Algoritmos de Ordenação – parte II

Indução Forte (primeira alternativa):

Caso base: para n = 1 temos um conjunto ordenado de um único elemento.

Passo indutivo: assumimos que sabemos ordenar qualquer conjunto com n/2 elementos. A partir dos dois subconjuntos ordenados recursivamente precisamos fazer a intercalação dos dois de forma a obter o conjunto ordenado.

Conjunto não ordenado

Por hipótese de indução, sabemos ordenar os conjuntos com n/2 elementos

Cada subconjunto está ordenado, mas o conjunto como um todo não, é necessário fazer a intercalação (ou fusão) dos dois subconjuntos

Esta indução dá origem ao algoritmo *Merge Sort* (Ordenador por Intercalação ou por Fusão).

Divisão: não é necessário fazer nada, só calcular o índice da posição da metade. **Conquista**: execução das duas chamadas recursivas (da metade do conjunto à esquerda e à direita). **Combinação**: é necessário realizar a intercalação (ou fusão) dos dois subconjuntos que estão ordenados.

Algoritmo Iterativo para a Fusão/Intercalação

```
static void merge(int[] A, int p, int q, int
r) {
    int i, j, k;
    int tamseq1 = q - p + 1;
    int tamseq2 = r - q;
    int[] seq1 = new int[tamseq1];
    int[] seq2 = new int[tamseq2];
    for (i = 0; i < seq1.length; i++) {
        seq1[i] = A[p + i];
    }
    for (j = 0; j < seq2.length; ++j) {
        seq2[j] = A[q + j + 1];
    }

    k = p;
    i = 0;
    j = 0;
```

```
    while (i<seq1.length && j<seq2.length) {
        if (seq2[j] < seq1[i]) {
            A[k] = seq2[j];
            j++;
        } else {
            A[k] = seq1[i];
            i++;
        }
        k++;
    }
    while (i < seq1.length) {
        A[k] = seq1[i];
        k++;
        i++;
    }
    while (j < seq2.length) {
        A[k] = seq2[j];
        k++;
        j++;
    }
}
```

O algoritmo de intercalação (ou fusão) não é recursivo.

Melhor caso (todos os elementos do segundo subconjunto são maiores ou iguais ao que os elementos do primeiro subconjunto)**:**
operações(n) = n/2

Pior caso (todos os elementos do segundo subconjunto são maiores ou iguais do que os elementos do primeiro subconjunto, exceto o último elemento do primeiro que é maior do que todos do segundo)**:**
operações(n) = n - 1

Para ambos os casos **$T(n) \in \Theta(n)$**

```java
private void mergeSort(int[] A, int ini,
int fim) {
    if (ini < fim) {
        int meio = (ini + fim) / 2;
        mergeSort(A, ini, meio);
        mergeSort(A, meio + 1, fim);
        merge(A, ini, meio, fim);
    }
}
```

Caso geral:

$T(n) = 0$ para $n \leq 1$

$T(n) = 2*T(n/2) + \Theta(n)$ para $n > 1$ (par)

$T(n) = T((n+1)/2) + T((n-1)/2) + \Theta(n)$ n ímpar

Melhor caso (em relação à intercalação)**:**

$T(n) = 0$ para $n \leq 1$

$T(n) = 2*T(n/2) + n/2$ para $n > 1$

Pior caso (em relação à intercalação)**:**

$T(n) = 0$ para $n \leq 1$

$T(n) = 2*T(n/2) + n - 1$ para $n > 1$

Se calcularmos a complexidade assintótica, teremos (para todos os casos): **$T(n) \in \Theta(n*\log n)$**

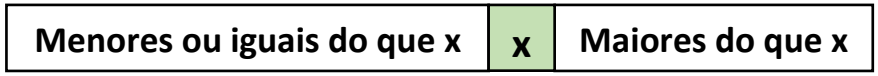
Indução Forte (segunda alternativa):

Caso base: para n = 1 temos um conjunto ordenado de um único elemento.

