UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE – UFRN

INSTITUTO METRÓPOLE DIGITAL – IMD

IMD0039 – ESTRUTURAS DE DADOS BÁSICA II

DOCENTE: VINICIUS PEREIRA SANTANA

DISCENTE: LUCAS MARCEL SILVA DE BRITO

LABORATÓRIO I

7 de novembro de 2024

MergeSort

Esse algoritmo é conhecido pelo uso da estratégia dividir o array em partes menores e ordená-las. Minha implementação dele está disponível abaixo:

```
6  public static void sort(int[] arr) {
7      if(Utils.isSorted(arr)) return;
8      sort(arr, 0, arr.length - 1);
9  };
10
11 private static void sort(int[] arr, int start, int end) {
12     if(start >= end) return;
13
14     int middle = Math.floorDiv(end - start + 1, 2);
15
16     int leftEnd = start + middle - 1;
17     int rightStart = start + middle;
18
19     sort(arr, start, leftEnd);
20     sort(arr, rightStart, end);
21
22     int index = 0;
23     int leftIndex = start;
24     int rightIndex = rightStart;
25
26     int[] temp = new int[end - start + 1];
27     while(index <= end - start) {
28         if(leftIndex > leftEnd) temp[index++] = arr[rightIndex++];
29         else if(rightIndex > end) temp[index++] = arr[leftIndex++];
30         else if(arr[leftIndex] < arr[rightIndex]) temp[index++] = arr[leftIndex++];
31         else temp[index++] = arr[rightIndex++];
32     };
33
34     for(int i = start; i <= end; i++) arr[i] = temp[i - start];
35 };
```

Separei em duas funções para melhorar a complexidade do melhor caso e também porque o Java não permite passar valores padrões para os parâmetros. Mas a implementação em si foi feita na segunda função, a privada (linha 11). Como estou utilizando recursividade, precisei definir um caso base, que é quando o intervalo recebido do *array* tem menos de dois elementos (linha 12). Em seguida, o array é dividido em dois intervalos que passam pelo processo de ordenação de forma independente (linhas 14 ~ 20).

Depois juntamos as duas partes já ordenadas selecionando o menor valor das próximas entradas de cada parte e colocando na ordem correta em um vetor temporário (linhas 22 ~ 32). Por fim, passamos os valores do vetor temporário já ordenado para o vetor final (linha 34). Não há necessidade de retorno porque, embora não seja explícito, no Java um array é tratado de forma similar a um ponteiro.

Complexidades

O pior caso não é tão fácil de se determinar, mas sabemos que verificar se o array já está ordenado (linha 7), o *while* (linhas 27 ~ 42) e o *for* (linha 34) são O(n), mas como temos uma recursão, precisamos de uma análise mais profunda.

Sejam f *a segunda função e* g a primeira função:

$$T(g(n)) = n + T(f(n)) = n + 2n + 2.T(f(n/2))$$

$$\Rightarrow T(g(n)) = n + T(f(n)) = n + 2n + 4(n/2 + 2.T(f(n/4)))$$

$$\Rightarrow T(g(n)) = n + T(f(n)) = n + 2n + 2n + 2n + 2n + \dots$$

E em um primeiro momento pode parecer que vai executar infinitamente, mas na verdade a segunda função continuará a recursividade até quando não for mais possível dividir n por 2. Portanto, seja k o número de vezes que será preciso realizar a divisão em duas partes, logo:

$$n/2^k = 1 \Rightarrow T\left(f\left(n/2^k\right)\right) = T(f(1)) = 1$$

Outra coisa que podemos obter é o valor de k em função de n:

$$n/2^k = 1 \Rightarrow n = 2^k \Rightarrow log_2 n = k$$

Como k é o número de vezes que as divisões serão feitas, podemos inferir também que:

$$T(f(n)) = k.2n = 2n.log_2 n$$

$$\Rightarrow T(g(n)) = n + T(f(n)) = n + 2n.log_2 n$$

Portanto, para o pior caso:

- Complexidade temporal: O($n.log_2 n$)
- Complexidade espacial: O(n)
    - Vale destacar que na teoria a cada término da chamada recursiva a memória alocada em *temp* é liberada (o que pode não ser bem o caso de linguagens com coletor de lixo, como o Java).

