Ordenação: Introdução e métodos elementares

Algoritmos e Estruturas de Dados II

Ordenação

- Objetivo:
  - Rearranjar os itens de um vetor ou lista de modo que suas chaves estejam ordenadas de acordo com alguma regra

- Estrutura:

```
typedef int chave_t;
struct item {
     chave_t chave;
     /* outros componentes */
};
```

Critérios de Classificação

- Localização dos dados:
  - Ordenação interna
    - Todas os dados estão em memória principal (RAM)
  - Ordenação externa
    - Memória principal não cabe todos os dados
    - Dados armazenados em memória secundária (disco)

Critérios de Classificação

- Estabilidade:
  - Método é *estável* se a ordem relativa dos registros com a mesma chave não se altera após a ordenação.

Adams	A
Black	B
Brown	D
Jackson	B
Jones	D
Smith	A
Thompson	D
Washington	B
White	C
Wilson	C

Adams	A
Smith	A
Washington	B
Jackson	B
Black	B
White	C
Wilson	C
Thompson	D
Brown	D
Jones	D

Adams	A
Smith	A
Black	B
Jackson	B
Washington	B
White	C
Wilson	C
Brown	D
Jones	D
Thompson	D

Critérios de Classificação

- Adaptabilidade:
  - Um método é *adaptável* quando a sequencia de operações realizadas depende da entrada
  - Um método que sempre realiza as mesmas operações, independende da entrada, é *não adaptável.*

Critérios de Classificação

- Uso da memória:
  - *In place:* ordena sem usar memória adicional ou usando uma quantidade constante de memória adicional
  - Alguns métodos precisam duplicar os dados

Critérios de Classificação

- Movimentação dos dados:
  - Direta: estrutura toda é movida

```
// struct item a;
// struct item b;
struct item aux = a;
a = b;
b = aux;
```

  - Indireta: apenas as chaves são acessadas e ponteiros para as estruturas são rearranjados

```
// struct item *a;
// struct item *b;
struct item *aux = *a;
a = b;
b = aux;
```

Critérios de Avaliação

- Seja n o número de registros em um vetor, considera-se duas medidas de complexidade:
  - Número de comparações $C(n)$ entre as chaves
  - Número de trocas ou movimentações $M(n)$ de itens

```
#define troca(A, B) {struct item c = A; A = B; B = c;}
void ordena(struct item *v, int n) {
   int i, j;
   for(i = 0; i < n-1; i++) {
      for(j = n-1; j > i; j--) {
         if(v[j-1].chave > v[j].chave) /* comparações */
            troca(v[j-1], v[j]); /* trocas */
      }
   }
}
```

Ordenação por Seleção

- Procura o n-ésimo menor elemento do vetor
- Troca do n-ésimo menor elemento com o elemento na n-ésima posição
- Repete até ter colocado todos os elementos em suas posições

- Elementos são movimentados apenas uma vez

Ordenação por Seleção

```
void selecao(struct item *v, int n){
  int i, j, min;
  for(i = 0; i < n - 1; i++) {
    min = i;
    for(j = i + 1 ; j < n; j++) {
      if(v[j].chave < v[min].chave)
        min = j;
    }
    troca(v[i], v[min]);
  }
}
```

E	X	E	M	P	L	O
E	X	E	M	P	L	O
E	E	X	M	P	L	O
E	E	L	M	P	X	O
E	E	L	M	P	X	O
E	E	L	M	O	X	P
E	E	L	M	O	P	X
E	E	L	M	O	P	X

Ordenação por Seleção: Complexidade

- Comparações – *C(n)*:

$$
\begin{aligned}
C(n) &= \sum_{i=0}^{n-2}(n-i-1) = \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 \\
&= n(n-1) - \frac{(n-2)(n-1)}{2} - (n-1) \\
&= \frac{n^2 - n}{2} = O(n^2)
\end{aligned}
$$

- Movimentações – *M(n)*:

$$M(n) = 3(n-1)$$

Ordenação por Seleção

- Vantagens:
  - Custo linear no tamanho da entrada para o número de movimentos de registros – a ser utilizado quando há registros muito grandes

