Atividade 4 - Ordenação

Davi Gabriel Domingues - 15447497
Pedro Martins Oliveira - 13696213

7 de Outubro de 2025

1 Explicações dos Métodos

O algoritmo contido em "byron.c" lida com uma situação mais extensa dessa vez, já que deve tratar a ordenação das informações dos brinquedos (cor, comprimento, nota e id), a partir de quatro perspectivas, sendo elas, respectivamente enumeradas de 1 a 4: Insertion Sort, Bubble Sort, Merge Sort e Quick Sort. Para o auxílio da construção do programa final, foram utilizadas funções auxiliares de troca e de comparação dos dados solicitados/mencionados no documento, os quais embasaram os métodos de ordenação devidamente tratados.

Listing 1: Função `capitaoGinyu` — troca de structs

```
void capitaoGinyu (Brinquedo *x, Brinquedo *y) { /* referncia de Dragon Ball Z:
    "trocaaaaaar!", ou seja, realiza a troca
de variveis no vetor, via uso de variveis temporrias. */
    Brinquedo temporario = *x;
    *x = *y;
    *y = temporario;
}
```

Listing 2: Função `compararBrinquedos` — critério de ordenação

```
int compararBrinquedos(const Brinquedo *brinquedo1, const Brinquedo *brinquedo2) { /* a
    funo vai forar a estabilidade
em todos os mtodos de  ordenao . */
/* retorna valor maior que 0, caso o brinquedo1 deva estar  frente do brinquedo2, mas
    retorna valor menor que zero,
caso contrrio */
    int x = strcmp(brinquedo1->cor, brinquedo2->cor); /* funo de "string.h" para analisar
        se os nomes so iguais (0 = iguais,
    1 = primeira string maior, -1 = segunda string maior). Usa-se o critrio de comparao
        lexicogrfica para esse tamanho. */
    if (x != 0)
        return x;

    if (brinquedo1->comprimento > brinquedo2->comprimento)
        return 1;

    if (brinquedo1->comprimento < brinquedo2->comprimento)
        return -1;

    // critrio da nota   especfico : maior nota aparecer em primeiro na impresso final
        presente na main().
    if (brinquedo1->nota > brinquedo2->nota)
        return -1;

    if (brinquedo1->nota < brinquedo2->nota)
```

```
        return 1;

    // Desempate por id para garantir ordem estvel/determinstica (menor id == entrou
         antes)
    if (brinquedo1->id < brinquedo2->id)
        return -1;

    if (brinquedo1->id > brinquedo2->id)
        return 1;

    return 0; // significa que os dados so "equivalentes" para o algoritmo de ordenao
}
```

Sendo assim, tais metodologias de ordenação podem ter seus desempenhos lidados separadamente, dado às suas construções diferenciadas, sendo, então notório destacar os seguintes âmbitos nos algoritmos em si:

1.1 Insertion Sort

Tem seu funcionamento embasado no princípio da inserção do elemento — denominado chave — em uma posição apropriada para, justamente, ordenar o vetor. A cada iteração o algoritmo extrai essa chave (`brinquedo[i]`) do vetor e a insere na posição correta dentro do subvetor já ordenado $[0, i-1]$.

Em vez de procurar essa posição elemento a elemento, a rotina usa busca binária (com os índices inicio, fim, meio e a variável `posInsercao`) para decidir rapidamente em qual metade do subvetor a chave deve ficar; ao localizar `posInsercao` realiza-se um deslocamento prévio do bloco de elementos à direita dessa posição para abrir espaço e então a chave é escrita. O invariante útil para raciocinar sobre o algoritmo é: ao iniciar a iteração com índice i, o subvetor $[0, i-1]$ já está ordenado — essa propriedade é preservada a cada passo.

