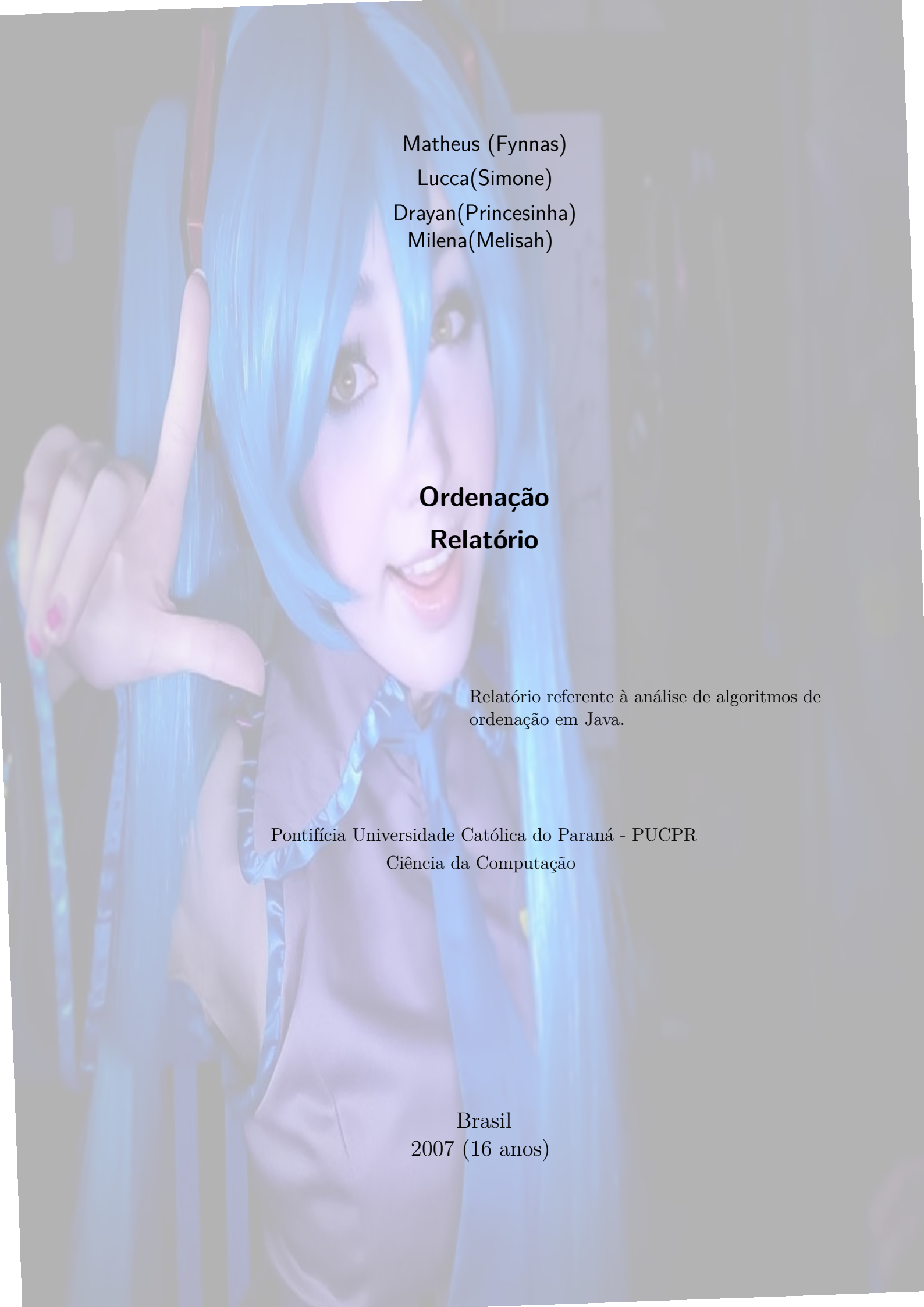


Matheus (Fynnas)
Lucca(Simone)
Drayan(Princesinha)
Milena(Melisah)

Ordenação Relatório

Brasil
2007 (16 anos)



Matheus (Fynnas)
Lucca(Simone)
Drayan(Princesinha)
Milena(Melisah)

Ordenação Relatório

Relatório referente à análise de algoritmos de ordenação em Java.