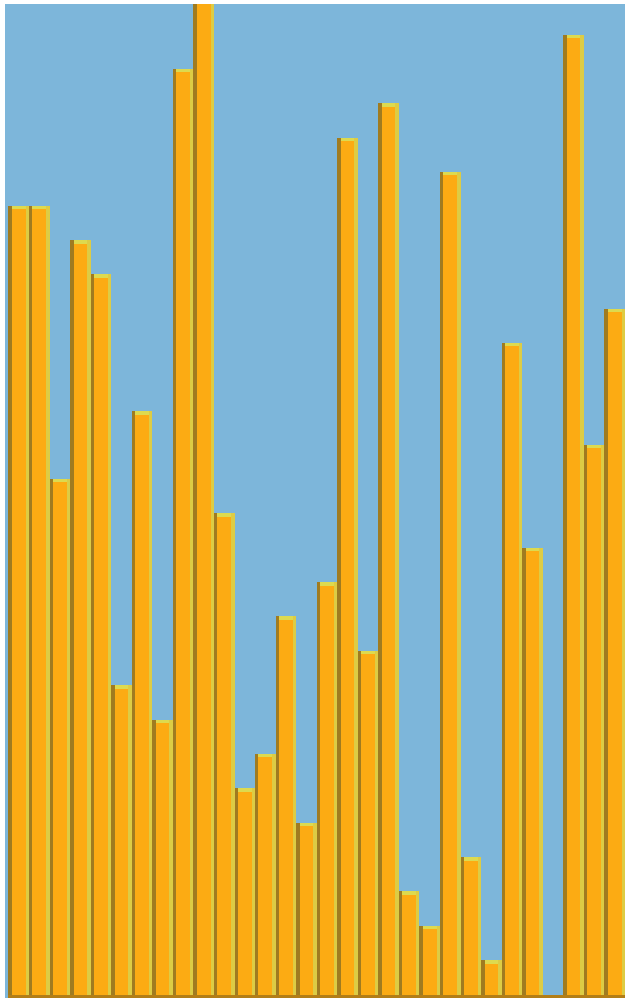
- Desvantagens:
  - Não adaptável
    - Não importa se o arquivo está parcialmente ordenado
  - Algoritmo não é estável

Ordenação por Inserção

- Algoritmo utilizado pelo jogador de cartas
  - Jogador mantém as cartas em sua mão ordenadas
  - Quando compra ou recebe uma nova carta, o jogador encontra qual posição ela deve ocupar em sua mão

- Implementação para vetores:
  - Mantemos os elementos entre zero e i-1 ordenados
    - Note que o arranjo entre formado por um elemento está ordenado, por definição
  - Achamos a posição do i-ésimo elemento e inserimos ele entre os i-1 que já estavam ordenados
  - O programa repete esse passo até ordenar todos os elementos

Método Inserção



Ordenação por Inserção

```
void insercao(struct item *v, int n) {
  int i, j;
  struct item aux;
  for(i = 1; i < n; i++) {
    for(j = i; j >= 0; k--) {
      if(v[j-1].chave > v[j].chave) {
         troca(v[j-1], v[j]);
      }
    }
  }
}
```

E	X	E	M	P	L	O
E	X	E	M	P	L	O
E	E	X	M	P	L	O
E	E	M	X	P	L	O
E	E	M	P	X	L	O
E	E	L	M	P	X	O
E	E	L	M	O	P	X

Ordenação por Inserção – Melhorado

```
void insercao(struct item *v, int n) {
  int i, j;
  struct item aux;
  for(i = 1; i < n; i++) {
    aux = v[i];
    j = i - 1;
    while((j >= 0) && (aux.chave < v[j].chave)) {
      v[j + 1] = v[j];
      j--;
    }
    v[j + 1] = aux;
  }
}
```

E	X	E	M	P	L	O
E	X	E	M	P	L	O
E	E	X	M	P	L	O
E	E	M	X	P	L	O
E	E	M	P	X	L	O
E	E	L	M	P	X	O
E	E	L	M	O	P	X

Ordenação por Inserção: Complexidade

- Comparações – *C(n)*:
  - Anel interno: i-ésima iteração, valor de C_i
    - melhor caso: $C_i = 1$
    - pior caso: $C_i = i$
    - caso médio: $C_i = (1 + 2 + 3 + \ldots + i) / i$

$$C_i = \frac{1}{i}\sum_{k=1}^{i} k = \frac{1}{i}\left(\frac{i(i+1)}{2}\right) = \frac{i+1}{2}$$

  - Para o caso médio, assumimos que todas as permutações de entrada são igualmente prováveis.

Ordenação por Inserção: Complexidade

- Comparações – $C(n)$:
  - Anel externo:

$$\sum_{i=1}^{n-1} C_i$$

  - Complexidade total:
    - Melhor caso (itens já estão ordenados)

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 = O(n)$$

    - Pior caso (itens em ordem reversa):

$$C(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

Ordenação por Inserção: Complexidade

- Comparações – *C(n)*:
  - Caso médio:

$$C(n) = \sum_{i=1}^{n-1} \frac{i+1}{2} = \frac{1}{2}\left(\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \right) = \frac{1}{2}\left(\frac{n(n-1)}{2} + (n-1) \right)$$

$$= \left(\left(\frac{n^2 - n}{4} \right) + \left(\frac{n-1}{2} \right) \right) = \frac{n^2}{4} + \frac{n}{4} - \frac{1}{2} = O(n^2)$$

Ordenação por Inserção: Exemplos

- Melhor Caso:

1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6

- Pior Caso:

