Introdução à ordenação

Nesta seção examinaremos os algoritmos mais básicos para ordenação de uma lista de dados (vetores). Os algoritmos examinados aqui não são eficientes e por isso para aplicações reais usualmente são implementados outros (como *Ordenação Rápida/Quick Sort*, *Ordenação por Fusão/Merge Sort* ou *Ordenação por Monte/Heap Sort*). O último a ser examinado (Inserção), é mais eficiente, principalmente se os dados estiverem "quase-ordenados".

O problema é, dada uma sequência de valores (em lista ou vetor), ordenar crescente (ou decrescente) essa sequência. Por exemplo, na figura abaixo, inicialmente os dados não estão ordenados `{4, 2, 3, 0, 9, 3, 5, 7}`. Um algoritmo de ordenação que recebesse esse vetor com `n=8` valores, deveria trocar os valores de posição de modo a conseguir a seguinte situação final para o vetor: `{0, 2, 3, 3, 4, 5, 7, 9}`.

Fig. 1. Ilustração dos dados na configuração inicial e na final (já ordenado `V[0]≤V[1]≤...≤V[n-1]`).

Para simplificar as explicações fixaremos um nome para o vetor e usaremos a notação `V[i]` para indicar o valor que está na posição `i` do vetor. Também convencionaremos que a primeira posição do vetor seja `V[0]`, logo um vetor com *N* elementos terá o último deles na posição `V[N-1]`.

Os métodos que examinaremos são os conhecidos: *Método da Bolha (Buble Sort)*, *Seleção Direta* e *Inserção Direta*.

1. Método da Bolha (ou "Pedregulho")

A ideia do **método da Bolha** é inicia comparando os dois últimos elementos, o menor fica à esquerda, então comparar os dois anteriores e fazer a mesma coisa, desse modo o **menor vai movendo-se para cima** (como as bolhas). Mas também podemos fazer o "inverso", seguir empurrando o maior para baixo e assim, poderíamos apelidar essa variante de **método do Pedregulho**.

Vamos detalhar a primeira fase do **método do Pedregulho**, nessa fase empurramos o maior do `N` elementos para a última posição. O primeiro passo é comparar `V[0]` com `V[1]`, se o valor em `V[0]` for maior que o valor em `V[1]`, troque seus valores (diremos apenas `Se (V[0] > V[1], troque-os`). O passo 2 é: `Se (V[1] > V[2], troque-os` (logo `V[2]` terá o maior dos 3 primeiros). O passo 3 é: `Se (V[2] > V[3], troque-os` (logo `V[3]` terá o maior dos 4 primeiros). Segue comparando `V[i]` com `V[i+1]`, até o passo N-1: `Se (V[N-2] > V[N-1], troque-os` (logo `V[N-1]` terá o maior dos 4 primeiros).

A figura abaixo ilustra os *7* primeiros passos, na **etapa 1**, do algoritmo para levar o maior elemento (considerando *N=8*) para a posição *8*.

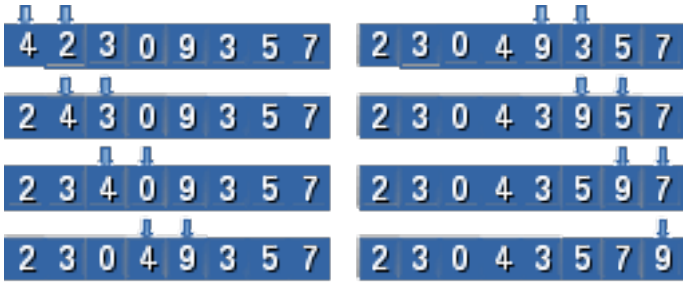

Fig. 2. Ilustração do Método do Pedregulho, coluna da esquerda as quatro primeiras "descida" do tipo `V[i]≤V[i+1]`.

Note que na primeira imagem (acima, esquerda), existem os *8* na ordem inicial `{4, 2, 3, 0, 9, 3, 5, 7}`. Cada passo de comparação dos pares de vizinhos está indicado por duas setas sobre os valores a serem comparados. Na representação seguinte, o vetor é novamente apresentado, eventualmente tendo sido trocados os (se `V[i]>V[i+1]`). Na passo *7*, usamos apenas uma seta para indicar que o maior dentre todos os elementos foi para a última posição.

