Analise de Complexidade (Big-O) em Algoritmos de ordenação

Enzo Tiriba Appi matricula: 1801130023

Bubble Sort

O *Bubble Sort* é um algoritmo simples que é usado para classificar um determinado conjunto de **n** elementos fornecidos na forma de uma matriz, vetor e/ou lista.

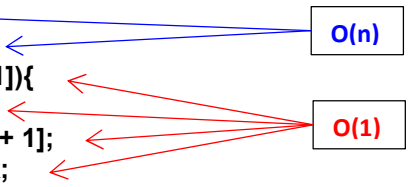
Este tipo de algoritmo de ordenação, classificado como bolha, compara todo o elemento um por um e os classifica com base em seus valores. O nome *bubble* advém do motivo de que a cada iteração realizada, o maior elemento da estrutura fornecida, borbulha em direção à última posição, ou ao índice mais alto, assim como uma bolha de água sobre à superfície da água.

Como temos **n** elementos, precisaremos repetir esse processo em **n-1** vezes. E, ordenação ocorre ao percorrermos todos os elementos, um por um, comparando-os com o elemento adjacente. Trocando-os, se necessário.

Se uma estrutura de dados especificada precisar ser ordenada de forma crescente, o algoritmo *bubble* iniciar-se-á comparando o primeiro elemento da estrutura, com o segundo elemento. Se o primeiro elemento for maior que o segundo, ele trocará a posição de ambos elementos entre si. E, em seguida, prossigará à compararação entre o segundo e o terceiro elemento da estrutura, e assim por diante.

Exemplo de pseudocódigo algoritmo *bubble sort*:

```
void bubbleSort(int* vetor, int n){
    for(k = 0; k < n; k++){
        for(j = 0; j < n - 1; j++){
            if(vetor[j] > vetor[j + 1]){
                aux = vetor[j];
                vetor[j] = vetor[j + 1];
                vetor[j + 1] = aux;
            }
        }
    }
}
```



Total das Complexidades = O(n) * O(n) + 4O(1) = $O(n^2)$ + 4O(1) = **$O(n^2)$**

No psedocódigo, anteriormente descrito, as comparações **n-1** serão feitas na 1ª passagem; **n-2** na 2ª passagem, **n-3** na 3ª passagem e assim por diante. Portanto, o número total de comparações será: (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = n(n-1)/2. Conclui-se que a **complexidade temporal, no pior cenário** do ***Bubble Sort*** **é $O(n^2)$.**

Selection Sort

A ordenação por seleção é, conceitualmente, o algoritmo de classificação mais simples. Esse algoritmo, primeiramente, encontrará o menor elemento da estrutura fornecida, e, o trocará pelo elemento que está na primeira posição. Em seguida, encontrará o segundo menor elemento e o trocará pelo elemento que está na segunda posição; E, continuará fazendo isso até que toda a estrutura seja ordenada.

É chamado de ordenação por seleção porque seleciona repetidamente o próximo menor elemento e o troca no lugar certo.

Exemplo de pseudocódigo algoritmo *selection sort*:

```
void selectionSort(int* vetor, int n){
    for(i = 0; i < n - 1; i++){
        for(j = i + 1; j < n; j++){
            if(vetor[i] > vetor[j]){
                aux = vetor[i];
                vetor[i] = vetor[j];
                vetor[j] = aux;
            }
        }
    }
}
```

O(n)

O(1)

Total das Complexidades = O(n) * O(n) + 4O(1) = O(n²) + 4O(1) = **O(n²)**

A ordenação por seleção requer dois *for* aninhados para que os *loops* sejam concluídos. Portanto, para um determinado tamanho de entrada **n**, a **complexidade temporal** do algoritmo no **pior cenário (Big-O)** é: **O(n²)**.

Insertion Sort

A ordenação por inserção é um algoritmo simples que organiza uma estrutura final um item por vez. É muito menos eficiente em estruturas grandes do que algoritmos mais avançados, como quicksort, heapsort ou merge sort.

O algoritmo itera, consumindo um elemento de entrada a cada repetição e aumenta uma estrutura de saídas, ordenada. A cada iteração um elemento dos dados de entrada é removido, localizando-se o local a que pertence na estrutura classificada e o inserindo-o lá. Ele se repete até que nenhum elemento de entrada permaneça.