6	5	4	3	2	1
5	6	4	3	2	1
4	5	6	3	2	1
3	4	5	6	2	1
2	3	4	5	6	1
1	2	3	4	5	6

Ordenação por Inserção

- Vantagens:
  - Laço interno é eficiente, inserção é adequado para ordenar vetores pequenos
  - É o método a ser utilizado quando o arquivo está “quase” ordenado
  - É um bom método quando se deseja adicionar poucos itens a um arquivo ordenado, pois o custo é linear
  - O algoritmo de ordenação por inserção é **estável**

- Desvantagens:
  - Número de comparações tem crescimento quadrático
  - Alto custo de movimentação de elementos no vetor

Ordenação pelo Método da Bolha

- Ideia
  - Passa no arquivo e troca elementos adjacentes que estão fora de ordem
  - Repete esse processo até que o arquivo esteja ordenado
- Algoritmo
  - Compara dois elementos adjacentes e troca de posição se estiverem fora de ordem
  - Quando o maior elemento do vetor for encontrado, ele será trocado até ocupar a última posição
  - Na segunda passada, o segundo maior será movido para a penúltima posição do vetor, e assim sucessivamente

Ordenação pelo Método da Bolha

E	X	E	M	P	L	O
E	E	X	M	P	L	O
E	E	M	X	P	L	O
E	E	M	P	X	L	O
E	E	M	P	L	X	O
E	E	M	P	L	O	X
E	E	M	P	L	O	X

1ª passada

E	E	M	P	L	O	X
E	E	M	L	P	O	X
E	E	M	L	P	O	X

2ª passada

Ordenação pelo Método da Bolha

```
void bolha(struct item *v, int n) {
  int i, j;
  for(i = 0; i < n-1; i++) {
    for(j = 1; j < n-i; j++) {
      if(v[j].chave < v[j-1].chave)
        troca(v[j-1], v[j]);
    }
  }
}
```

Ordenação pelo Método da Bolha

```
void bolha(struct item *v, int n) {
  int i, j;
  for(i = 0; i < n-1; i++) {
    for(j = 1; j < n-i; j++) {
      if(v[j].chave < v[j-1].chave)
        troca(v[j-1], v[j]);
    }
  }
}
```

E	X	E	M	P	L	O
E	E	M	P	L	O	X
E	E	M	L	O	P	X
E	E	L	M	O	P	X
E	E	L	M	O	P	X
E	E	L	M	O	P	X

E	E	L	M	O	P	X
E	E	L	M	O	P	X

Ordenação pelo Método da Bolha: Complexidade

```
void bolha(struct item *v, int n) {
  int i, j;
  for(i = 0; i < n-1; i++) {
    for(j = 1; j < n-i; j++) {
      if(v[j].chave < v[j-1].chave)
        troca(v[j-1], v[j]);
    }
  }
}
```

$$
\begin{aligned}
C(n) &= \sum_{i=0}^{n-2}\sum_{j=1}^{n-i} 1 = \sum_{i=0}^{n-2}(n-i-1) = \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 \\
&= n(n-1) - \frac{(n-2)(n-1)}{2} - (n-1) \\
&= \frac{n^2 - n}{2} = O(n^2)
\end{aligned}
$$

Ordenação pelo Método da Bolha: Complexidade

- Movimentações – $M(n)$:
  - Pior caso (elementos em ordem decrescente):

$$M(n) = \sum_{i=0}^{n-2}\sum_{j=1}^{n-i} 3 = 3\sum_{i=0}^{n-2}(n-i-1) = 3\frac{n^2-n}{2} = O(n^2)$$

Ordenação pelo Método da Bolha

- Vantagens
  - Algoritmo simples
  - Algoritmo estável
- Desvantagens
  - Não adaptável
  - Muitas trocas de itens

Exercícios

1. Dê um exemplo de um vetor com N elementos que maximiza o número de vezes que o mínimo é atualizado no método de ordenação seleção.
2. Mostre um exemplo de entrada que demonstra que o método de ordenação seleção não é estável.
3. O método da bolha não é adaptável, altere o código para que ele se torne adaptável.
4. Qual dos métodos: bolha, inserção e seleção executa menos comparações para um vetor de entrada contendo valores idênticos.

Variação Bolha: Ordenação Par-Ímpar

```
void ParImpar(struct item *v, int n) {
   int ordenado = 0;
   while(!ordenado) {
      ordenado = 1;
      for(int i = 0; i < n-1; i += 2)
         if(v[i] > v[i+1]) {
            troca(v[i], v[i+1]);
            ordenado = 0;
         }
      for(int i = 0; i < n-1; i += 2)
         if(a[i] > a[i+1]) {
            troca(v[i], v[i+1]);
            ordenado = 0;
         }
   }
}
```

Visualização de Algoritmos de Ordenação

- http://www.sorting-algorithms.com