Passo indutivo: assumimos que sabemos ordenar qualquer conjunto com menos do que n elementos. Dado um elemento qualquer (escolhido como pivô) vamos colocar os elementos menores ou iguais do que ele do lado esquerdo do arranjo e maiores do lado direito.

Conjunto não ordenado

Separamos à esquerda os elementos menores ou iguais a x e a direita os maiores e colocamos o x (pivô) entre eles



Por hipótese de indução, sabemos ordenar os conjuntos com menos do que n elementos

Esta indução dá origem ao algoritmo *Quick Sort*.

Divisão: separar os elementos menores ou iguais ao pivô no lado esquerdo do arranjo e maiores do lado direito, colocando o pivô entre estes dois subconjuntos. **Conquista**: execução das duas chamadas recursivas (dos elementos à esquerda do pivô e dos elementos à direita). **Combinação**: não é necessário fazer nada, o arranjo resultante já está ordenado.

```java
static int particao (int[] A, int ini, int
fim) {
    int i, j, temp;
    int x = A[fim]; // pivô
    i = ini;
    j = fim - 1;
    while (i <= j) {
        if(A[i] <= x) {
            i++;
        } else if (A[j] > x) {
            j--;
        } else { // trocar A[i] e A[j]
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
            i++;
            j--;
        }
    }
    A[fim] = A[i]; // reposicionar o pivô
    A[i] = x;
    return i;
}
```

O algoritmo de partição não é recursivo.

Melhor caso 1 (para a partição, um melhor caso é: x é o maior elemento do conjunto analisado)**:**
operações(n) = n - 1

Melhor caso 2 (A[i] é sempre maior que x e A[j] é sempre menor ou igual [x será a mediana, mas do lado esquerdo estarão os maiores que x e do lado direito os menores ou iguais)]**:**
operações(n) = 2*(n – 1)/2 = n - 1

Pior caso (para a partição, o pior caso é: o primeiro elemento é o maior de todos e x é o menor elemento do conjunto analisado)**:**
operações(n) = 2*n - 2

Para os três casos: **$T(n) \in \Theta(n)$**

```
static void QuickSort(int[] A, int ini,
int fim) {
    if (ini < fim) {
        int q = particao(A, ini, fim);
        QuickSort(A, ini, q - 1);
        QuickSort(A, q + 1, fim);
    }
}
```

Melhor caso:

$T(n) = 0$ para $n \leq 1$

$T(n) = 2*T(n/2) + n - 1$ para $n > 1$

Se calcularmos a complexidade assintótica, teremos: $\mathbf{T(n) \in \Theta(n*\log n)}$

Pior caso:

$T(n) = 0$ para $n \leq 1$

$T(n) = T(n-1) + T(0) + 2*n - 2$ para $n > 1$

Que é o mesmo que:

$T(n) = T(n-1) + 2*n - 2$ para $n > 1$

Se calcularmos a complexidade assintótica, teremos: $\mathbf{T(n) \in \Theta(n^2)}$

O caso médio do *Quick Sort* tem complexidade **$\Theta(n*\log n)$**

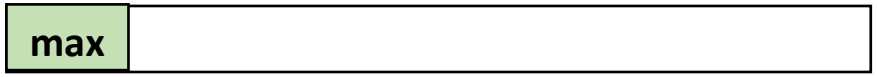
Terceira alternativa:

Caso base: para n = 1 temos um conjunto ordenado de um único elemento.

Passo indutivo: encontramos o maior elemento do conjunto e assumimos que sabemos ordenar **n-1** (excluindo-se o maior).

Conjunto não ordenado 

Organizamos os dados como um Heap Máximo

Encontramos o maior 

Trocamos o max de lugar com o último número do subconjunto que estamos analisando (***mas mantendo o restante da estrutura como um Heap Máximo***). Por hipótese de indução, sabemos ordenar n-1 elementos

Não falta fazer nada pois o max já está no lugar certo (todos os elementos à esquerda dele são menores do que ele).

