Relatório dissertativo da Atividade Avaliativa 4 (19/09/2025)

Nome: Davi Gabriel Domingues

Nº USP: 15447497

Objetivo do documento: comparar o desempenho dos códigos produzidos, a partir do exercício "Entrega 03 – pt.02", do dia 19/09/2025, com as devidas especificações. No caso, será discutido o desempenho final observado (notação "Big O"), além dos pontos nos quais foram pertinentes ao desenvolvimento do código, para a resolução da situação problema em si.

Versão Final)

Temos a seguinte versão:

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

typedef struct {
    int *tamanhos;
    int tamanho;
    int capacidade;
} Grupo;

typedef struct {
    int comparacoes;
    int movimentacoes;
} Dados;

void adicionarTamanho(Grupo *grupo, int tamanho) { /* Função para contabilizar o tamanho de cada grupo,
sob o princípio de alocação dinâmica */
    if (grupo->tamanho >= grupo->capacidade) {
        grupo->capacidade = (grupo->capacidade == 0) ? 10 : grupo->capacidade * 2; /* Multiplicação em potência de 2
        para diminuir o gasto computacional de aumento linear de espaço (procedimento da entrega03 - parte 1). */
        grupo->tamanhos = realloc(grupo->tamanhos, grupo->capacidade*sizeof(int));
    }

    grupo->tamanhos[grupo->tamanho] = tamanho;
    grupo->tamanho++;
}
```

```c
int contarCaracteres(char *nome) { /* Função para contagem dos caracteres de cada nome declarado
no seu respectivo grupo do evento (usp ou externa). */
    int tamanho = 0;

    for (int i = 0; nome[i] != '\0'; i++) {
        if (nome[i] != ' ')
            tamanho++;
    }

    return tamanho;
}
```

```c
Dados ordenacaoBubbleSort(int *vetor, int tamanho) { /* Uso do Cocktail Shaker Sort (shake sort),
que é uma otimização do bubble sort */
    Dados dados = {0, 0}; // Inicialização padrão.

    if (tamanho <= 1 || vetor == NULL)
        return dados;

    int inicio = 0;
    int fim = tamanho - 1;

    while (inicio < fim) {
        // Varredura da esquerda para a direita
        for (int i = inicio; i < fim; i++) {
            dados.comparacoes++;
            if (vetor[i] > vetor[i+1]) {
                int temporario = vetor[i];
                vetor[i] = vetor[i+1];
                vetor[i+1] = temporario;
                dados.movimentacoes++;
            }
        }
        // Diminui o fim pois o maior já está no lugar
        fim--;

            // Varredura da direita para a esquerda
            for (int i = fim; i > inicio; i--) {
                dados.comparacoes++;
                if (vetor[i-1] > vetor[i]) {
                    int temporario = vetor[i-1];
                    vetor[i-1] = vetor[i];
                    vetor[i] = temporario;
                    dados.movimentacoes++;
                }
            }
            // Aumenta o inicio pois o menor já está no lugar
            inicio++;
        }

        return dados;
}

void imprimirGrupo(char *tipo, Grupo grupo, Dados dados) { // Função padronizada de impressão dos dados.
    printf("%s - [", tipo);

    for (int i = 0; i < grupo.tamanho; i++) { /* Loop para percorrer as structs e captar os
    comprimentos de cada nome presente. */
        printf("%d", grupo.tamanhos[i]);

        if (i < grupo.tamanho - 1)
            printf(", ");
    }

    printf("]\n");
    printf("Comparações: %d, Trocas: %d", dados.comparacoes, dados.movimentacoes);
}
```

