UFMA - Universidade Federal do Maranhão
São Luís, 14 de abril de 2024
Dupla: Maxwell Pires Silva e Grace Kelly Barros Araujo dos Santos

Estrutura de Dados II

Relatório - Trabalho de Implementação I

Introdução

O presente relatório visa demonstrar a lógica e os códigos utilizados para resolver as questões propostas no trabalho de implementação de algoritmos de ordenação, bem como explicar como os resultados foram alcançados com os algoritmos desenvolvidos. O trabalho foi dividido em pacotes. No pacote *Algorithms* temos a classe de todos algoritmos de ordenação necessários para resolver as questões. No pacote *Questoes* estão as classes de cada questão e por fim a Main onde o código deve ser rodado. Na Main está presente um menu de escolha para a rápida análise de cada questão individualmente.

Objetivos

Implementar os algoritmos de ordenação apresentados em sala de aula, junto de otimizações instruídas no enunciado das questões do trabalho.

Problema 1

Foi implementada uma versão do MergeSort com corte para sub-vetores pequenos, de tamanho menor ou igual a 15. Nesse caso, é feita a chamada da função InsertSort, responsável por ordenar os vetores menores, melhorando o tempo de execução em cerca de 10 a 15%.

O MergeSort é um algoritmo de divisão e conquista, com a etapa merge() do algoritmo responsável por conquistar os vetores divididos em etapas anteriores. Para aumentar a eficiência em tempo de execução, foi adicionado uma verificação para ignorar a chamada merge() caso o vetor já esteja ordenado.

```
Selecione uma opção: 1
Vetor original:
{ 0 : 424 }
{ 1 : 653 }
{ 2 : 888 }
{ 3 : 526 }
{ 4 : 707 }
{ 5 : 340 }
{ 6 : 185 }
{ 7 : 23 }
{ 8 : 458 }
{ 9 : 40 }
{ 10 : 863 }
{ 11 : 323 }
{ 12 : 886 }
{ 13 : 39 }
{ 14 : 902 }
```

```
Vetor após MergeSort modificado:
{ 7 : 23 }
{ 13 : 39 }
{ 9 : 40 }
{ 6 : 185 }
{ 11 : 323 }
{ 5 : 340 }
{ 0 : 424 }
{ 8 : 458 }
{ 3 : 526 }
{ 1 : 653 }
{ 4 : 707 }
{ 10 : 863 }
{ 12 : 886 }
{ 2 : 888 }
{ 14 : 902 }
```

Problema 2

Foi implementada uma versão modificada do algoritmo SelectSort. Originalmente, ele organiza o vetor procurando um valor menor que um determinado valor utilizado como referência, sendo esse o primeiro elemento do vetor. Ao encontrar, ele troca os elementos de lugar, colocando cada um em sua posição correta, ordenando o vetor de forma crescente. Na modificação implementada, fizemos com que o vetor fosse ordenado procurando tanto os menores quanto os maiores elementos e realizando trocas para garantir a ordenação correta a cada iteração.

```
Vetor original:
[ 502 234 934 809 206 366 480 47 753 540 924 174 913 749 583 ]

Vetor após SelectSort modificado:
[ 47 174 206 234 366 480 502 540 583 749 753 809 913 924 934 ]
```

Questão 03

Nesta questão devemos implementar um Algoritmo de QuickSort, porém, com o pivô sendo a mediana de 3 elementos (inicio, meio e fim). Além disso, devemos encontrar uma valor L para qual o tamanho da partição do QuickSort vai parar e passar a ordenar essa participação de tamanho menor ou igual a L com o Algoritmo de BubbleSort. Assim, realizamos a implantação do BubbleSort, mas agora recebendo o índice de início e o índice de fim, para delimitar onde o Bubble vai agir. Foram implementados, também, os Algoritmos do Quick com a mediana de 3 e também, com a verificação do tamanho da partição para chamar o BubbleSort. Abaixo as funções padrão, com alterações e utilizando a função *quick_mediana* para encontrar a mediana.