Mas, como o objetivo é ordenar, não apenas descobrir o maior, podemos usar a mesma ideia (de modo *indutivo*): aplica-se o mesmo algoritmo (empurrar o maior para a última posição), mas considerando agora apenas os *7* primeiro elementos, pois o maior já está na posição *8*. Dessa forma, o *maior* elemento atual dentre os *7* primeiros irá para a última posição relativa, ou seja, esse passo resultará no *segundo maior* geral irá para a sétima posição geral (em *V[6]*)

Mais uma vez (em geral) o vetor **não** estará ordenado, mas podemos aplicar a mesma ideia mais *5* vezes:

- para `V[0]..V[5]` (maior deles para `V[5]`);
- para `V[0]..V[4]` (maior deles para `V[4]`);
- para `V[0]..V[3]` (maior deles para `V[3]`);
- para `V[0]..V[2]` (maior deles para `V[2]`);
- para `V[0]..V[1]` (maior deles para `V[1]`), sobrando em `V[0]` o **menor valor global**.

Desse modo obtém-se o vetor em ordem crescente! Abaixo apresentamos o algoritmo do *Pedregulho (Bolha)* em *C* e em *Python*.

Método de ordenação Bolha/Pedregulo em *C* e em *Python*

Exemplo em *C*

```c
// Ordena bolha/pedregulho
#include <stdio.h>
void imprimir (int X[], int n) {
  int i;
  printf("(");
  for (i=0; i<n; i++) printf("%3d ",
X[i]);
  printf(")\n");
```

Exemplo em *Python*

```python
# Ordena bolha/pedregulho
from __future__ import print_function; # imprime
sem mudar linha Python 2
def imprimir (X, n) :
  print("(", end="");
  for i in range(n) :
    print("%3d " % X[i], end="");
  print(")");
```

```c
  }
void ordPedregulho (int V[], int n) {
  int i, j, aux;
  for (i=0; i<n; i++)
    for (j=0; j<n-i-1; j++)
      if (V[j]>V[j+1]) { // troque-os
        aux = V[j]; // presevar V[j]
        V[j] = V[j+1];
        V[j+1] = aux; }
  }
int main (void) {
  int Y[] = { 4, 2, 3, 0, 9, 3, 5, 7 }; //
definir vetor dados
  imprimir(Y, 8);
  ordPedregulho(Y, 8);
  imprimir(Y, 8);
  }
```

```python
def ordPedregulho (V, n) :
  for i in range(n) :
    for j in range(n-i-1) :
      if (V[j]>V[j+1]) : # troque-os
        aux = V[j]; # preservar V[j]
        V[j] = V[j+1];
        V[j+1] = aux;

def main () :
  V = [4, 2, 3, 0, 9, 3, 5, 7]; # definir vetor
dados
  imprimir(V,8);
  ordPedregulho(V, 8);
  imprimir(V,8);

main();
```

Cód. 1. Método da Bolha em *C* e em *Python*.

Para saber mais: complexidade de algoritmos. Um tópico avançado no estudo de algoritmos é a área de **complexidade de algoritmos**, que resumidamente está interessado em determinar o quão rápido/lento pode ser determinado algoritmo.
Uma das análises é examinar a *complexidade do pior caso*, em qual configuração dos dados o algoritmo tem seu pior desempenho. A análise é feita a partir do "tamanho" da *entrada de dados* (digamos *n*), determinando-se uma função que limita superiormente seu tempo de execução (em termos da operação *central* para o algoritmo).
No caso do *método da Bolha*, a operação central é a comparação `V[j] > V[j+1]` que é realizada *n-1* vezes para "descer" o elemento "mais pesado" (ou "empurrar para a direita"), *n-2* vezes para "descer" o segundo elemento "mais pesado" e, assim por diante, até *1* vez para "descer" o *n-1*-ésimo elemento "mais pesado" (equivalente ao segundo mais "leve").
Portanto, no pior caso o *método da Bolha* realiza da ordem de n^2 operações, representado por $O(n^2)$ (pois a soma das comparações é limitada por n^2: $(n-1)+(n-2)+...+(n-(n-1)) = n(n-1)/2 = (n^2-n)/2$).

2. Método da Seleção Direta

A ideia do **método da Seleção Direta** é novamente deixar na última posição o maior elemento (ou o menor na primeira), mas diferentemente do *método da Bolha*, nesse mantém-se fixado a observação sempre no último elemento considerado, comparando-o com o primeiro, com o segundo e assim por diante. Considerando novamente um vetor com *8* elementos, após *7* comparações (e eventuais trocas), o maior deles estará na oitava posição (em *V[7]*).