Para o melhor caso (*array* já ordenado):

- Complexidade temporal: $\Omega(n)$
- Complexidade espacial: $\Omega(1)$

BogoSort

Esse algoritmo é bem ineficiente e possui duas variações: probabilístico e determinístico. Aqui analisarei apenas a variação determinística, que consiste em testar todas as permutações possíveis do *array* até encontrar a versão ordenada. Minha implementação dele está disponível abaixo:

```
6  public static void sort(int[] arr) {
7      if(Utils.isSorted(arr)) return;
8      sort(arr, 0);
9  };
10
11 private static void sort(int[] arr, int offset) {
12     if(offset == arr.length - 1) return;
13
14     for(int i = offset; i < arr.length; i++) {
15         Utils.swap(arr, offset, i);
16         sort(arr, offset + 1);
17         if(Utils.isSorted(arr)) return;
18         Utils.swap(arr, offset, i);
19     };
20 };
```

Separei novamente em duas funções, onde a primeira (linhas 6 ~ 9) ficou responsável por verificar se o *array* já não está ordenado (linha 7) e, se for necessário, chamar a segunda função (linha 8). Na segunda função (linhas 11 ~ 20) é dado o início do processo de ordenação.

A variável *offset* está representando quantos elementos no *array* já tiveram sua posição determinada na permutação das chamadas recursivas. O caso base dessa função é, portanto, quando todos os elementos já possuem sua posição definida na permutação atual (linha 12). No *for* (linhas 14 ~ 19) o vetor recebido como parâmetro é percorrido a partir do *offset* atual. A cada interação é criada uma permutação nova através da chamada recursiva (linhas 15 ~ 16) . Quando a nova permutação é concluída, é verificado se ela está ordenada (linha 17). Se estiver, a função termina, caso contrário, a troca que gerou a permuta que não deu certo é desfeita (linha 18) para testar a próxima permutação.

Complexidades

Sabemos que o for (linhas 14 ~ 19) e verificar se o array está ordenado (linhas 8 e 17) é O(n). Além disso, as trocas de elementos (linhas 15 ~ 18) é O(1). Mas, como temos uma recursão, a análise volta a ficar complexa.

Sejam f *a segunda função e* g a primeira função:

$$T(g(n)) = n + T(f(n)) = n + n(n + T(f(n - 1)))$$

$$\Rightarrow T(g(n)) = n + T(f(n)) = n + n(n + (n - 1)(n - 1 + T(n - 2)))$$

$$\Rightarrow T(g(n)) = n + T(f(n)) = n(n - 1)(n - 2) \dots 1 + \dots.$$

$$\Rightarrow T(g(n)) = n + T(f(n)) = !n + \dots$$

No geral, o termo $!n$ é o mais dominante dos termos que aparecem na equação. Portanto, para o pior caso:

- Complexidade temporal: O(!n)
- Complexidade espacial: O(1)

Para o melhor caso (*array* já ordenado):

- Complexidade temporal: $\Omega(n)$
- Complexidade espacial: $\Omega(1)$

HeapSort

Esse algoritmo é aparentemente o mais complexo de se entender e se baseia no uso de uma estrutura de dados chamada *heap*. Para não ter que implementar essa estrutura na forma de objeto, optei pela representação vetorial dela. Como o objetivo aqui não é falar sobre ela em si, não entrarei em detalhes sobre ela.

Minha implementação do algoritmo está disponível abaixo:

```
6  public static void heapfy(int[] arr) {
7      for(int i = Math.floorDiv(arr.length, 2); i >= 0; i--) {
8          maxHeap(arr, i, arr.length);
9      };
10 };
11
12 public static void maxHeap(int[] arr, int node, int size) {
13     int leftChild = (node * 2) + 1;
14     int rightChild = leftChild + 1;
15
16     int bigger = node;
17     boolean hasLeft = leftChild < size;
18     boolean hasRight = rightChild < size;
19
20     if(!hasLeft && !hasRight) return;
21     if(hasLeft && arr[leftChild] > arr[bigger]) bigger = leftChild;
22     if(hasRight && arr[rightChild] > arr[bigger]) bigger = rightChild;
23
24     if(node != bigger) {
25         Utils.swap(arr, node, bigger);
26         maxHeap(arr, bigger, size);
27     };
28 };
29
30 public static void sort(int[] arr) {
31     for(int i = 0; i < arr.length; i++) {
32         Utils.swap(arr, 0, arr.length - 1 - i);
33         maxHeap(arr, 0, arr.length - 1 - i);
34     };
35 };
```

Ao contrário das outras implementações, dessa vez temos três funções. A primeira (linhas 6 ~ 10) na verdade tem como objetivo transformar um vetor na representação vetorial da *heap* (não entrarei em detalhes). A segunda (linhas 12 ~ 28) é essencial para as demais, e tem como objetivo analisar um determinado nó e seus filhos a fim de decidir qual armazena o maior valor (linhas 22) e, depois, no caso do pai não conter o maior valor, trocar o pai pelo filho com maior

valor (linha 25). Se ocorrer uma troca, para que a representação vetorial da *heap* se mantenha em ordem, repetimos o processo utilizando recursividade (linha 26).

