Relatório Projeto Entregável 03 - Volta USP (Parte 2)

João Pedro Neves e Juan Pablo Tomba

26 de setembro de 2025

1 Introdução

Neste relatório, explicaremos o processo de criação de uma versão de código que visa resolver o seguinte problema: ler uma sequência de participantes de uma competição no CEFER, onde cada linha contem NOME - (usp ou externo), e depois separando em dois vetores(usp e externo), cada um com o número de caracteres de cada nome, separados em sua devida categoria. Após isso, utilizando o **Bubble Sort**, ordenamos os vetores, contando o número de trocas e comparações realizados pelo algoritmo.

Serão discutidos aspectos como tempo de execução, manutenibilidade, simplicidade, repertório e análise de desempenho.

2 Metodologia

O primeiro passo do projeto foi criar um repositório público no GitHub onde ambos os membros do grupo poderiam subir os códigos. Dessa forma, poderíamos desenvolver simultaneamente os diferentes códigos e ao final ter uma forma prática de juntá-los e/ou acessá-los.

https://github.com/joaopneves1570/Lab-ICC2/tree/main/aula4

Em seguida, criados os arquivos para o projeto, utilizamos o ambiente de desenvolvimento do Visual Studio Code para escrever e compilar os mesmos.

3 Códigos

Após isso, elaboramos o seguinte código:

3.1 Recursivo com bubble sort

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

/*
Joao Pedro Neves: 14713404
Juan Pablo Tomba: 15638548
*/

int conta_letras(char string[]){
    char letra = string[0];
    if (letra == '-')
        return 0;

    return (isalnum(string[0])? 1 : 0) + conta_letras(string + 1);

}

void bubbleSort(char v[], int n, int* resultados){
    if (resultados != NULL){
        char temp;
        int trocas = 0;
        int comparacoes = 0;

```

```
        for (int i = 0; i < n - 1; i++){
            for (int j = 0; j < n - i - 1; j++){
                comparacoes++;
                if (v[j] > v[j+1]){
                    temp = v[j];
                    v[j] = v[j+1];
                    v[j+1] = temp;
                    trocas++;
                }
            }
        }

        resultados[0] = comparacoes;
        resultados[1] = trocas;
    }

}


int main(){
    char string[200];

    char USP[100000];
    char Externa[100000];

    int i = 0, j = 0;

    while (fgets(string, 201, stdin) != NULL){
        string[strcspn(string, "\n")] = '\0';
        int tamanho_string = strlen(string) - 1;
        if (string[tamanho_string] == 'a')
            Externa[i++] = (char)conta_letras(string);
        else
            USP[j++] = (char)conta_letras(string);
    }

    int resultados_USP[2];
    int resultados_Externa[2];

    bubbleSort(USP, j, resultados_USP);
    printf("USP - [");
    if (j > 0){
        for (int k = 0; k < j - 1; k++) printf("%d, ", USP[k]);
        printf("%d]\n", USP[j - 1]);
    } else {
        printf("]\n");
    }
    printf("Comparacoes: %d, Trocas: %d\n", resultados_USP[0], resultados_USP[1]);

    printf("\n");

    bubbleSort(Externa, i, resultados_Externa);
    printf("Externa - [");
    if (i > 0){
        for (int k = 0; k < i - 1; k++) printf("%d, ", Externa[k]);
        printf("%d]\n", Externa[i - 1]);
    } else {
        printf("]\n");
    }
    printf("Comparacoes: %d, Trocas: %d", resultados_Externa[0], resultados_Externa[1]);

