Algoritmos de Ordenação

InsertionSort

(Fonte: Material adaptado dos Slides do prof. Monael.)

Problema Fundamental

- O problema da ordenação:
  - Permutar (ou rearranjar) os elementos de um vetor v[0..n-1] de tal modo que eles fiquem em ordem crescente:
  - v[0] <= v[1] <= ... <= v[n-1]

0	1	2	3	4	5	6	7	8	9	10
111	999	222	999	333	888	444	777	555	666	555

111	222	333	444	555	555	666	777	888	999	999

Ordenação por Inserção

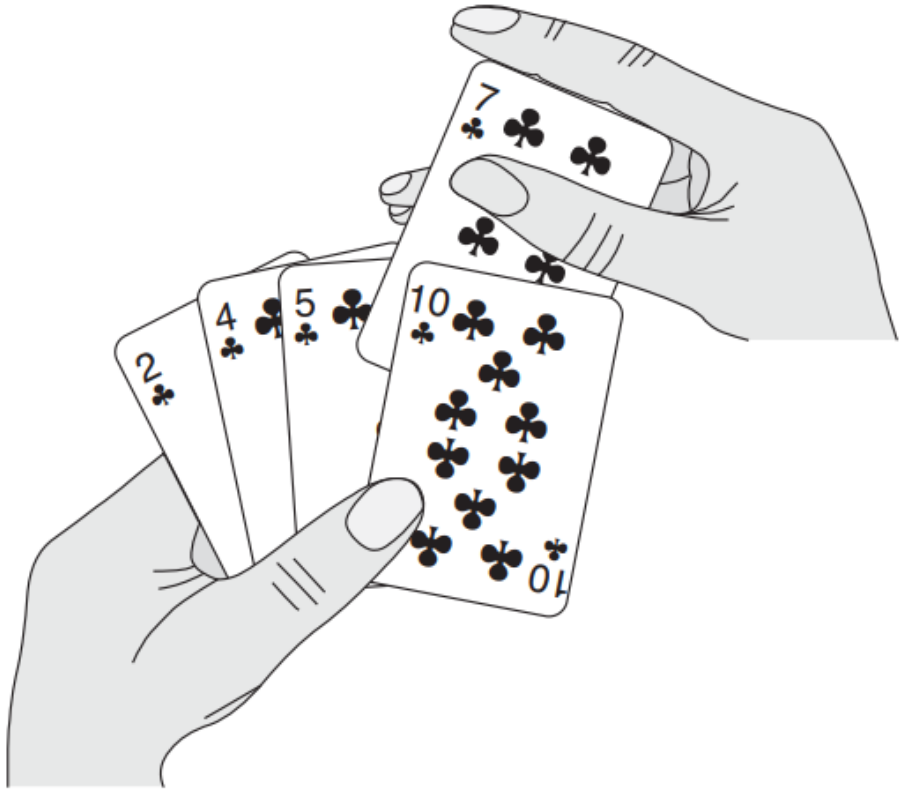
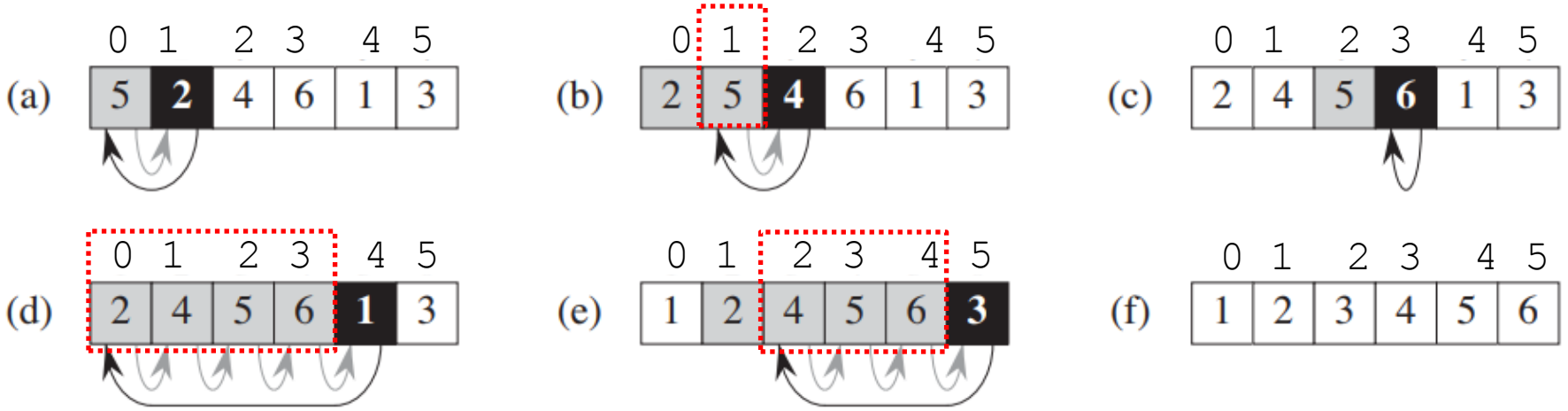


Figure 2.1 Sorting a hand of cards using insertion sort.

