Relatorio de Laboratório de Introdução à Ciência da Computação II – Avaliativo 2

Leonardo Kenzo Tanaka e Pedro Teidi de Sá Yamacita

8 de outubro de 2025

1 Análise do Bubble Sort

```
void BubbleSort(LISTA *lista){

    // Percorre o array comparando o elemento i com i + 1
    for (int i = 0; i < lista->tamanho; i++){
        bool estaOrdenado = true;
        for (int j = 0; j < lista->tamanho - 1; j++){

            // Usa a funcao PodeTrocar() para fazer as comparacoes dos brinquedos
            if (PodeTrocar(lista->elementos[j], lista->elementos[j + 1])){

            // Troca o brinquedo
                BRINQUEDO *aux = lista->elementos[j];
                lista->elementos[j] = lista->elementos[j + 1];
                lista->elementos[j + 1] = aux;
                estaOrdenado = false;
            }
        }

        // Se ja esta ordenado ele para o bubble sort
        if (estaOrdenado)
            break;
    }
}
```

O código acima apresenta a implementação do método **Bubble Sort** para resolver o problema de ordenação dos brinquedos. Esse algoritmo percorre o array comparando pares de elementos adjacentes e trocando suas posições sempre que estão fora de ordem. Além disso, ele verifica se o array já está ordenado após percorrer toda a estrutura, interrompendo a execução caso esteja. A cada passagem completa pelo vetor, o maior elemento vai para o final, ocupando a posição correta.

A complexidade do Bubble Sort é $\mathbf{O(n^2)}$, pois, no pior caso, é necessário percorrer o vetor n vezes e realizar comparações entre n elementos em cada passagem. Por isso, o algoritmo não é recomendado para conjuntos de dados muito grandes.

2 Análise do Insertion Sort

```
void InsertionSort(LISTA *lista){
    int j;
    for (int i = 1; i < lista->tamanho; i++){

        // Pega uma chave fixa para comparar
        BRINQUEDO *chave = lista->elementos[i];
        j = i - 1;

        // Desloca todos os elementos maiores que a chave uma casa para a frente
        while (j >= 0 && PodeTrocar(lista->elementos[j], chave)){
            lista->elementos[j + 1] = lista->elementos[j];
            j--;
        }
```

```
14
15          // Insere o brinquedo chave na posicao correta ordenada
16          lista->elementos[j + 1] = chave;
17      }
18  }
```

O código acima se refere a implementação do método **Insertion Sort** para resolver o problema de ordenação. Esse algoritmo percorre o array inserindo cada elemento em sua posição correta em relação aos elementos anteriores. A cada iteração, o elemento chave é comparado com os anteriores e deslocado até que a ordem correta da chave seja alcançada. O processo se repete até que todo o vetor esteja ordenado.

A complexidade do Insertion Sort é $\mathbf{O(n^2)}$, pois cada elemento é comparado com todos os anteriores. No entanto, no melhor caso (quando o vetor já está ordenado), sua complexidade é O(n), o que o torna eficiente para pequenas quantidades de dados ou listas quase ordenadas.

3 Análise do Merge Sort

```
1   // Funcao auxiliar para o Merge Sort (Conquistar)
2   void Merge(LISTA *lista, int inicio, int meio, int fim, BRINQUEDO **auxiliar){
3
4       // Copia a lista original na lista auxiliar
5       for(int i = inicio; i <= fim; i++){
6           auxiliar[i] = lista->elementos[i];
7       }
8       int posLista1 = inicio;
9       int posLista2 = meio + 1;
10      int posListaOriginal = inicio;
11
12      // Junta as duas listas ordenadamente comparando os primeiros elementos de cada
13      while (posLista1 <= meio && posLista2 <= fim){
14          if (!PodeTrocar(auxiliar[posLista1], auxiliar[posLista2])){
15              lista->elementos[posListaOriginal] = auxiliar[posLista1];
16              posLista1++;
17          }
18          else{
19              lista->elementos[posListaOriginal] = auxiliar[posLista2];
20              posLista2++;
21          }
22          posListaOriginal++;
23      }
24
25      // Se sobrou elementos, adiciona na lista principal
26      while(posLista1 <= meio){
27          lista->elementos[posListaOriginal] = auxiliar[posLista1];
28          posListaOriginal++;
29          posLista1++;
30      }
31       while(posLista2 <= fim){
32          lista->elementos[posListaOriginal] = auxiliar[posLista2];
33          posListaOriginal++;
34          posLista2++;
35      }
36  }
37
38  void MergeSort(LISTA *lista, int inicio, int fim, BRINQUEDO **auxiliar){
39
40      // Condicao de parada da recursao
41      if (inicio < fim){
42          int meio = (int)(inicio + fim) / 2;
43          // Dividir
44          MergeSort(lista, inicio, meio, auxiliar);
45          MergeSort(lista, meio + 1, fim, auxiliar);
46
47          // Conquistar
48          Merge(lista, inicio, meio, fim, auxiliar);
49      }
50  }
```

O código acima mostra a implementação do método **Merge Sort** para resolver o problema de ordenação. Esse algoritmo utiliza a estratégia de **divisão e conquista**, dividindo recursivamente o array em partes menores usando um array auxiliar até que cada subarray contenha apenas um elemento. Em seguida, essas partes são combinadas em ordem crescente, formando progressivamente o vetor ordenado. Esse processo garante uma ordenação eficiente e estável.