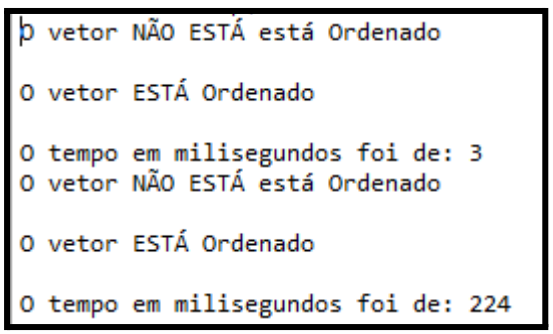
```java
public int particiona_vetor(T[] array, int inicio, int fim) {
        quick_mediana(array, inicio, fim);
        T pivo = array[inicio];

        int i = inicio+1; int j = fim;

        while(i <= j) {
            if(pivo.compareTo(array[i]) >= 0) {
                i++;
            } else if(array[j].compareTo(pivo) > 0) {
                j--;
            } else {
                T aux = array[i];
                array[i] = array[j];
                array[j] = aux;
                i++;
                j--;
            }
        }
        array[inicio] = array[j];
        array[j] = pivo;
        return j;
    }
```

```java
public class QuickSort<T extends Comparable<T>> {

    BubbleSort<T> bubble = new BubbleSort<>();

    public void quick_sort(T[] array, int inicio, int fim, int L) {
        if(inicio < fim) {
            if(fim-inicio > L) {
                int posicao_pivo = particiona_vetor(array, inicio, fim);
                quick_sort(array, inicio, posicao_pivo-1, L);
                quick_sort(array, posicao_pivo+1, fim, L);
            } else {
                bubble.buble_sort(array, inicio, fim);
            }
        }
    }
```

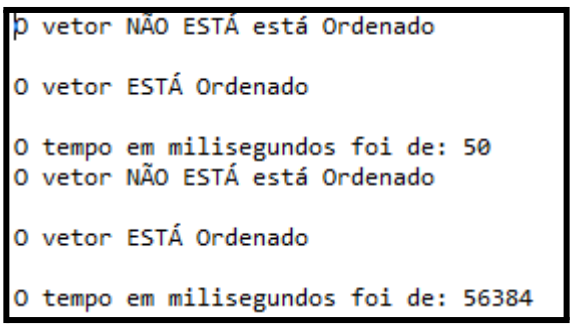
```java
public void quick_mediana(T[] array, int inicio, int fim) {
        int maior = inicio;

        if(array[fim].compareTo(array[maior]) > 0) {
            maior = fim;
        } else if(array[(inicio+fim)/2].compareTo(array[maior]) > 0){
            maior = (inicio+fim)/2;
        }

        T aux = array[inicio];
        array[inicio] = array[maior];
        array[maior] = aux;
    }
```

Resposta:
Devido à quantidade de elementos que deve-se testar para obter o valor de L, torna-se dificultoso colocar imagens da resposta com os vetores impressos. Então utilizaremos, somente, aqui, a imagem com a informação se o vetor está ou não ordenado.
Obviamente, a resposta que se tem de imediato é L = 0, correto? Uma vez que L = 0, todo o vetor é ordenado apenas com Bubble Sort. Empiricamente é o que se nota. Porém, fizemos estudo para valores de L maiores que 0. Como resultado L = n/2 (L igual ao número de elementos dividido por 2) se saiu melhor em todos os testes realizados. Abaixo, um teste com 10 mil elementos.

Teste com L = 0 / L = 500 (1000 elementos)



Teste com L = 0/ L=50.000 (100.000 elementos)



Agora, comparando em números absolutos, para 10.000 elementos, L = 5000 foi melhor do que L = 4000(imagem abaixo esquerda) e L = 6000 (imagem abaixo direita)

```
O vetor NÃO ESTÁ está Ordenado

O vetor ESTÁ Ordenado

O tempo em milisegundos foi de: 32914
O vetor NÃO ESTÁ está Ordenado

O vetor ESTÁ Ordenado

O tempo em milisegundos foi de: 34960
```

```
O vetor NÃO ESTÁ está Ordenado

O vetor ESTÁ Ordenado

O tempo em milisegundos foi de: 47961
O vetor NÃO ESTÁ está Ordenado

O vetor ESTÁ Ordenado

O tempo em milisegundos foi de: 53437
```

Com 1.000.000 demorou muito (4,4 horas) para finalizar a operação de ordenar, ainda mais se comprado com L = 0, como temos aqui e L = 500.000 . Os últimos resultados foram esses.

```
O vetor NÃO ESTÁ está Ordenado

O vetor ESTÁ Ordenado

O tempo em milisegundos foi de: 945
O vetor NÃO ESTÁ está Ordenado

O vetor ESTÁ Ordenado

O tempo em milisegundos foi de: 15853574
```