Ordenação por Inserção



Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```

Ordenação por Inserção

```
                                                    // Consumo de Tempo
OrdenaPorInsercao (vetor A, int n) {                // Rascunho:
1. for (j = 1; j < n; j++) {                        // 1. O(n)
2.      chave = A[j]                                // 2. O(1) x O(n)
3.// Insere A[j] na seq ordenada A[0..j-1]         //
4.      i = j - 1                                   // 4. O(1) x O(n)
5.      while (i >= 0 && chave < A[i]) {            // 5. O(n) x O(n)
6.          // desloca para direita                 //
7.          A[i+1] = A[i]                           // 7. O(1) x O(n^2)
8.          i = i - 1                               // 8. O(1) x O(n^2)
9.      }                                           //
10.     A[i+1] = chave                              // 10. O(1) x O(n)
}
                                                    // Total: O(n^2)
```

Ordenação por Inserção

```
   OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
   }
```

i j chave=2

	0	1	2	3	4	5
(a)	5	2	4	6	1	3

Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```

Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```

i j chave=2

	0	1	2	3	4	5
(a)	5	5	4	6	1	3

Ordenação por Inserção

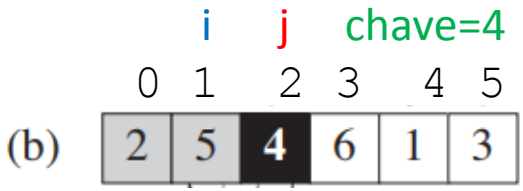
```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```

i j chave=2

	0	1	2	3	4	5
(a)	2	5	4	6	1	3

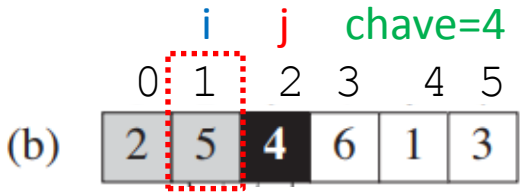
Ordenação por Inserção

```
   OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
   }
```



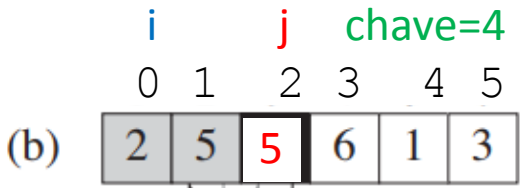
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



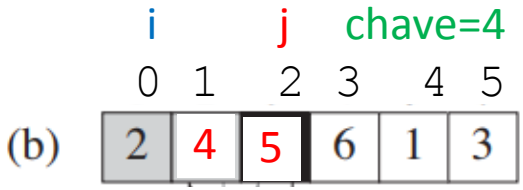
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



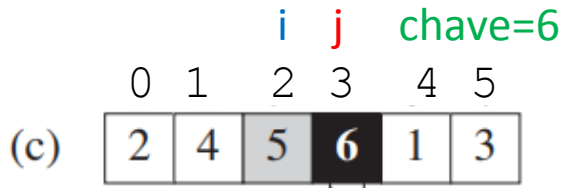
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



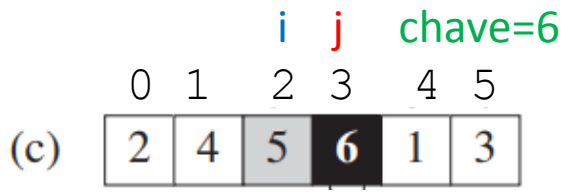
Ordenação por Inserção