```c
int main() {
    Grupo usp = {NULL, 0, 0};
    Grupo externa = {NULL, 0, 0};
    char buffer[256];

    while (fgets(buffer, sizeof(buffer), stdin)) { /* Loop de leitura de dados até lidar com EOF ("End Of File")
        -> comando CTRL+D (terminal Linux), CTRL+D (Windows). */
        char nome[256], tipo[256];

        sscanf(buffer, "%[^-]-%s", nome, tipo); // Regex para fazer função split()

        int tamanho = contarCaracteres(nome);

        if (strcmp(tipo, "usp") == 0)
            adicionarTamanho(&usp, tamanho);

        else if (strcmp(tipo, "externa") == 0)
            adicionarTamanho(&externa, tamanho);
    }

    Dados dusp = ordenacaoBubbleSort(usp.tamanhos, usp.tamanho);
    Dados dexterna = ordenacaoBubbleSort(externa.tamanhos, externa.tamanho);

    imprimirGrupo("USP", usp, dusp);
    printf("\n\n");
    imprimirGrupo("Externa", externa, dexterna);
    printf("\n");
```

```c
    Dados dusp = ordenacaoBubbleSort(usp.tamanhos, usp.tamanho);
    Dados dexterna = ordenacaoBubbleSort(externa.tamanhos, externa.tamanho);

    imprimirGrupo("USP", usp, dusp);
    printf("\n\n");
    imprimirGrupo("Externa", externa, dexterna);
    printf("\n");

