

**EDA 2023.1**

**Diogo F.**

**Algoritmos de Ordenação**

Pontos a serem observados em um algoritmo:

**CHECKLIST DE ALGORITMOS:**

[OK] Métodos elementares:

(OK) Selection

(OK) Bubble (+ Shaker)

(OK) Insertion ( + Insertion otimizado)

(OK) ShellSort (não é tão elementar assim)

[OK] MergeSort

[OK] QuickSort

**1) Complexidade assintótica:**

* **O (n2):** arquivos pequenos, muitas comparações - alto CUSTO;
* **O (nlogn):** arquivos grandes, menos comparações (porém, complexos); custo menor.

**2) Estabilidade:** é estável quando a *ORDEM RELATIVA* dos elementos é mantida;

**3) Adaptabilidade:** é adaptativo quando a ordenação existente é aproveitada.

**4) Memória extra (in place):** a quantidade de memória extra usada é um fator que afeta o tempo de execução. Ele será ***IN-PLACE*** quando não utilizar nenhuma estrutura de memória extra.

**METODOS ELEMENTARES:**

**I) Vantagens:** são mais simples, bons para arquivos pequenos e com um propósito geral (independe da estrutura);

**II) Desvantagens:** custoso -> O (n2).

**III) Quais são eles?**

**1) SelectionSort (OK)**

**2) InsertionSort (OK)**

**3) BubbleSort (OK)**

**4) ShellSort (OK)**

**[SelectionSort]:**

* **void selection(int v[], int l, int r)**
* Precisa de 2 estruturas “for”. A primeira, com de i = 0, até i <= r. Essa fará com que cada posição do vetor tenha o elemento correto, dentro dela haverá a variável “menor = i”, para fins de comparação e procura do menor elemento. A outra, de j = i+1, até j <= r. Essa fará as comparações para achar os menores elementos.
* Acha o menor número dentro de um vetor v[], e coloca na posição [0];
* Depois, acha o segundo menor número e coloca na posição [1], e assim por diante...
* Utiliza uma variável “menor”, que começa valendo 0 (i);

**[BubbleSort]**

* **void bubble (int v[], int l, int r);**
* A ideia é ir flutuando o menor número até o topo *(r até l)*. É necessárias 2 estruturas “for”. A primeira, com o incremento “i++”, de l até r, para delimitar cada flutuação. O segundo incremento “j—“, sendo este de l até > i, realizando o swap sempre que j for menor que j-1.
* É in place, não é estável.
* Sobre o **SHAKER**: é um bubble otimizado. Consiste em realizar uma iteração para colocar o *menor elemento* em **cima** e na *volta* colocar o *maior elemento* no **fundo**

**[InsertionSort]**

* O melhor, comparado aos outros dois elementares
* É adaptativo, se o vetor já possui uma pré ordenação, ele é muito eficiente
* **Void insertion(int v[], int l, int r)**
* Consiste em comparar um elemento com seu antecessor, de l+1 até <= r, realizando as trocas.
* **1º for:** for (int i = l+1; i <=r; i++): *l+1*, pois o v[0] já será acessado dentro do 2º for.
* **2º for:** for (int j = i; j > l; j--): aqui, de fato, ele compara um elemento com o seu antecessor, ordenando.
* Problemas dessa versão: realiza muitas trocas *(swap)* desnecessárias.
* Solução: utilizar um sentinela. Estratégia: colocar o menor elemento do vetor diretamente no v[0], com o seguinte for: *for (int i = r; i > 0; i--) compara e troca v[i] e v[i-1]*. = semelhante ao bubble, porém apenas com o menor elemento.
* Após isso, o 1º for passa a ter i = l + 2. Utiliza-se uma variável *tmp*. Abre espaço e realiza a troca quando for necessário.

**[ShellSort]**

* Donald Shell
* Complexidade (próxima) n2
* Desempenho razoavelmente bom, comparado ao insertion.
* É difícil calcular a complexidade assintótica, porque escolher os saltos é bastante importante para o desempenho, ou seja, **DEPENDE**.
* Surgiu devido à **ineficiência** do insertionsort (em casos em que o elemento está muito deslocado da sua posição final, ele irá demorar).
* Invés de olhar apenas adjacentes, ele procura em intervalos. (Ex: de 3 em 3). Ainda fará várias passagens (com saltos diferentes), e obrigatoriamente, o menor salto é com o *1* (que é exatamente o insertionsort). Ou seja, a última passada é exatamente o Insertion.
* Depende do ***insertionH***

**[MergeSort]**

* Duas funções: merge\_sort -> separa em subvetores, e MERGE (ao final)
* Chamadas recursivas, a cada chamada divide em sub-vetores para serem ordenados, até que l >= r, ou seja, tenha tamanho **unitário**

**merge\_sort(v, l, m)**

**merge\_sort(v, m+1, r)**

**merge(v, l, m, r)**

* Melhor/médio/pior caso: O(NlogN) = ou seja, não muda, *uma grande vantagem*.
* Dividir em pequenas partes, ordená-las, e combinar novamente *(merge)*, até formar uma sequência ordenada.
* **NÃO é in-place** (ou seja, utiliza memória extra significativa).
* É estável.
* Não é recomendada a utilização caso tenha problema com o espaço disponível.
* Sub-vetores pequenos: recomenda-se alterar para o **InsertionSort** (cerca de 15 itens, mais ou menos)
* Tempo similar ao ShellSort (porém, no shell ainda não foi comprovado que é NlogN para dados aleatórios)

**[QuickSort]**

* Um dos mais utilizados (simples, eficiente)
* **Melhor caso**: O(N log N)
* **Pior caso**: O (N2) – *é raro* – muitos dados repetidos, etc
* Estável: **não** altera a ordem de dados iguais
* Desvantagens: como escolher o pivot?
* Método **dividir e conquistar**
* Diferença pro **Merge**:
  + Merge = divide, ordena separadamente, combina reordenando (e vai conquistando um vetor mais ordenado), repete;
  + Quick = rearranja, conquista um elemento ordenado e dois sub-vetores pseudo-ordenados; divide, repete;
* **Partition (operação fundamental):** escolhe um *elemento de referencia*: pivot
* Partition original = 3 \* o tamanho do vetor, pois utiliza-se um vetor para salvar os maiores e menores que o pivot.
* Partition do Cormen
* Partition do Sedgewick
* Rearranjar: todos os elementos posteriores ao pivot, e reposiciona o pivot;
* Os *menores* que o pivot de um lado, os *maiores* de outro lado
* Dividir o vetor em dois
* Repetir o processo até ordenar todos os elementos
* **Quicksort(int v[], int l, int r)**

Int j;

If(l>=r) return // fim da recursividade;

J = partition(v, l, r);

Quicksort(v, l, j-1);

Quicksort(v, j, r);