```
   OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.     chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.     i = j - 1
5.     while (i >= 0 && chave < A[i]) {
6.         // desloca para direita
7.         A[i+1] = A[i]
8.         i = i - 1
9.     }
10.    A[i+1] = chave
   }
```



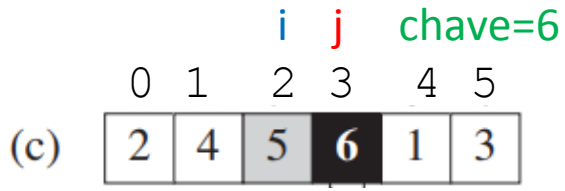
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



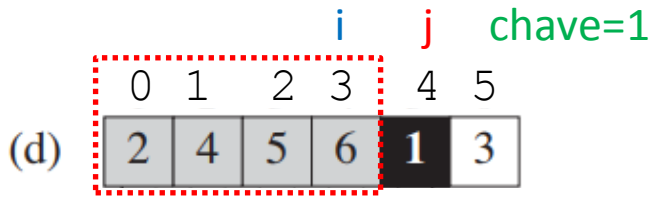
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1.  for (j = 1; j < n; j++) {
2.      chave = A[j]
3.  // Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```

i j chave=1

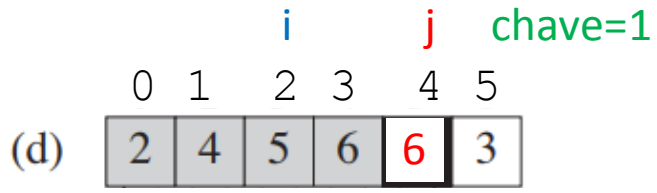
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



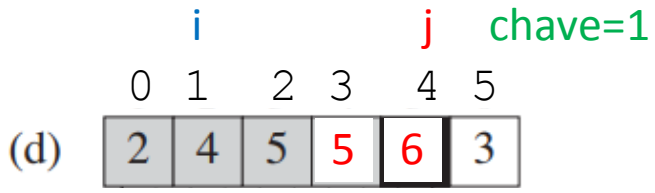
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



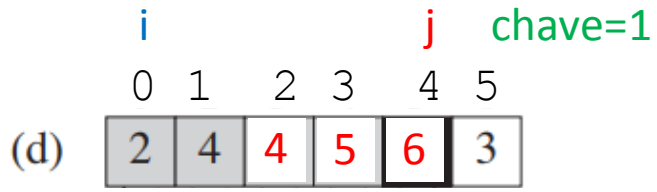
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



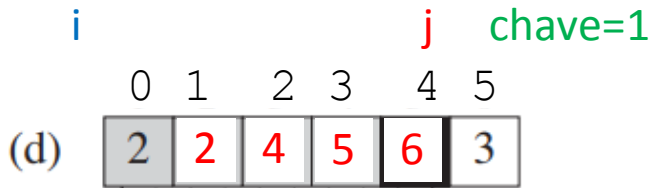
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



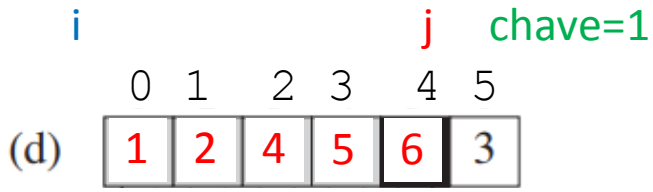
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



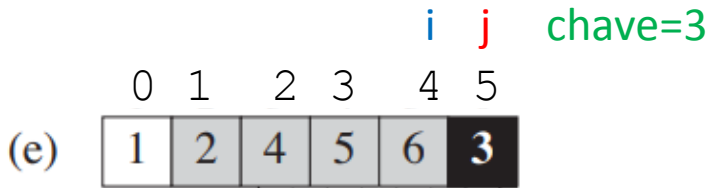
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



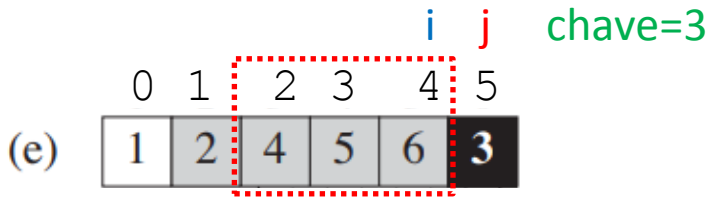
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



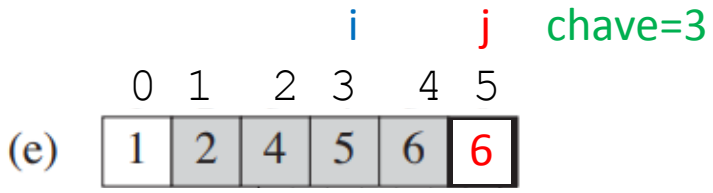
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



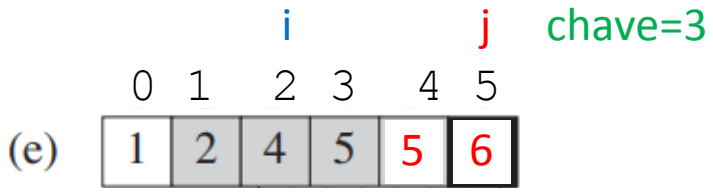
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



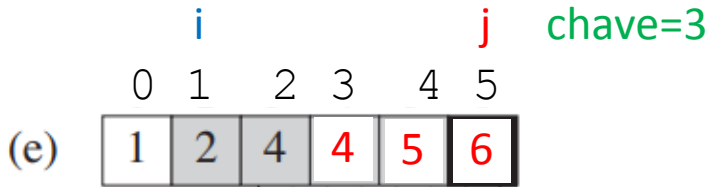