Exemplo de pseudocódigo algoritmo *insertion sort*:

```
void insertionSort(int* vetor, int n) {
    for(int i = 1; i < n; i++) {
        int tmp = vetor[i];
        int j = i;
        while(j > 0 && tmp < vetor[j - 1]) {
            vetor[j] = vetor[j - 1];
            j--;
        }
        vetor[j] = tmp;
    }
}
```

O(n) — for(int i = 1; i < n; i++) {
O(1) — int tmp = vetor[i]; / int j = i;
O(n) — while(j > 0 && tmp < vetor[j - 1]) {
O(1) — vetor[j] = vetor[j - 1]; / j--;

Total das Complexidades = O(n) * O(n) + 5O(1) = $O(n^2)$ + 5O(1) = **$O(n^2)$**

QuickSort

Quicksort é um algoritmo de dividir e conquista que funciona selecionando um elemento 'pivô', da estrutura fornecida, e particionando os outros elementos em duas sub-estruturas, dependendo se são menores ou maiores que o pivô. As sub-estruturas são, então, ordenadas recursivamente. Isso pode ser feito localmente, exigindo pequenas quantidades adicionais de memória para realizar a ordenação. O que significa que ele pode classificar itens de qualquer tipo para os quais uma relação "menor que" é definida. Implementações eficientes do *Quicksort* não são uma estáveis, o que significa que a ordem relativa dos itens de classificação igual não é preservada.

Exemplo de pseudocódigo algoritmo quick *sort*:

```
void quickSort(int* vetor, int n) {
   int localizacao;
   if (inicio < fim) {
      localizacao = particao(vetor, inicio, fim)
      quickSort(vetor, inicio, localizacao);
      quickSort(vetor, localizacao + 1, fim);
}
```

O(1) — int localizacao; / if (inicio < fim) {
O(n) — localizacao = particao(vetor, inicio, fim)
O(n) — quickSort(vetor, inicio, localizacao);
O(n-1) — quickSort(vetor, localizacao + 1, fim);

Total das Complexidades = O(1) + O(n) + (O(n) + O(n-1)) = O(n) +$O(n^2)$ = **$O(n^2)$**

O pior caso ocorre para as entradas que correspondem às listas ordenadas ou invertidas, em que sistematicamente, após cada partição resulta uma sublista vazia. Após cada varredura n1 = 0 ou n2 = 0. Daí pode-se escrever que:

$C(n) \leq n + 1 + C(n - 1)$
$C(n - 1) \leq n + C(n - 2)$
$C(n - 2) \leq n - 1 + C(n - 3)$
. . .
$C(2) \leq 3 + C(1)$
Como C(1) = 0, por substituições sucessivas obtemos:
$C(n) \leq (n + 1) + n + (n - 1) + ... + 3 = 1$
/2 (n – 1) (n + 4)
Portanto wC(n) ≤ ½ (n – 1) * (n + 4) = **$O(n^2)$**

Shell Sort

Advindo de um refinamento do método de ordenação por inserção. O método *Shellsort* realiza a troca de elementos distantes um do outro. Os itens separados por h posições são agrupados e ordenados, o valor de h é decrementado seguindo uma sequencia qualquer (não há uma sequencia definitiva) até que atinja o valor unitário, onde a ordenação corresponde ao algoritmo de inserção, mas nesse ponto nenhum elemento precisa se mover para uma posição muito distante. O *Shellsort* é uma ótima opção para listas de tamanho médio, é um método simples e eficiente.

Exemplo de pseudocódigo algoritmo *shell sort*:

```
void shellSort (int* vetor, int n) {
    int h, i, j, t;
    for (h = n; h /= 2;) {
        for (i = h; i < n; i++) {
            t = vetor[i];
            for (j = i; j >= h && t < vetor[j - h]; j -= h) {
                vetor[j] = vetor[j - h];
            }
            vetor[j] = t;
        }
    }
}
```

A complexidade temporal do algoritmo, em seu pior Caso (quando os elementos do vetor estão na ordem reversa), é: **$O(n \log^2 n)$**

As fórmulas de complexidade deste algoritmo são bem vagas, pois até hoje ninguém foi capaz de analisar este algoritmo, isso se deve-se a problemas matemáticos muito difíceis que sua análise traz.

Merge Sort

Exemplo de pseudocódigo algoritmo *shell sort*:

```
void mergeSort(int* vetor, int l, int r){
	if(r > l){
		int i, m = (l + (r - 1)) / 2;
		mergeSort(vetor, l, m);
		mergeSort(vetor, m + 1, r);
		intercalar(vetor, l, m, r);
	}
}
```