    return 0;
}
```

Discussão técnica: O algoritmo segue um esquema bastante parecido com o da entrega passada, do trabalho do dia 12/09/2025, porém implementa um algoritmo de ordenação para se analisar a quantia total de comparações e de movimentações realizadas durante o processo de ordenação dos dados, no caso, o algoritmo é o “BubbleSort” e os dados são, justamente, os nomes dos participantes de cada grupo devido (usp ou externa). O algoritmo de ordenação selecionado funciona basicamente pela seguinte analogia: “as bolhas de menor densidade sobem para o início do vetor e as de maior densidade descem para o final”, ou seja, o programa percorre todo o vetor para comparar um certo valor fixo, havendo a troca direta, caso ele seja maior do que o dado a comparado no momento de percorrimento. Percebe-se o uso de dois laços de repetição aninhados, configurando o **caso médio** como $O(n^2)$, já que cada laço, por definição, tem rendimento $O(n)$.

Caso	Status	Tempo de CPU
Caso 1	Correto	0.0012 s
Caso 2	Correto	0.0024 s
Caso 3	Correto	0.6520 s
Caso 4	Correto	2.6922 s
Caso 5	Correto	0.8949 s

Tempo de execução dos 5 casos padrão do runcodes (casos gerais/médios)

Além disso, temos as perspectivas da contagem de comparações e de movimentações de todos os casos possíveis, sendo elas:
Melhor caso:

Cenário de execução 5 do problema

A situação é tal que o vetor de dados já se apresenta devidamente ordenado. Apesar do objetivo almejado para a ordenação, o algoritmo “BubbleSort”, por causa de seu estilo de funcionamento, ainda realizará comparações durante o percorrimento do vetor em si.

Equações gerais experimentais:

- Comparações: C = n(n-1)/2
- Movimentações: M = 0

Pior caso:

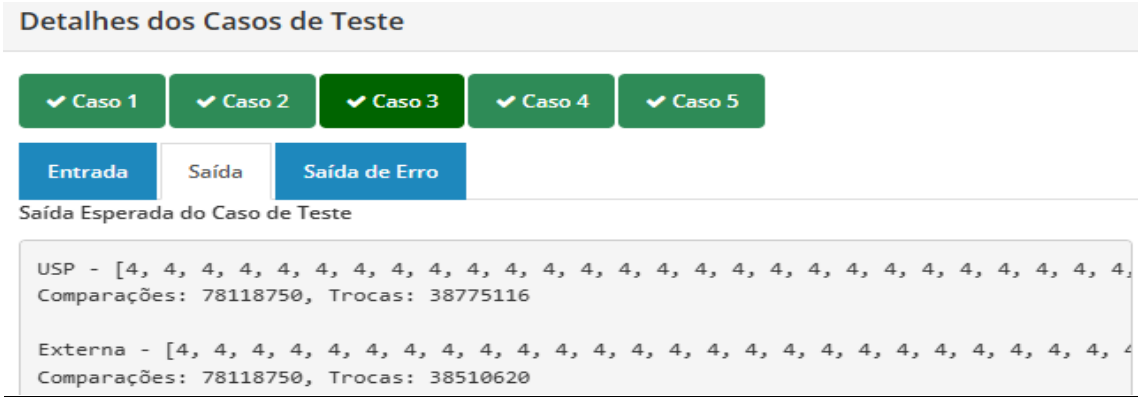
Cenário de execução 4 do problema

A situação é tal que o vetor de dados já se apresenta devidamente ordenado, porém de forma decrescente, ou seja, os maiores dados são os primeiros, em detrimento do último. Dessa maneira, o algoritmo “BubbleSort”, por causa de seu estilo de funcionamento, realizará todas as comparações possíveis durante o percorrimento do vetor, além do número máximo de movimentações/trocas (processo de swap e do uso de variável temporária).

Equações gerais experimentais:

- Comparações: C = n(n-1)/2
- Movimentações: M = n(n-1)/2

Caso médio:



Cenário de execução 3 do problema (o caso 1 ou 2 também serviriam para ilustração)

A situação é tal que o vetor de dados se apresenta em um caso mais geral, se comparado ao melhor e ao pior cenário de implementação do "BubbleSort". Considerando, de acordo com a literatura, que a probabilidade do caso médio abarcar uma estrutura de dados, de certo modo, ordenada é de 50%, ainda teremos o percorrimento padrão do algoritmo de ordenação, mas com uma quantia relativamente intermediária de trocas de valores (entre 0 e o averiguado pelo pior caso).

Equações gerais experimentais:

- Comparações: C = n(n-1)/2
- Movimentações: M ~= n(n-1)/4 (chance geral de 50% dos elementos estarem ordenados: M_piorCaso/2)

Obs[1]: o código produzido não otimizou a questão da parada de comparações, ou seja, não utilizou um flag de parada para sinalizar que a troca ocorreu na interação passada do segundo laço aninhado. Caso isso fosse utilizado, o total de comparações, no melhor caso, valeria C = n – 1. Dessa forma, todos os cenários de casos possíveis possuem o mesmo desempenho $O(n^2)$, porém o melhor caso teria O(n), se, justamente, a função de ordenação apresentasse a otimização discutida.

Obs[2]: o algoritmo de ordenação "BubbleSort" utilizado para esse problema possui adaptações as quais otimizam a sua versão original, sendo a função de ordenação usada para o problema denominada de "ShakeSort". Basicamente, em vez do algoritmo sempre percorrer sempre da esquerda para a direita, ele fará uma ida (esquerda → direita) e depois uma volta (direita → esquerda). Isso permite

"empurrar" ao mesmo tempo os maiores valores para o fim e os menores para o início, garantindo um menor gasto de tempo de CPU (menor intervalo de execução) no melhor caso, quando não há trocas, embora a complexidade média e no pior caso permaneçam $O(n^2)$. Nesse cenário, o uso da memória cache do computador é otimizado porque os acessos contíguos reutilizam referências dos valores já percorridos pelos laços iterativos do algoritmo, reduzindo assim o custo computacional de processamento sequencial dos dados, embora esse padrão de acesso também seja comum a outras variações lineares de ordenação, como o “InsertionSort”.

Obs[3]: é possível notar que o total de comparações realizadas pelo algoritmo “BubbleSort”, quaisquer tipos de cenários que enfrente/lide, implementado é o mesmo, nesse caso a fórmula fechada para o somatório padrão de Gauss. Logo, o total de comparações de valores, durante o percorrimento do vetor de dados, será relativo ao total de dados de entrada, sob um padrão quadrático, por definição, denunciando, justamente, o padrão $O(n^2)$ de eficiência do “BubbleSort” aplicado no contexto da “Volta USP”.