Questão 04

Nesta questão o objetivo era desenvolver uma modificação do Algoritmo de HeapSort, chamado Double HeapSort que construísse dois Heaps, um máximo e um mínimo e logo após coordenasse a inserção em um vetor auxiliar, este por último que deve estar ordenado. A solução foi utilizar implementar a classe HeapSort com as funções tradicionais do HeapSort, utilizando também o Generics para ordenação de outros tipos de vetores. Foram implementadas nessa classe tanto as funções para um MinHeapSort como para um MaxHeapSort. Após, na classe na *Questao_04* foi programada a solução, construindo dois Heaps, um máximo e outro mínimo e usando um loop percorrendo metade do vetor, uma vez que vamos ordenar dois valores de uma vez só em cada iteração, utilizando as funções *min_heapfy* e *max_heapfy*, coordenando a inserção no vetor de resposta *answer*.

Funções MaxHeap

```java
public class HeapSort<T extends Comparable<T>> {

    public void build_max_heap(T[] array) {
        int n = array.length;
        for(int i=n/2-1; i>=0; i--) {
            max_heapfy(array, i, n);
        }
    }

    public void max_heapfy(T[] array, int i, int n) {
        int left = 2*i;
        int right = 2*i+1;
        int largest = i;

        if(left < n && array[left].compareTo(array[largest]) > 0) {
            largest = left;
        }

        if(right < n && array[right].compareTo(array[largest]) > 0) {
            largest = right;
        }

        if(largest != i) {
            T aux = array[i];
            array[i] = array[largest];
            array[largest] = aux;
            max_heapfy(array, largest, n);
        }

    }
```

Funções MinHeap

```java
public void build_min_heap(T[] array) {
        int n = array.length;
        for(int i=n/2-1; i>=0; i--) {
            min_heapfy(array, i, n);
        }

    }

    public void min_heapfy(T[] array, int i, int n) {
        int left = 2*i;
        int right = 2*i+1;
        int minimum = i;

        if(left < n && array[minimum].compareTo(array[left]) > 0) {
            minimum = left;
        }

        if(right < n && array[minimum].compareTo(array[right]) > 0) {
            minimum = right;
        }

        if(minimum != i) {
            T aux = array[i];
            array[i] = array[minimum];
            array[minimum] = aux;
            min_heapfy(array, minimum, n);
        }

    }
```

A resolução da questão é feita no código seguinte:

```java
public void double_heap_sort(T[] array,T[]copy, T[]answer) {
        int n = array.length;
        int end = n-1;
        int begin = 0;

        heapsort.build_max_heap(array);
        answer[end] = array[0];
        heapsort.build_min_heap(copy);
        answer[begin] = copy[0];

        T aux = array[0];
        array[0] = array[end];
        array[end] = aux;

        aux = copy[0];
        copy[0] = copy[end];
        copy[end] = aux;

        end--;
        begin++;

        for(int i=n-2; i>n/2-1; i--) {
            heapsort.max_heapfy(array, 0, i);
            answer[end] = array[0];
            aux = array[0];
            array[0] = array[i];
            array[i] = aux;

            heapsort.min_heapfy(copy,  0,  i);
            answer[begin] = copy[0];
            aux = copy[0];
            copy[0] = copy[i];
            copy[i] = aux;

            end--;
            begin++;
        }
```

Com as variáveis *end* e *begin* conseguimos coordenar a inserção no vetor de resposta. Além disso, precisamos de uma cópia do vetor que se quer ordenar, pois precisamos de dois vetores para construir os dois Heaps.

Resposta:

```
Vetor Original:
[ 12 2 4 10 9 1 3 5 ]
O vetor NÃO ESTÁ está Ordenado

Vetor Resposta:
[ 1 2 3 4 5 9 10 12 ]
O vetor ESTÁ Ordenado
```

Conclusão

Dessa forma, podemos perceber ao longo da implementação dos Algoritmos o quanto eles são importantes para o estudo da ordenação de elementos em vetores como um todo. Além disso, a utilização do Generics permite que os elementos desses vetores sejam de vários tipos, permitindo que os Algoritmos sejam aplicados em diversas situações.