Listing 3: `ordenacaoViaInsertionSort` (inserção com busca binária)

```
void ordenacaoViaInsertionSort(Brinquedo *brinquedo, int numeroBrinquedos) { /* foi
     escolhida a insero binria, por conta
de seu desempenho amortizado, comparado  insero direta tradicional */
    for (int i = 1; i < numeroBrinquedos; i++) {
        Brinquedo chave = brinquedo[i];
        int j = i - 1;

        // encontra a  posio  onde a 'chave' deve ser inserida no subarray chave[0...i-1]
        int inicio = 0, fim = j, posInsercao = i;

        while (inicio <= fim) {
            int meio = inicio + (fim - inicio) / 2;

            if (compararBrinquedos(&chave, &brinquedo[meio]) < 0) {
                posInsercao = meio;
                fim = meio - 1;
            }

            else
                inicio = meio + 1;
        }
```

```
        // deslocamento dos elementos
        for (j = i - 1; j >= posInsercao; j--) {
            brinquedo[j + 1] = brinquedo[j];
        }

        brinquedo[posInsercao] = chave;
    }
}
```

Quanto ao desempenho, percebe-se que a busca binária reduz o número de comparações necessárias para localizar a posição de inserção — para cada elemento a busca custa $O(\log i)$, resultando em $O(n \log n)$ comparações no total. Contudo, após encontrar `posInsercao` ainda é preciso deslocar os elementos do subvetor para a direita, e esses deslocamentos custam linearmente no tamanho do bloco deslocado.

Somando todos os deslocamentos ao longo do algoritmo o custo de movimentação é $O(n^2)$. Assim, embora haja uma melhoria nas comparações graças ao comportamento logarítmico da busca, o tempo total do algoritmo permanece $O(n^2)$ devido às movimentações.

1.2 Bubble Sort

Tem seu funcionamento baseado no princípio de varreduras sucessivas por pares adjacentes: o algoritmo percorre o vetor comparando vizinhos e os trocando quando estão na ordem errada, de modo que valores maiores "sobem" em cada varredura e valores menores "descem". No código foi adotada a variação chamada *Shake Sort* (ou *Cocktail Shaker Sort*), que alterna varreduras da esquerda para a direita e da direita para a esquerda usando as variáveis `inicio`, `fim` e a *flag* `trocou`.

Na varredura da esquerda para a direita o maior elemento do intervalo atual é empurrado até `fim`, na varredura da direita para a esquerda o menor elemento é empurrado até `inicio`. Após cada par de varreduras, os limites `inicio` e `fim` são atualizados, reduzindo o intervalo ativo. O invariante que facilita o raciocínio é: ao término de cada varredura completa (ida e volta), todos os elementos fora do intervalo $[\text{inicio}, \text{fim}]$ já estão nas posições finais corretas.

Listing 4: `ordenacaoViaBubbleSort` (Shake Sort)

```
void ordenacaoViaBubbleSort(Brinquedo *brinquedo, int numeroBrinquedos) {
    if (numeroBrinquedos <= 1 || brinquedo == NULL)
        return;

    int inicio = 0, fim = numeroBrinquedos - 1;
    bool trocou = true;

    while (trocou) {
        trocou = false;

        // varredura ocorrer da esquerda para a direita (move o maior elemento para o fim)
        for (int i = inicio; i < fim; i++) {
            if (compararBrinquedos(&brinquedo[i], &brinquedo[i+1]) > 0) {
                capitaoGinyu(&brinquedo[i], &brinquedo[i+1]);
                trocou = true;
            }
        }

        if (!trocou)
```

```
            break;

        // diminui o fim, pois o maior elemento j est na sua  posio  correta
        fim--;

        // Varredura da direita para a esquerda (move o menor elemento para o incio)
        for (int i = fim - 1; i >= inicio; i--) {
            if (compararBrinquedos(&brinquedo[i], &brinquedo[i+1]) > 0) {
                capitaoGinyu(&brinquedo[i], &brinquedo[i+1]);
                trocou = 1;
            }
        }

        // aumenta o incio, pois o menor elemento j  est  na sua  posio  correta
        inicio++;
    }
}
```

Quanto aos impactos e ao desempenho, o *Shake Sort* tenta amortizar o comportamento do *Bubble Sort* clássico ao mover os extremos em ambas as direções, o que pode reduzir o número de passadas necessárias em entradas onde grandes valores estão no começo e pequenos valores no fim. Além disso, a flag `trocou` torna o algoritmo adaptativo — se uma passada não produzir nenhuma troca, o algoritmo interrompe cedo, alcançando custo $O(n)$ no melhor caso (vetor já ordenado). Ainda assim, no pior caso (vetor inversamente ordenado) tanto comparações quanto trocas continuam na ordem de $O(n^2)$, porque cada elemento ainda pode precisar ser trocado muitas vezes.