    return 0;
}
```

4 Resultados

Após submeter o código, obtivemos os seguintes resultados:

4.1 Tempo de Execução

Para medir os tempos de execução, fizemos o upload do código no RunCodes e de lá retiramos o tempo de execução fornecido pela própria plataforma.

- **Caso 1**: 0.0013 s
- **Caso 2**: 0.0019 s
- **Caso 3**: 0.4427 s
- **Caso 4**: 1.9095 s
- **Caso 5**: 0.8978 s

4.2 Análise dos três possíveis cenários:

Após submeter o código, rodamos localmente o código com uma entrada de 100 linhas (50 usp e 50 externa) em três diferentes casos(Os arquivos com as entradas estão disponíveis na página do github): Vetor já ordenado, ordenado de forma inversa, e aleatoriamente ordenado. Esses casos devem corresponder ao melhor caso, pior caso e caso médio respectivamente.

4.2.1 Melhor Caso - Vetor Ordenado

No cenário em que os vetores já estavam ordenados, o algoritmo não precisou realizar trocas, apenas comparações:

- USP: 1225 comparações, 0 trocas
- Externa: 1225 comparações, 0 trocas

Isso confirma que o Bubble Sort é eficiente quando os dados já estão na ordem desejada, realizando apenas comparações.

4.2.2 Pior Caso - Vetor Inversamente Ordenado

Quando os vetores estavam ordenados de forma inversa, o algoritmo precisou realizar diversas trocas e comparações, caracterizando o pior caso:

- USP: Comparações 1225, Trocas: 1225
- Externa: Comparações 1225, Trocas: 1225

Esse resultado evidencia a complexidade quadrática do Bubble Sort no pior caso, pois o número de comparações e trocas realizadas (1225) para um vetor de tamanho $n = 50$ coincide com o valor esperado da fórmula

$$\frac{n \cdot (n-1)}{2} = \frac{50 \cdot 49}{2} = 1225,$$

que cresce proporcionalmente a n^2. Dessa forma, observa-se que o comportamento experimental confirma a análise teórica de complexidade $O(n^2)$.

4.2.3 Caso Médio - Vetor Aleatoriamente Ordenado

No caso de vetores com ordem aleatória, o número de trocas e comparações intermediário apresenta variação:

- USP: 1225 comparações, 621 trocas
- Externa: 1225 comparações, 646 trocas

Esses resultados representam o comportamento médio esperado do algoritmo, mantendo complexidade quadrática, mas com número de trocas menor que o pior caso. Nota-se que o número de comparações nunca muda, já que todas as entradas tem o mesmo número de nomes para usp e externa.

4.3 Manutenibilidade

A manutenibilidade está relacionada à clareza do código e à facilidade de adaptá-lo para futuras modificações, como mudança no formato da entrada ou inclusão de novas categorias:

- **Recursivo:** concentra boa parte da lógica em uma função recursiva (`conta_letras`). Embora elegante, a recursão torna a manutenção menos direta, principalmente para quem não está acostumado com recursão.Caso seja necessário expandir para além de USP e externos, exigiriam maior cuidado para modificação no código. A função **bubbleSort** pode ser reaproveitada em outro códigos, com alguns ajustes, já que só precisa do vetor a ser ordenado e o tamanho do mesmo.

4.4 Simplicidade

A simplicidade avalia a legibilidade e a facilidade de compreensão:

- **Recursivo:** menos simples, já que envolve chamadas recursivas e exige compreender como o problema é reduzido a casos menores. Apesar disso, sua implementação é compacta. Já a função **bubbleSort** é bem simples, com dois laços e um if, que compara o elemento j com o elemento j + 1, e caso o j for maior, realiza a troca.

5 Conclusão

A análise da versão **recursiva** evidencia que o algoritmo apresenta uma solução compacta e elegante para a contagem de caracteres nos nomes, utilizando a função `conta_letras`.

O uso do **Bubble Sort** permite ordenar os vetores de forma clara, com contagem de comparações e trocas. No melhor caso (vetor já ordenado), o algoritmo realiza apenas comparações, sem trocas, mostrando eficiência nesse cenário. Já no pior caso (vetor inversamente ordenado), o número de comparações e trocas cresce significativamente, refletindo a complexidade quadrática $O(n^2)$ do Bubble Sort.

Portanto, a versão recursiva cumpre bem seu objetivo de forma concisa e funcional, demonstrando um bom equilíbrio entre elegância da implementação e eficiência na ordenação para entradas pequenas.