Pontifícia Universidade Católica do Paraná - PUCPR
Ciência da Computação

Brasil
2007 (16 anos)

1 Introdução

Implementação de algoritmos de ordenação. Comparando e analisando seu desempenho com vetores de inteiros com 100, 500, 1000 e 5000 e 10.000 elementos para a matéria de Resolução de Problemas Estruturados em Computação.

2 Implementação

2.1 Insert Sort (Chârù Sort)

Percorre a lista da esquerda para a direita, inserindo cada elemento na posição correta, de acordo com o valor.

```
public class ChârùSort {  
    1 usage  🧑 Matheus Gabriel Pereira Nogueira +1  
    public static void ChârùSort(int[] ararray, int Tamanyo, int[] Contages) {  
  
        for (int i = 1; i < Tamanyo; i++) {  
            int espaco_de_trabalho = ararray[i];  
            int j = i - 1;  
            Contages[1]++;  
            while (j >= 0 && ararray[j] > espaco_de_trabalho) {  
                ararray[j + 1] = ararray[j];  
                j--;  
                Contages[0]++;  
            }  
            ararray[j + 1] = espaco_de_trabalho;  
        }  
    }  
}
```

2.2 Selection Sort (A-Selecao-Sort)

Encontra o menor elemento na lista e o coloca na primeira posição, repetindo o processo para o restante da lista.

```
public class A_Selecao_Sort {  
  
    1 usage  🧑 Matheus Gabriel Pereira Nogueira  
    public static void FlamingoSort(int[] array, int Tamanho, int[] Contages) {  
  
        for (int i = 0; i < Tamanho - 1; i++) {  
            int D_Menoh = i;  
            for (int j = i + 1; j < Tamanho; j++) {  
                Contages[1]++;  
                if (array[j] < array[D_Menoh]) {  
                    D_Menoh = j;  
                }  
            }  
            int Ichiji_teki = array[D_Menoh];  
            array[D_Menoh] = array[i];  
            array[i] = Ichiji_teki;  
            Contages[0]++;  
        }  
    }  
}
```


2.3 BubbleSort (AwaSort)

Compara pares de elementos adjacentes e os troca se estiverem fora de ordem, repetindo esse processo até que a lista esteja ordenada.

```
2 usages  🧑 Matheus Gabriel Pereira Nogueira +1
public class AwaSort {
    1 usage  🧑 Matheus Gabriel Pereira Nogueira +1
    public static void AwaSort(int[] ararray, int Tamanico, int[] Contages) {

        for (int i = 0; i < Tamanico - 1; i++) {
            for (int j = 0; j < Tamanico - i - 1; j++) {
                Contages[1]++;
                if (ararray[j] > ararray[j + 1]) {
                    int Ichiji_teki = ararray[j];
                    ararray[j] = ararray[j + 1];
                    ararray[j + 1] = Ichiji_teki;
                    Contages[0]++;
                }
            }
        }
    }
}
```

2.4 Merge Sort (MergianaSort)

Divide a lista em partes menores, ordena cada parte e depois mescla as partes ordenadas para obter a lista final ordenada.

```
2 usages  📌 Matheus Gabriel Pereira Nogueira
public class MergianaSort {
    1 usage  📌 Matheus Gabriel Pereira Nogueira
    public static void Morganah(int[] array, int Tamanho, int[] Contages) {

        MorganahSort(array, Mais_Pra_kah: 0, Mais_Pra_lah: Tamanho - 1, Contages);
    }

    // mais pra kah = esquerda
    // mais pra lah = direita
    3 usages  📌 Matheus Gabriel Pereira Nogueira
    private static void MorganahSort(int[] array, int Mais_Pra_kah, int Mais_Pra_lah, int[] Contages) { // separa os moco
        if (Mais_Pra_kah < Mais_Pra_lah) {
            int Meiukah = (Mais_Pra_kah + Mais_Pra_lah) / 2;
            MorganahSort(array, Mais_Pra_kah, Meiukah, Contages);
            MorganahSort(array, Mais_Pra_kah: Meiukah + 1, Mais_Pra_lah, Contages);
            Mergeh(array, Mais_Pra_kah, Meiukah, Mais_Pra_lah, Contages);
        }
    }
}
```

```
1 usage  📌 Matheus Gabriel Pereira Nogueira
private static void Mergeh(int[] array, int Mais_Pra_kah, int Meiukah, int Mais_Pra_lah, int[] Contages) { // junta os moco
    int arara1 = Meiukah - Mais_Pra_kah + 1; // vê uma metade
    int arara2 = Mais_Pra_lah - Meiukah; // vê a outra metade

    int[] ararayDum = new int[arara1]; // cria os array de um e outro
    int[] ararayDotro = new int[arara2];

    for (int i = 0; i < arara1; i++) { // add nos bff
        ararayDum[i] = array[Mais_Pra_kah + i];
    }
    for (int j = 0; j < arara2; j++) {
        ararayDotro[j] = array[Meiukah + 1 + j];
    }

    int Comp1 = 0, Comp2 = 0;
    int poze_atual = Mais_Pra_kah;
    while (Comp1 < arara1 && Comp2 < arara2) {
        Contages[1]++;

        if (ararayDum[Comp1] <= ararayDotro[Comp2]) {
            array[poze_atual] = ararayDum[Comp1];
            Comp1++;
        } else {
            array[poze_atual] = ararayDotro[Comp2];
            Comp2++;
        }
        poze_atual++;
    }
}
```

```
while (Comp1 < arara1) {  
    arary[poze_atual] = ararayDum[Comp1];  
    Comp1++;  
    poze_atual++;  
}  
  
while (Comp2 < arara2) {  
    arary[poze_atual] = ararayDotro[Comp2];  
    Comp2++;  
    poze_atual++;  
}  
}
```