Em termos práticos, o *Shake Sort* frequentemente reduz constantes (menos passadas efetivas) e pode ser mais rápido que o *Bubble Sort* em alguns padrões de dados, mas não altera a classe de complexidade assintótica no pior caso. Uma das propriedades adicionais relevantes ao algoritmo está relacionada com o fato dele ser *in-place* (uso de memória extra $O(1)$).

1.3 Merge Sort

Tem seu funcionamento baseado no princípio divide-and-conquer: o algoritmo divide recursivamente o vetor em metades até obter subvetores triviais (tamanho 0 ou 1) e então reconstrói o vetor ordenado juntando (merge) pares de subvetores já ordenados. Na sua implementação essa etapa de junta é feita alocando temporariamente leftArr e rightArr no heap e copiando os blocos correspondentes antes de mesclar — essa escolha evita VLA / estouro de pilha e simplifica a implementação. É um método, portanto, de ordenação estável e baseado na divisão do conjunto de dados em partes menores para facilitar a ordenação.

Podemos visualizar o funcionamento da ordenação pelo seguinte passo a passo: Primeiro o conjunto de dados é divido recursivamente na metade, até que tenha sobrado apenas 1 elemento.

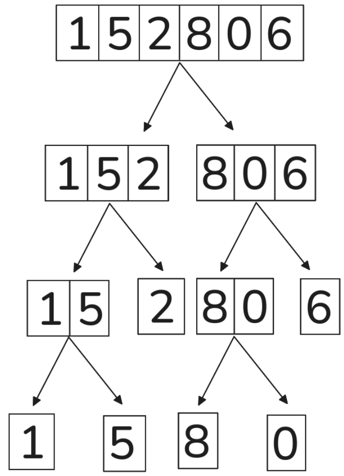

Figure 1: Separação dos elementos. Autoria Própria.

Em seguida, fazemos o caminho inverso, ou seja, juntamos os elementos novamente porém em cada junção ordenamos eles. Esse processo de ordenação é simplificado pois como sabemos que os dados em cada conjunto estão ordenados, podemos iterar pelos dois conjuntos que estamos juntando apenas colocando o menor valor em cada iteração. Fazemos isso até juntar todos os elementos, obtendo assim o conjunto ordenado.

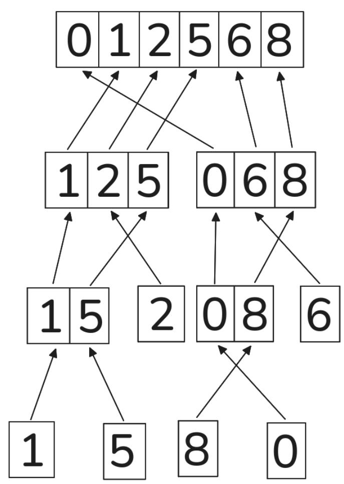

Figure 2: Junção dos elementos. Autoria Própria

No código, a implementação seria esta:

Listing 5: Função `junta` (merge helper)e ordenacaoViaMergeSort

```
// Usar heap em vez de VLA para evitar estouro de pilha em entradas grandes
Brinquedo *leftArr = NULL;
Brinquedo *rightArr = NULL;

if (n1 > 0) {
```

```c
        leftArr = (Brinquedo*)malloc(n1 * sizeof(Brinquedo));
        if (!leftArr)
            return; // falha em alocar -> evita comportamento indefinido
    }

    if (n2 > 0) {
        rightArr = (Brinquedo*)malloc(n2 * sizeof(Brinquedo));

        if (!rightArr) {
            free(leftArr);
            return;
        }
    }

    for (i = 0; i < n1; i++)
        leftArr[i] = brinquedos[inicio + i];

    for (j = 0; j < n2; j++)
        rightArr[j] = brinquedos[meio + 1 + j];

    i = 0;
    j = 0;
    k = inicio;

    while (i < n1 && j < n2) {
        if (compararBrinquedos(&leftArr[i], &rightArr[j]) <= 0) {
            brinquedos[k] = leftArr[i];
            i++;
        }
        else {
            brinquedos[k] = rightArr[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        brinquedos[k] = leftArr[i];
        i++;
        k++;
    }

    while (j < n2) {
        brinquedos[k] = rightArr[j];
        j++;
        k++;
    }

    free(leftArr);
    free(rightArr);
}

void ordenacaoViaMergeSort(Brinquedo *brinquedos, int inicio, int fim) {
    if(inicio >= fim) return; // Condio de  sada

    int meio = inicio + (fim - inicio)/2;
```

```
    ordenacaoViaMergeSort(brinquedos, inicio, meio);
    ordenacaoViaMergeSort(brinquedos, meio + 1, fim);

    junta(brinquedos, inicio, meio, fim);

    return;
}
```