Esse método está ilustrado na figura abaixo. Na primeira coluna estão os *7* passos para obter o maior elemento em `V[7]`. Na coluna 2 estão os *6* passos para obter o segundo maior para `V[6]`. Na coluna 3 estão os *5* passos para obter o terceiro maior para `V[5]` e na coluna 4 estão os *4* passos para obter o quarto maior para `V[4]`, seguindo-se de forma análoga para *3* e *2* (que não está representado na figura).

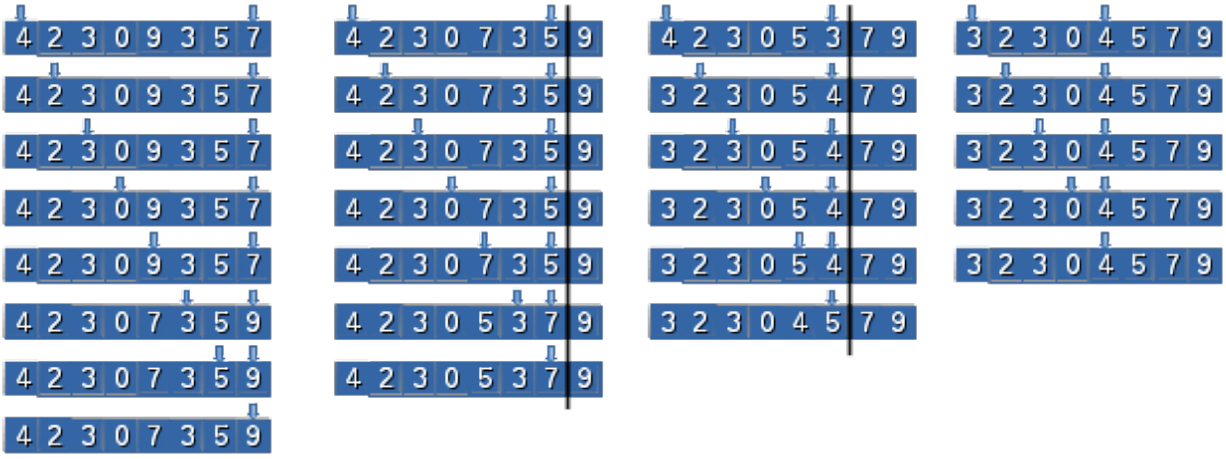

*Fig. 3. Ilustração do Método da Seleção Direta, coluna da esquerda os primeiros passos (na **etapa 1**), resultando em* `V[i]≤V[n-1]`, `i≤n-1` *(n=8).*

Abaixo está uma implementação do método *Seleção Direta* em *C* e em *Python*.

Método de ordenação Seleção Direta em *C* e em *Python*

Exemplo em *C*

```c
// Ordena Selecao Direta
#include <stdio.h>
void imprimir (int X[], int n) {
  int i;
  printf("(");
  for (i=0; i<n; i++) printf("%3d ",
X[i]);
  printf(")\n");
  }
void ordSelecao (int V[], int n) {
  int i, j, aux;
  for (i=n-1; i>0; i--)
    for (j=0; j<i; j++)
      if (V[j]>V[i]) { // troque-os
        aux = V[j]; // presevar V[j]
        V[j] = V[i];
        V[i] = aux;
        }
  }
int main (void) {
  int Y[] = { 4, 2, 3, 0, 9, 3, 5, 7 }; //
definir vetor dados
  imprimir(Y, 8);
  ordSelecao(Y, 8);
  imprimir(Y, 8);
  }
```

Exemplo em *Python*

```python
# Ordena Selecao Direta
from __future__ import print_function; # imprime
sem mudar linha Python 2
def imprimir (X, n) :
  print("(", end="");
  for i in range(n) :
    print("%3d " % X[i], end="");
  print(")");

def ordSelecao (V, n) :
  for i in range(n-1, 0, -1) :
    for j in range(i) :
      if (V[j]>V[i]) : # troque-os
        aux = V[j]; # preservar V[j]
        V[j] = V[i];
        V[i] = aux;

def main () :
  V = [4, 2, 3, 0, 9, 3, 5, 7]; # definir vetor
dados
  imprimir(V,8);
  ordSelecao(V, 8);
  imprimir(V,8);

main();
```

Cód. 2. Método da Seleção Direta em *C* e em *Python*.