Diferença para o *Selection Sort*: organizaremos os dados de forma a encontrar o valor do máximo de maneira mais eficiente, utilizando uma estrutura de dados chamada de *Heap*. **Esta indução dá origem ao algoritmo *Heap Sort*.**

Heap Máximo:

Estrutura de Dados baseada numa árvore binária completa (todos os elementos preenchidos de cima para baixo e da esquerda para a direita).

A regra do Heap Máximo (além de ser uma árvore binária completa) é que todo nó tem chave/valor maior ou igual ao valor dos seus filhos.

```java
static void refazHeapMax(int A[], int i,
int compHeap) {
    int esq, dir, maior, temp;
    esq = 2 * i + 1;
    dir = 2 * i + 2;
    if (esq<compHeap && A[esq]>A[i]) {
        maior = esq;
    } else {
        maior = i;
    }

    if (dir<compHeap && A[dir]>A[maior]) {
        maior = dir;
    }

    if (maior != i) {
        temp = A[i];
        A[i] = A[maior];
        A[maior] = temp;
        refazHeapMax(A, maior, compHeap);
    }
}
```

Melhor caso[*,**] (o elemento atual sempre está no lugar certo):

$T(n) = 0$ para $n \leq 1$

$T(n) = 2$ para $n > 1$

[*] **É possível acontecer no método constrói heap se os elementos já estavam organizados como um heap máximo.** [**] **Só vai acontecer 'sempre' durante a execução Heap Sort se todos os elementos forem iguais (irreal).**

Fora desse melhor caso, a recursão poderá ir para a esquerda ou direita. Dependendo do valor de n a esquerda e a direita terão o mesmo tamanho ((n-1)/2) [n igual a uma potência de 2 menos um, para ocorrer sempre], porém se n for igual a soma de duas potências consecutivas de dois mais um, então teremos o lado esquerdo com cerca de 2/3 dos elementos e o direito com 1/3, mas isso só ocorrerá uma vez[***]**).**

Pior caso:

$T(n) = 0$ para $n \leq 1$

$T(n) = T(2*n/3)$[***] $+ 2$ para $n > 1$

Pior caso geral:

$T(n) = 0$ para $n \leq 1$

$T(n) = T(\lfloor n/2 \rfloor) + 2$ para $n > 1$

Se calcularmos a complexidade assintótica, dos casos reais: **$T(n) \in O(\log n)$**

```java
// VERSÃO ITERATIVA

static void constroiHeapMaxIterativo(int[]
A) {
    int compHeap = A.length;
    for (int i=(A.length)/2-1; i>=0; i--){
        refazHeapMax(A, i, compHeap);
    }
}

// VERSÃO RECURSIVA

static void constroiHeapMaxRec(int[] A,
int i) {
    int n = A.length;
    if (i < n/2){
        constroiHeapMaxRec(A, 2*i+1);
        constroiHeapMaxRec(A, 2*i+2);
        refazHeapMax(A, i, n);
    }
}
```

Equação de recorrência (da versão recursiva):

Caso geral:

$T(n) = 0$ para $n \leq 1$

$T(n) = 2*T(\lfloor n/2 \rfloor) + O(\log n)$ para $n > 1$

Se calcularmos a complexidade assintótica, dos casos reais: **$T(n) \in \Theta(n)$**

```java
static void heapSort(int A[]) {
    int i, compHeap, temp;
    compHeap = A.length;
    constroiHeapMaxRec(A,0);

    for (i = A.length - 1; i > 0; i--) {
        temp = A[0];
        A[0] = A[i];
        A[i] = temp;
        compHeap--;
        refazHeapMax(A, 0, compHeap);
    }
}
```

O algoritmo *Heap Sort* em si costuma ser implementado de maneira iterativa usando o princípio incremental / de indução fraca. Porém, ele chama dois métodos baseados em indução forte: *constroiHeapMaxRec* e *refazHeapMax*.

Operações(n) = Θ(n) + (n-1)*O(log n)
Operações(n) ϵ O(n*log n)