2.5 Quick Sort (QuicksilverSort)

Divide a lista em torno de um elemento pivô, colocando elementos menores à esquerda e elementos maiores à direita. Repete o processo nas sublistas.

```
public class QuicksilverSort {  
  
    3 usages  📌 Matheus Gabriel Pereira Nogueira  
    public static void Qicksilver(int[] ararray, int Menoh_doPedaco, int Maioh_doPedaco, int[] Contages) {  
  
        if (Menoh_doPedaco < Maioh_doPedaco) {  
            int Pivet = particionah(ararray, Menoh_doPedaco, Maioh_doPedaco, Contages);  
            Qicksilver(ararray, Menoh_doPedaco, Maioh_doPedaco: Pivet - 1, Contages);  
            Qicksilver(ararray, Menoh_doPedaco: Pivet + 1, Maioh_doPedaco, Contages);  
        }  
    }  
}
```

```
private static int particionah(int[] ararray, int Menoh_doPedaco, int Maioh_doPedaco, int[] Contages) {  
    int Pivet = ararray[Maioh_doPedaco];  
    int i = (Menoh_doPedaco - 1);  
    for (int j = Menoh_doPedaco; j < Maioh_doPedaco; j++) {  
        Contages[1]++; // iteracao  
        if (ararray[j] < Pivet) {  
            i++;  
            Contages[0]++; // troca  
            int Ichiji_teki = ararray[j];  
            ararray[j] = ararray[i];  
            ararray[i] = Ichiji_teki;  
        }  
    }  
    Contages[0]++;  
    int Ichiji_teki = ararray[i + 1];  
    ararray[i + 1] = ararray[Maioh_doPedaco];  
    ararray[Maioh_doPedaco] = Ichiji_teki;  
    return i + 1;  
}
```


2.6 Shell Sort (ShelldoSort)

Parecido com o Insertion Sort, mas divide a lista em subgrupos menores e aplica o Insertion Sort a cada subgrupo.

```
public class ShelldoSort {  
    1 usage  📌 Matheus Gabriel Pereira Nogueira  
    public static void shellSort(int array[], int Tamanho_do_cabra, int[] Contages) {  
  
        int caba da troca, posicao de pulo;  
        // defini quantos pulos vai ter ta dano  
        int saltados = Tamanho_do_cabra / 2;  
  
        // enquanto o tamanho do pulo dao for maior q 0  
        while (saltados > 0) {  
            for (int i = saltados; i < Tamanho_do_cabra; i++) {  
                caba da troca = array[i];  
                posicao de pulo = i; // verificando onde o elemento pode entrar  
                Contages[1]++;  
                while (posicao de pulo >= saltados && array[posicao de pulo - saltados] > caba da troca) {  
                    array[posicao de pulo] = array[posicao de pulo - saltados];  
                    posicao de pulo = posicao de pulo - saltados;  
                    Contages[0]++;  
                }  
                array[posicao de pulo] = caba da troca;  
            }  
            saltados = saltados / 2;  
        }  
    }  
}
```