Pequena melhoria no algoritmo. Note que os comandos para trocar os conteúdos entre `V[j]` e `V[i]` (as 3 atribuições subordinadas ao `if (V[j]>V[i])`) podem ser realizados várias vezes (no pior caso, quando os dados estão ordenados decrescente são feitas *n-1* trocas).

Poderíamos reduzir isso para uma única troca se utilizarmos mais duas variáveis auxiliares, `imax` e `max`, respectivamente, para armazenar o índice do maior valor e o maior valor (`max = V[imax]`) e, desse modo, ao final do laço `for j` faríamos apenas uma troca (entre `V[i]` e `V[imax]`). Nessa variante do algoritmo, o número de passos de comparações é o mesmo, entretanto o número de trocas (geralmente) seria bem menor!**Para saber mais: complexidade de algoritmos**. Novamente examinando a *operação central* do *método da Seleção Direta*, `V[j] > V[i]`, esse algoritmo também é $O(n^2)$, pois na primeira passada realiza *n-1* comparações para selecionar o maior, depois *n-2* para selecionar o segundo maior e, assim por diante até o *1* comparação para selecionar *n-1*-ésimo maior, ou seja, somando todas as comparações $(n-1)+(n-2)+...+(n-(n-1)) = n(n-1)/2 = (n^2-n)/2$).

3. Método da Inserção Direta

A ideia do **método da Inserção Direta** é um pouco diferente da ideia comum aos *métodos da Bolha* e *Seleção Direta*, naqueles, ao final de execução do laço interno `for j`, o maior elemento (dentre os analisados) ficava em sua posição "definitiva", no *Inserção* não. A ideia do **Inserção** é (na **etapa i de ordenação**): suponha que `V[0]` até `V[i-1]` esteja ordenado, então podemos estender a ordem localizando o primeiro elemento, da direita para a esqueda, que **não** seja maior que `V[i]`, digamos que seja o `V[j]`, então move-se todos eles uma posição para a sua direita, abrindo a posição `V[j]` para **inserir** o `V[i]`. Esse princípio é ilustrado abaixo com `j=3`:

1. `V[0]`$\leq$`V[1]`$\leq$`V[2]` odernados;
2. `V[3]` libera seu espaço (cópia em `aux`);
3. todos os elementos com valor estritamente maior que `aux` "dão um passo para a direita";
4. a posição `i=0` é liberada para inserir ali `aux` (estendendo a ordenação para 4 elementos).

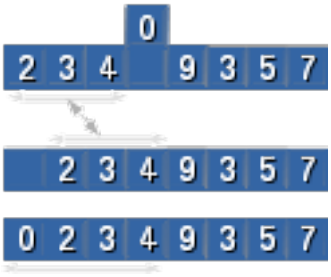

*Fig. 4. Ilustração dos "deslocamentos" para **inserir** a chave (inicialmente) na posição `V[3]` para a posição correta (`V[0]`$\leq$`V[1]`$\leq$`V[2]`$\leq$`V[3]`.*

Esse método está ilustrado na figura abaixo. Na primeira coluna estão as *3* instruções para obter uma ordenação com os *2* primeiros elementos. Na coluna 2 estão as *4* instruções para obter ordenação dos *4* primeiros elementos e na coluna 3 estão as *5* instruções para obter ordenação dos *5* primeiros elementos.