A complexidade do Merge Sort é **O(n log n)** em todos os casos, pois o vetor é sempre dividido em duas metades e cada nível da recursão envolve o "merge"de todos os elementos. Apesar de exigir espaço adicional para as operações de mesclagem, ele é muito eficiente para grandes conjuntos de dados.

4 Análise do Quick Sort

```c
void QuickSort(LISTA *lista, int inicio, int fim){
    int meio = (inicio + fim) / 2;

    // Escolha do pivo
    BRINQUEDO *pivo = lista->elementos[meio];
    int i = inicio, j = fim - 1;

    // Faz comparacoes entre os elementos
    while (i <= j){
        while (PodeTrocar(pivo, lista->elementos[i]))
            i++;
        while (PodeTrocar(lista->elementos[j], pivo))
            j--;

        // Realiza a troca de elementos
        if (i <= j){
            BRINQUEDO *aux = lista->elementos[i];
            lista->elementos[i] = lista->elementos[j];
            lista->elementos[j] = aux;
            i++;
            j--;
        }
    }

    // Repete o processo recursivamente ate estar ordenado
    if (j > inicio)
        QuickSort(lista, inicio, j + 1);
    if (i < fim)
        QuickSort(lista, i, fim);
}
```

O código acima apresenta a implementação do método **Quick Sort** para resolver o problema de ordenação.. Esse algoritmo derivado do **Bubble Sort** também utiliza a estratégia de **divisão e conquista**, escolhendo um elemento como pivô e reorganizando o array de modo que todos os elementos menores que o pivô fiquem à sua esquerda, e os maiores, à direita. O processo é então repetido recursivamente para as duas partições até que todo o vetor esteja ordenado, resultando em uma ordenação eficiente na maioria dos casos. Os elementos comparados podem estar distantes um do outro no array, e conseguem ser trocados dessa maneira, por isso uma derivação do Bubble sort.

A complexidade do Quick Sort é **O(n log n)** em média quando se escolhe um bom pivô como a mediana de 3 elementos, mas pode chegar a $\mathbf{O(n^2)}$ no pior caso, quando o pivô escolhido é sempre o menor ou o maior elemento. Apesar disso, por ter boa performance prática e baixo uso de memória, é amplamente utilizado.

5 Função Auxiliar

```c
bool PodeTrocar(BRINQUEDO *item1, BRINQUEDO *item2){
    if (item1 && item2){

        // Verifica a ordem alfabetica da cor
        int cmp = strncmp(item1->cor, item2->cor, 10);
        if (cmp > 0)
```

```
            return true;
        else if (cmp < 0)
            return false;

        // Verifica o comprimento
        if (item1->comprimento > item2->comprimento)
            return true;
        else if (item1->comprimento < item2->comprimento)
            return false;

        // Verifica a nota
        if (item1->nota < item2->nota)
            return true;
        else if (item1->nota > item2->nota)
            return false;

        // Verifica o id do item
        if (item1->id > item2->id)
            return true;
        return false;
    }
    return false;
}
```

O código acima mostra a função auxiliar de todos os Sorts para realizar a comparação dos elementos, retornando true caso seja necessário a troca ou false caso já esteja ordenado.

Ele compara todas as características do brinquedo priorizando cor, comprimento, nota e índice respectivamente, dessa forma a comparação **garante a estabilidade** de todos os métodos de ordenação, até no Quick Sort que por natureza não é estável.

6 Comparação Entre Os Métodos

Ao comparar os métodos de ordenação Bubble Sort, Insertion Sort, Merge Sort e Quick Sort, o fator determinante é a complexidade de cada algoritmo, especialmente quando o tamanho dos dados n cresce. Para $n = 10^6$ (um milhão de elementos):

- **Bubble Sort e Insertion Sort** tornam-se inviáveis, pois o número de operações chega a 10^{12}, o que pode levar horas para concluir a ordenação.
- **Merge Sort e Quick Sort**, por outro lado, executam cerca de $10^6 \times log(10^6) \approx 10^6 \times 20 \approx 2 \times 10^7$ operações, o que é perfeitamente tratável em poucos segundos.

Algoritmo	Complexidade	n = 10^3	n = 10^6
Bubble Sort	$O(n^2)$	Rápido	Extremamente lento
Insertion Sort	$O(n^2)$	Rápido	Extremamente lento
Merge Sort	$O(n.log(n))$	Muito rápido	Muito Rápido
Quick Sort	$O(n.log(n))$	Muito rápid	Muito Rápido