2.7 Heap Sort (RipBoSort)

Transforma a lista em uma estrutura de monte e então remove repetidamente o elemento máximo (ou mínimo) do heap, obtendo uma lista ordenada.

```
public class RipBoSort {  
  
    1 usage  Matheus Gabriel Pereira Nogueira  
    public static void HeepySort(int ararray[], int Tamanho, int[] Contages) {  
        for (int i = Tamanho / 2 - 1; i >= 0; i--) {  
            hepysort(ararray, Tamanho, i, Contages);  
        }  
  
        for (int j = Tamanho - 1; j > 0; j--) {  
            int temp = ararray[0];  
            ararray[0] = ararray[j];  
            ararray[j] = temp;  
            Contages[0]++;  
            hepysort(ararray, j, i: 0, Contages);  
        }  
        System.out.println(Contages[0] + " trocas, " + Contages[1] + " iteracoes.");  
    }  
}
```

```
3 usages  Matheus Gabriel Pereira Nogueira  
public static void hepysort(int[] ararray, int n, int i, int[] Contages) {  
    int raize = i;  
    int filho_da_esquerda = 2 * i + 1;  
    int filho_da_direita = 2 * i + 2;  
    Contages[1]++;  
  
    if (filho_da_esquerda < n && ararray[filho_da_esquerda] > ararray[raize]) {  
        raize = filho_da_esquerda;  
    }  
    if (filho_da_direita < n && ararray[filho_da_direita] > ararray[raize]) {  
        raize = filho_da_direita;  
    }  
  
    if (raize != i) {  
        int temp = ararray[i];  
        ararray[i] = ararray[raize];  
        ararray[raize] = temp;  
        Contages[0]++;  
        hepysort(ararray, n, raize, Contages);  
    }  
}
```

3 Análise de desempenho

* Tempo medido em milisegundos *

Função	Tamanho	Tempo Médio
Insertion Sort	50	0,1312
	500	1,29178
	1000	3,42921
	5000	16,28803
	10000	42,32091
Selection Sort	50	0,1155
	500	1,37958
	1000	4,67411
	5000	23,30608
	10000	97,27565
Bubble Sort	50	0,1303
	500	2,25613
	1000	4,81854
	5000	40,54539
	10000	173,92083
Merge Sort	50	0,1168
	500	0,3890
	1000	1,08998
	5000	2,82461
	10000	2,57610
quick Sort	50	0,1694
	500	0,5046
	1000	0,3572
	5000	0,9154
	10000	1,54325
shell Sort	50	0,079
	500	0,4453
	1000	0,9362
	5000	2,60354
	10000	3,04853
heap Sort	50	1,89711
	500	2,06862
	1000	1,99560
	5000	3,05371
	10000	5,47580

Função		media trocas	media interações
Insertion Sort	50	614	49
	500	61636	499
	1000	245812	999
	5000	6243023	4999
	10000	24829164	9999
Selection Sort	50	49	1225
	500	499	124750
	1000	999	499500
	5000	4999	12497500
	10000	9999	49995000
Bubble Sort	50	566	1225
	500	62940	124750
	1000	249454	499500
	5000	6247814	12497500
	10000	25049432	49995000
Merge Sort	50	0	222
	500	0	3863
	1000	0	8710
	5000	0	55233
	10000	0	120482
quick Sort	50	147	255
	500	2565	4775
	1000	5964	10635
	5000	35219	69592
	10000	84465	158774
shell Sort	50	127	203
	500	2892	3506
	1000	7350	8006
	5000	61275	55005
	10000	144435	120005
heap Sort	50	237	262
	500	4040	4290
	1000	9083	9583
	5000	57110	59610
	10000	124164	129164

4 Conclusão

O "Merge Sort", "Quick Sort", "Shell Sort" e "Heap Sort" mostram desempenhos muito mais eficientes em termos de média de trocas e interações, mesmo com tamanhos maiores de arrays. Isso os torna escolhas melhores para ordenação de dados grandes.

Ao analisar o tempo médio de execução, observamos que o "Insertion Sort", "Selection Sort" e "Bubble Sort" têm tempos significativamente maiores à medida que o tamanho do array aumenta. Os algoritmos mais eficientes em termos de tempo médio de execução são o "Merge Sort", o "Quick Sort" e o "Shell Sort", especialmente para conjuntos de dados maiores.

REFERÊNCIAS

- [1] CANAVESE, Filipe Germano. O Testamento de Dona Balbina: um estudo de caso sobre escravidão e propriedade em Guarapuava (1851-1865). Assis, SP, 2011.. Disponível em: <https://galeriaderacistas.com.br/visconde-de-guarapuava/>.