O invariante que facilita o raciocínio, a partir do arquivo `byron.c`: ao entrar na função `junta(inicio, meio, fim)` os intervalos `inicio..meio` e `meio+1..fim` estão, cada um, ordenados; ao sair de `junta` o intervalo `inicio..fim` está ordenado. Na rotina de *merge* do código a condição "`if compararBrinquedos(&leftArr[i], &rightArr[j]) <= 0)`" privilegia o elemento da esquerda em empate — essa escolha garante que a implementação seja *estável*, preservando desempates por `id` definidos em `compararBrinquedos`. Quanto aos impactos e ao desempenho: a recorrência é

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n),$$

logo tem complexidade $\mathcal{O}(n \log n)$ no melhor, no médio e no pior caso, com espaço extra $\mathcal{O}(n)$ (além da pilha recursiva $\mathcal{O}(\log n)$). Em termos práticos, a boa localidade (leituras/escritas sequenciais) favorece cache e I/O — o que costuma refletir em tempos estáveis, uma vez que a complexidade assintótica não varia entre os casos (isto é, não há melhor/pior caso assintótico distinto para o Merge Sort).

1.4 Quick Sort

Tem seu funcionamento baseado em escolher um elemento pivô, particionar o vetor em dois subintervalos (elementos ¡= pivô à esquerda e ¿ pivô à direita) e então aplicar recursivamente o procedimento a cada subintervalo, no nosso caso escolhemos o último elemento, então iteramos nesses intervalos, inserindo todos os valores/elementos menores antes do pivô e todos os maiores depois do mesmo. No código do arquivo "byron.c" a partição escolhe o último elemento como pivô e realiza trocas em-loco com capitaoGinyu, movendo structs inteiros para reorganizar o array — abordagem simples e in-place, mas que copia Brinquedo nas trocas e, portanto, pode ficar cara quando a struct é grande. Ou seja, assim como o *Merge Sort*, o *Quick Sort* também é baseado na divisão do conjunto em partes menores.

Podemos ver a lógica de funcionamento pelo exemplo simplificado a seguir:

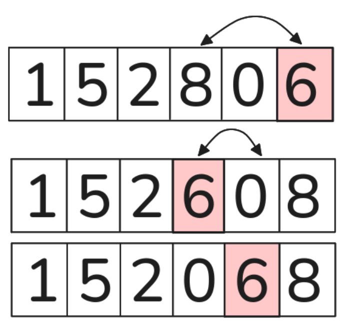

Figure 3: Primeira Iteração do *Quick Sort*. Autoria Própria

Logo, percebe-se que a função é chamada recursivamente para o conjunto à esquerda eÀa direita do pivô, até que, novamente, só reste 1 elemento.

A lógica aplicada no programa final segue a seguinte estruturação prática:

Listing 6: Função `particiona` (particionamento do QuickSort) e ordenacaoViaQuickSort

```c
int particiona(Brinquedo *brinquedo, int inicio, int fim) {
    //  funo  de particionamento para o QuickSort
    // - escolhe o ltimo elemento como pivo (estratgia simples e determinstica)
    // - reorganiza o subarray de modo que todos elementos <= pivo fiquem  esquerda,
    //   e os > pivo fiquem  direita, mantendo estabilidade relativa onde possvel.
    // - retorna o ndice final do pivo aps a  partio .

    Brinquedo pivo = brinquedo[fim];
    int i = inicio - 1;

    for (int j = inicio; j < fim; j++) {
        // se brinquedo[j]  menor ou igual ao pivo segundo nosso critrio,
        // incrementa i e troca para mover esse elemento para a "parte menor".
        if (compararBrinquedos(&brinquedo[j], &pivo) <= 0) {
            i++;
            capitaoGinyu(&brinquedo[i], &brinquedo[j]); // troca estvel via struct completo
        }
    }
    // coloca o pivo na sua posio ordenada final
    capitaoGinyu(&brinquedo[i + 1], &brinquedo[fim]);
    return i + 1;
}

void ordenacaoViaQuickSort(Brinquedo *brinquedo, int inicio, int fim) {
    //  Implementao  recursiva do QuickSort:
    // - chama particiona para dividir o array em duas partes em torno do pivo
    // - aplica recursivamente nas subpartes esquerda e direita
    // -  condio  de parada: subarray com zero ou um elemento (inicio >= fim)
    if (inicio < fim) {
        int pi = particiona(brinquedo, inicio, fim);
        ordenacaoViaQuickSort(brinquedo, inicio, pi - 1);
        ordenacaoViaQuickSort(brinquedo, pi + 1, fim);
    }
}
```

