```

```

129         numbers[j] = tmp;
130     }
131 }
132 }
133 temp = fopen("temp.txt", "a");
134 for (int i = 0; i < count; i++) {
135     fprintf(temp, "%d;", numbers[i]);
136 }
137 fclose(temp);
138 }
139
140 fclose(f);
141 free(numbers);
142
143 // Fase de merge
144 FILE *output = fopen(PROCESS, "w");
145 if (!output) {
146     printf("[ERRO] Falha ao abrir arquivo de saída.\n");
147     return;
148 }
149 mergeSortedChunks(output);
150 fclose(output);
151 remove("temp.txt");
152 }
153
154 int main() {
155     int quantidade_dados;
156     printf("Quantidade de elementos por vez:\n");
157     if (scanf("%d", &quantidade_dados) != 1 || quantidade_dados <= 0) {
158         printf("[ERRO] Entrada inválida\n");
159         return 0;
160     }
161
162     cloneFile(INPUT, PROCESS);
163     externalSort(quantidade_dados);
164     cloneFile(PROCESS, OUTPUT);
165     remove(PROCESS);
166

```

```
167     printf("Saída final disponível em [ SAIDA.TXT ]\n");
168     return 0;
169 }
170
```