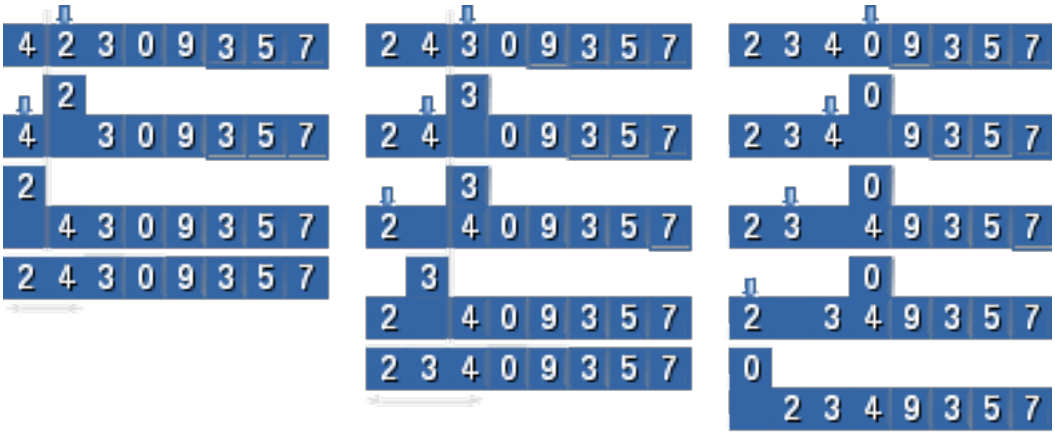

Fig. 5. Ilustração do Inserção Direta: a primeira coluna apresenta a ***etapa 1 de ordenação****, quando a chave é* `2` *na posição* `V[1]`*; na segunda a chave é* `3` *na posição* `V[2]`*.*

É importante notar que se os dados estiverem já ordenados, serão realizadas apenas `n=7` operações! De modo geral, se a *etapa de ordenação i* resultar em ordenação completa dos dados, deste ponto em diante, serão feitas apenas *n-i-1* comparações e nenhuma troca (!). Pode-se usar uma bandeira para identificar tal fato e finalizar o algoritmo.
Por exemplo, na última coluna da figura 5, se o valor em `V[4]` fosse `5` (ou qualquer outro valor maior), haveria apenas três instruções (independe do valor *5*): copiar `V[3]` para `aux`, comparar `aux` com `V[2]` e, como `aux` é maior (ou igual), finalizaria, copiando `aux` novamente para `V[3]`.

Abaixo está uma implementação do método *Inserção Direta* em *C* e em *Python*.

Método de ordenação Inserção Direta em *C* e em *Python*

Exemplo em *C*

```c
// Ordena Insercao Direta
#include <stdio.h>
void imprimir (int X[], int n) {
  int i;
  printf("(");
  for (i=0; i<n; i++) printf("%3d ", X[i]);
  printf(")\n");
  }
void ordInsercao (int V[], int n) {
  int i, j, aux;
  for (i=1; i<n; i++) {
    aux = V[i]; // "abrir espaco" copiando V[i]
    j = i-1;
    while (j > -1 && V[j] > aux) { // mova para direita os maiores que aux
      V[j+1] = V[j]; // mover para direita V[j]
      j -= 1;
      }
```

Exemplo em *Python*

```python
# Ordena Insercao Direta
from __future__ import print_function; # imprime sem mudar linha Python 2
def imprimir (X, n) :
  print("(", end="");
  for i in range(n) :
    print("%3d " % X[i], end="");
  print(")");

def ordInsercao (V, n) :
  for i in range(1, n) :
    aux = V[i]; # "abrir espaco" copiando V[i]
    j = i-1;
    while (j > -1 and V[j] > aux) : # "mova" para direita os maiores que aux
      V[j+1] = V[j]; # mover para direita V[j]
      j -= 1;
    V[j+1] = aux; # coloque aux na posicao correta
```

```c
    V[j+1] = aux; // coloque aux na
posicao correta
    }
  }
int main (void) {
  int Y[] = { 4, 2, 3, 0, 9, 3, 5, 7 }; //
definir vetor dados
  imprimir(Y, 8);
  ordInsercao(Y, 8);
  imprimir(Y, 8);
  }
```

```python
def main () :
  V = [4, 2, 3, 0, 9, 3, 5, 7]; # definir
vetor dados
  imprimir(V,8);
  ordInsercao(V, 8);
  imprimir(V,8);

main();
```

Cód. 3. Método da Inserção Direta em *C* e em *Python*.

Para saber mais: complexidade de algoritmos. Novamente examinando a *operação central* do *método da Inserção Direta*, `V[j] > aux`, esse algoritmo também é $O(n^2)$, pois na primeira passada realiza, no pior caso, *1* comparação para localizar a posição do elemento *1*, depois *2* comparações para localizar a posição do elemento *2*, depois *3* comparações para localizar a posição do elemento *3* e, assim por diante até *n-1* comparações para localizar a posição do elemento *n-1*, ou seja, somando todas as comparações $1 + 2 + 3 + ... + (n-1) = n(n-1)/2 = (n^2-n)/2$.
Mas essa é a análise de *pior caso*, a análise oposta concluiria que, no melhor caso, que esse algritmo gasta apenas *n-1* comparações (quando o vertor está ordenado).
Bem, em resumo é isso. Experimente copiar esses códigos, tente algumas modificações, teste até estar seguro de ter compreendido.

Leônidas de Oliveira Brandão
http://line.ime.usp.br

Alterações