O invariante a acompanhar durante a partição, no programa final ”byron.c”, é: ao processar o índice j, todos os elementos até i são ¡= pivô, os elementos entre i+1 e j-1 são ¿ pivô, e os elementos de j em diante ainda não foram processados; ao fim do loop o pivô é trocado para i+1 e as duas subpartes ficam prontas para serem recursadas. Por usar trocas em-loco, a implementação padrão não é estável — elementos iguais segundo compararBrinquedos podem ter sua ordem relativa alterada. Entretanto, dado que compararBrinquedos finalmente desempata por id (definindo uma ordem total), a não-estabilidade não compromete a determinismo da ordenação.

Dessa maneira, apesar de muito eficiente, tal método possui um problema quando o conjunto já está ordenado, pois, nesse caso, o pivô não irá mudar de posição, fazendo com que tenhamos uma quantidade muito grande de chamadas recursivas e nenhuma delas fará ativamente nenhuma troca, isso faz com que a complexidade do algoritmo nesse caso seja de $O(n^2)$. Ainda assim, em outros casos não temos esse problema, logo o pior e o caso médio para esse algoritmo possuem complexidade $O(n \log n)$, uma vez que, na prática, o QuickSort costuma ter constantes menores e um excelente uso de cache, o tornando rápido em muitos casos de input. Nesse cenário, percebe-se que as partições são, de certa forma, caracteristicamente balanceadas para o algoritmo de ordenação em si.

2 Comparação dos Métodos

Uma boa maneira de avaliar as vantagens e desvantagens de cada algoritmo é comparar o tempo de execução deles, sendo que utilizaremos 3 conjuntos de valores de entrada diferentes, nas situações a seguir:

- Melhor caso: conjunto já ordenado.
- Caso Médio: conjunto em ordem aleatória.
- Pior caso: conjunto ordenado inversamente.

Esses tipos de dados, assim como os seus contextos de aplicação como entrada para cada um dos algoritmos, foram aplicados e suas quantidades foram devidamente variadas, sendo que os resultados obtidos foram colocados na tabela abaixo (tabela 2):

Método	70.000	700.000	7000.000	70 000.000
		Melhor caso		
Bubble	2.160	8.090	72.530	541.630
Insertion	1.000	1.000	3.170	18.340
Merge	2.500	1.090	1.000	1.000
Quick	7.670	26.810	192.500	1354.440
		Caso médio		
Bubble	2.790	15.120	144.150	1414.460
Insertion	1.000	2.480	17.240	153.820
Merge	1.710	1.050	1.000	1.070
Quick	1.210	1.000	1.200	1.000
		Pior caso		
Bubble	2.850	25.560	258.170	2117.650
Merge	1.310	1.000	1.000	1.000
Insertion	1.000	6.080	54.620	432.940
Quick	2.620	18.460	135.380	1070.590

Table 1: Tempos relativos por algoritmo e quantidade de itens

Como conteúdo complementar, temos as execuções apuradas no runcodes, tendo seus rendimentos expostos a seguir:

Caso	Bubble	Insertion	Merge	Quick
1.	0.001	0.000	0.000	0.000
2.	7.400	0.845	0.030	0.012
3.	x	x	0.209	x
4.	304.540	40.376	0.172	0.134
5.	x	x	x	0.168

Table 2: Tempos por algoritmo e arquivo de teste, medidos em segundos

Obs: o ”x” indica que não houve como realizar a medição de tempo, devido à inexistência do input do arquivo de teste associado a cada caso de execução do runcodes.