Relatório Projeto Entregável 04 - Seguindo Ordens

João Pedro Neves e Juan Pablo Tomba

9 de outubro de 2025

1 Introdução

Neste relatório, explicaremos o processo de criação de uma versão de código que visa resolver o seguinte problema: ajudar um pai a organizar os brinquedos de sua filha, utilizando o bubble sort, insertion sort, merge sort e quick sort.

Serão discutidos aspectos como tempo de execução, descrição de como cada algoritmo funciona e discussão sobre suas complexidades.

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/aula5

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

3 Código

Após isso, elaboramos o seguinte código:

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

typedef struct brinquedo B;

enum peso{
    primeiro,
    segundo,
    terceiro,
    quarto,
    quinto,
    sexto,
    setimo
};

struct brinquedo{
    float tamanho;
    float nota;
    int id;
    enum peso ordem;
};

int comparaBrinquedos(B brinquedo1, B brinquedo2){
    if (brinquedo1.ordem != brinquedo2.ordem)
        return (brinquedo1.ordem > brinquedo2.ordem ? 1 : 0);

    if (brinquedo1.tamanho != brinquedo2.tamanho)
        return (brinquedo1.tamanho > brinquedo2.tamanho ? 1 : 0);

```

```
31      if (brinquedo1.nota != brinquedo2.nota)
32          return (brinquedo1.nota < brinquedo2.nota ? 1 : 0);
33
34      return (brinquedo1.id > brinquedo2.id ? 1 : 0);
35  }
36
37  void bubbleSort(B brinquedos[], int n){
38      B aux;
39      for (int i = 0; i < n; i++){
40          int trocou = 0;
41          for (int j = 0; j < n - 1 - i; j++){
42              if (comparaBrinquedos(brinquedos[j], brinquedos[j+1])){
43                  aux = brinquedos[j];
44                  brinquedos[j] = brinquedos[j+1];
45                  brinquedos[j+1] = aux;
46                  trocou = 1;
47              }
48          }
49          if (trocou == 0) break;
50      }
51  }
52
53  void insertionSort(B brinquedos[], int n){
54      B aux;
55
56      for (int i = 0; i < n; i++){
57          B atual = brinquedos[i];
58          int j = i - 1;
59          while (j >= 0 && comparaBrinquedos(brinquedos[j], atual)){
60              brinquedos[j+1] = brinquedos[j];
61              j = j - 1;
62          }
63          brinquedos[j+1] = atual;
64      }
65  }
66
67  void combina(B brinquedos[], int inicio, int meio, int fim){
68      int n1 = meio - inicio + 1;
69      int n2 = fim - meio;
70
71      B* esq = (B*)malloc(sizeof(B)*n1);
72      B* dir = (B*)malloc(sizeof(B)*n2);
73
74      for (int i = 0; i < n1; i++)
75          esq[i] = brinquedos[inicio + i];
76      for (int j = 0; j < n2; j++)
77          dir[j] = brinquedos[meio + 1 + j];
78
79      int i = 0, j = 0, k = inicio;
80
81      while (i < n1 && j < n2){
82          if (!comparaBrinquedos(esq[i], dir[j])){
83              brinquedos[k++] = esq[i++];
84          } else {
85              brinquedos[k++] = dir[j++];
86          }
87      }
88
89      while (i < n1){
90          brinquedos[k++] = esq[i++];
91      }
92
93      while (j < n2){
94          brinquedos[k++] = dir[j++];
95      }
96
```

```
97      free(esq);
98      free(dir);
99
100 }
101
102 void mergeSort(B brinquedos[], int inicio, int fim){
103     if (inicio < fim){
104         int meio = (inicio + fim)/2;
105         mergeSort(brinquedos, inicio, meio);
106         mergeSort(brinquedos, meio + 1, fim);
107         combina(brinquedos, inicio, meio, fim);
108     }
109 }
110
111 int reparte(B brinquedos[], int inicio, int fim){
112     B pivo = brinquedos[fim];
113
114     B aux;
115
116     int j = inicio;
117     int i = j - 1;
118     for(j; j < fim; j++){
119         if (comparaBrinquedos(pivo, brinquedos[j])){
120             i++;
121             aux = brinquedos[i];
122             brinquedos[i] = brinquedos[j];
123             brinquedos[j] = aux;
124         }
125     }
126
127     aux = brinquedos[i + 1];
128     brinquedos[i + 1] = brinquedos[fim];
129     brinquedos[fim] = aux;
130
131     return i + 1;
132 }
133
134 void quickSort(B brinquedos[], int inicio, int fim){
135     if (inicio < fim){
136         int pivo = reparte(brinquedos, inicio, fim);
137
138         quickSort(brinquedos, inicio, pivo - 1);
139         quickSort(brinquedos, pivo + 1, fim);
140     }
141 }
142
143 int main() {
144     int n;
145     scanf("%d", &n);
146
147     char cores[7][10] = {"amarelo", "azul", "branco", "preto", "rosa", "verde", "vermelho"};
148
149     B* brinquedos = (B*)malloc(sizeof(B)* n);
150     for (int i = 0; i < n; i++){
151         char cor[20];
152         float tamanho, nota;
153         scanf("%s %f %f", cor, &tamanho, &nota);
154
155         brinquedos[i].tamanho = tamanho;
156         brinquedos[i].nota = nota;
157         brinquedos[i].id = i;
158
159         for (int j = 0; j < 7; j++){
160             if (strcmp(cor, cores[j]) == 0){
161                 brinquedos[i].ordem = j;
162                 break;
```