A terceira função (linhas 30 ~ 35) é essencialmente o algoritmo de ordenação. O array é percorrido (linhas 31 ~ 34) e o primeiro valor da *heap* (que é o maior) é trocado com o valor do final do *array*. Feito isso, corrigimos a representação vetorial da *heap* (linha 33), porém, agora olhando para uma *heap* com o tamanho reduzido. Assim, o maior valor continuará no final do *array*.

Complexidades

Nenhuma das funções implementadas alocam recurso extra em função de n, logo todas tem complexidade espacial O(1). A primeira função percorre o *array* um total de $n/2$ vezes chamando a segunda e a terceira um total de n vezes realizando uma troca O(1) e chamando a segunda. Como novamente estamos usando recursão, se faz necessário uma análise mais complexa.

Sejam f a última função, g a segunda e h a primeira.

$$T(g(n)) = 2 + T(g(n/2))$$

$$T(g(n)) = 2 + 2 + T(g(n/4))$$

$$T(g(n)) = 2 + 2 + 2 + T(g(n/8))$$

E assim a segunda função continuará a recursividade até não ser mais possível dividir n por 2. Portanto, seja k o número de vezes que será preciso realizar a divisão em duas partes, logo:

$$n/2^k = 1 \Rightarrow T\left(g\left(n/2^k\right)\right) = T(g(1)) = 1$$

Outra coisa que podemos obter é o valor de k em função de n:

$$n/2^k = 1 \Rightarrow n = 2^k \Rightarrow log_2 n = k$$

Como k é o número de vezes que as divisões serão feitas, podemos inferir também que:

$$T(g(n)) = k.2 \Rightarrow 2.log_2 n$$

$$\Rightarrow T(f(n)) = n + T(g(n)) = n + 2.log_2 n$$

$$\Rightarrow T(f(h(n)) = n/2 + T(g(n/2)) + T(f(n))$$

$$\Rightarrow T(f(h(n)) = 3n/2 + 2.log_2(n/2) + 2.log_2 n$$

Portanto, para o pior caso (precisando ou não do *heapify*):

- Complexidade temporal: O($n.log_2 n$)
- Complexidade espacial: O(1)

Ao contrário dos outros algoritmos, não temos um vetor ordenado como melhor caso, pois a representação vetorial de uma *heap* não é ordenada. Logo, o melhor caso será o que precisar de menos trocas. Mas, como na ordenação a heap está sendo alterada a cada iteração até mesmo em tamanho, a complexidade se mantém. Portanto, para o melhor caso (precisando ou não do *heapify*):

- Complexidade temporal: $\Omega(n.log_2 n)$
- Complexidade espacial: $\Omega(1)$

Métodos auxiliares

Segue abaixo métodos todos os demais métodos implementados para o desenvolvimento deste laboratório:

```
3  public class Utils {
4      public static void swap(int[] arr, int a, int b)  {
5          int value = arr[a];
6          arr[a] = arr[b];
7          arr[b] = value;
8      };
9
10     public static boolean isSorted(int[] arr) {
11         for(int i = 0; i < arr.length - 1; i++) {
12             if(arr[i] > arr[i + 1]) return false;
13         };
14
15         return true;
16     };
17 };
```

Testes

Segue abaixo os testes realizados:

```
public class Main {
    private static Random rand = new Random();
    public static void main(String[] args) {
        rand.setSeed(LocalTime.now().toNanoOfDay());
        System.out.println("Testing merge sort....");
        for (int i = 0; i < 30; i++) {
            int[] raw = generateArray(10);
            MergeSort.sort(raw);
            _assert(Utils.isSorted(raw));
        };

        System.out.println("Testing heap sort....");
        for (int i = 0; i < 30; i++) {
            int[] raw = generateArray(10);
            HeapSort.heapfy(raw);
            HeapSort.sort(raw);
            _assert(Utils.isSorted(raw));
        };

        System.out.println("Testing bogo sort....");
        for (int i = 0; i < 30; i++) {
            int[] raw = generateArray(10);
            BogoSort.sort(raw);
            _assert(Utils.isSorted(raw));
        };

        System.out.println("All tests passed!");
    };

    private static void _assert(boolean bool) {
        if(!bool) throw new Error("Teste falhou...");
    };

    private static int[] generateArray(int size) {
        int[] arr = new int[size];
        for (int i = 0; i < size; i++) arr[i] = rand.nextInt(100) - 50;
        return arr;
    };
};
```