```
            }
        }
    }

    int ordenador;
    scanf("%d", &ordenador);

    switch (ordenador)
    {
    case 1:
        bubbleSort(brinquedos, n);
        break;

        case 2:
        insertionSort(brinquedos, n);
        break;

        case 3:
        mergeSort(brinquedos, 0, n-1);
        break;

    default:
        quickSort(brinquedos, 0, n-1);
        break;
    }

    for (int i = 0; i < n; i++){
        printf("%d;", brinquedos[i].id);
    }

    free(brinquedos);
    return 0;
}
```

4 Funcionamento de cada algoritmo

Nesta seção, explicaremos como cada algoritmo utilizado funciona:

4.1 Bubble Sort

Funcionamento: Percorre o vetor (n-1) vezes, comparando elementos adjacentes e trocando-os se estiverem fora de ordem. A cada passada, o maior elemento é alocado para o final do vetor. Mas utilizamos o Bubble Sort aprimorado, em que, caso em uma passagem pelo vetor ele não realizar nenhuma troca, o algoritmo para, pois o vetor já está ordenado.

Complexidades:

- Melhor caso: $O(n)$ — quando o vetor já está ordenado
- Caso médio: $O(n^2)$
- Pior caso: $O(n^2)$ — vetor em ordem inversa

4.2 Insertion Sort

Funcionamento: A cada passagem, ele fixa o n-ésimo elemento, e insere o próximo na posição correta, deslocando os outros elementos para a inserção.

Complexidades:

- Melhor caso: $O(n)$ - quando o vetor já esta ordenado
- Caso médio: $O(n^2)$
- Pior caso: $O(n^2)$

4.3 Merge Sort

Funcionamento: Baseia-se na técnica de divisão e conquista. O vetor é dividido em duas metades, cada uma é ordenada recursivamente (seguindo o mesmo algoritmo) e, em seguida, as duas metades são juntadas em ordem.

Complexidades:

- **Note que em todos os casos, a complexidade não muda**
- Melhor caso: $O(n \log n)$
- Caso médio: $O(n \log n)$
- Pior caso: $O(n \log n)$

4.4 Quick Sort

Funcionamento: Também utiliza divisão e conquista. Escolhe-se um pivô e o vetor é dividido de forma que os menores valores fiquem à esquerda e os maiores à direita. O processo é repetido recursivamente nas divisões do vetor. Esse algoritmo é o único, entre os utilizados, que **NÃO É ESTÁVEL**

Complexidades:

- Melhor caso: $O(n \log n)$
- Caso médio: $O(n \log n)$
- Pior caso: $O(n^2)$
- **Obs:** seu pior caso é quando o pivô escolhido é sempre o menor ou o maior elemento do conjunto, fazendo com que o vetor fique dividido de forma muito desigual.

5 Resultados

5.1 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**: 0.0011 s
- **Caso 2**: 0.2636 s
- **Caso 3**: 0.1566 s
- **Caso 4**: 0.1195 s
- **Caso 5**: 0.1482 s

5.2 Observação

Observa-se que, no **caso 2**, foi utilizado o *Insertion Sort* com uma entrada de 30.000 brinquedos. Já no **caso 4**, foi aplicado o *Quick Sort* com 199.999 brinquedos. Mesmo com uma entrada significativamente maior, o *Quick Sort* apresentou um tempo de execução menor. Essa diferença ocorre devido à complexidade dos algoritmos: enquanto o *Quick Sort* possui complexidade $O(n \log n)$, o *Insertion Sort* apresenta complexidade $O(n^2)$. Assim, o *Quick Sort* é consideravelmente mais eficiente em casos médios e grandes entradas.

6 Conclusão

A análise dos algoritmos de ordenação implementados — *Bubble Sort aprimorado*, *Insertion Sort*, *Merge Sort* e *Quick Sort* — evidencia as diferenças significativas entre suas implementações e desempenhos.

O **Bubble Sort aprimorado**, embora simples, mostrou-se eficiente apenas para pequenas entradas, graças à verificação antecipada que interrompe o processo quando o vetor já está ordenado. O **Insertion Sort**, por sua vez, apresentou comportamento semelhante, sendo também mais indicado para vetores curtos ou quase ordenados.

Já os algoritmos baseados em *divisão e conquista* — **Merge Sort** e **Quick Sort** — apresentaram desempenhos superiores. O *Merge Sort* manteve sua complexidade $O(n \log n)$ mesmo no pior caso, enquanto o *Quick Sort* se destacou pelo menor tempo de execução em média, especialmente em grandes volumes de dados, apesar de possuir um pior caso $O(n^2)$.

De modo geral, algoritmos quadráticos como o *Bubble Sort* e o *Insertion Sort* são adequados para entradas pequenas, enquanto os algoritmos de complexidade $O(n \log n)$ — *Merge Sort* e *Quick Sort* — são preferíveis para entradas maiores, apresentando maior eficiência.