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



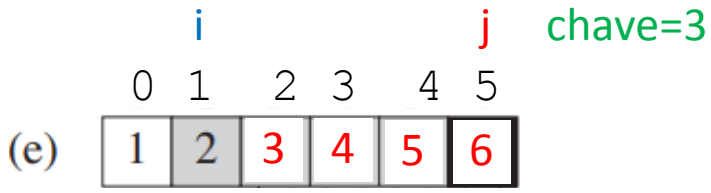
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



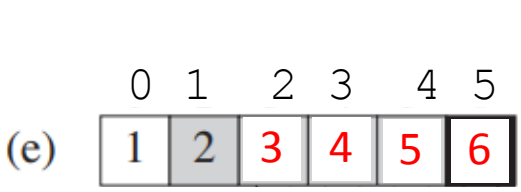
Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



Ordenação por Inserção

(Rascunho:)

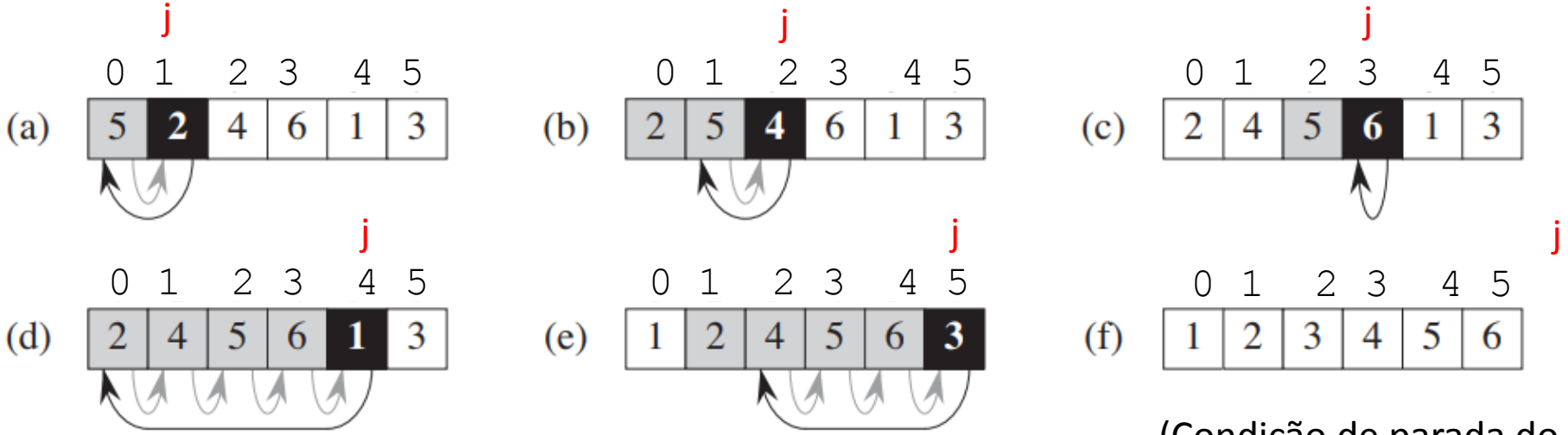
- O algoritmo é correto e é baseado no seguinte invariante:
  - No início de cada iteração (linha 1), temos que o subvetor A[0..j-1] está ordenado

```
   OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.     chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.     i = j - 1
5.     while (i >= 0 && chave < A[i]) {
6.         // desloca para direita
7.         A[i+1] = A[i]
8.         i = i - 1
9.     }
10.    A[i+1] = chave
   }
```

Ordenação por Inserção

(Rascunho:)

- O algoritmo é correto e é baseado no seguinte invariante:
  - No início de cada iteração (linha 1), temos que o subvetor A[0..j-1] está ordenado



(Condição de parada do algoritmo: A[0..n-1])

Ordenação por Inserção

(Rascunho:)

- O algoritmo é correto e é baseado no seguinte invariante:
  - No início de cada iteração (linha 1), temos que o subvetor A[0..j-1] está ordenado.
  - A condição de parada do algoritmo (j = n) garante que após executarmos o algoritmo, temos que A[0..n-1] está ordenado.

Análise de Algoritmos

- Consumo de tempo
  - Qual é a complexidade de tempo do algoritmo?
  - Resp: O(n^2)

- Correção do algoritmo
  - O algoritmo é correto?
  - Resp: Sim, é baseado em invariante.

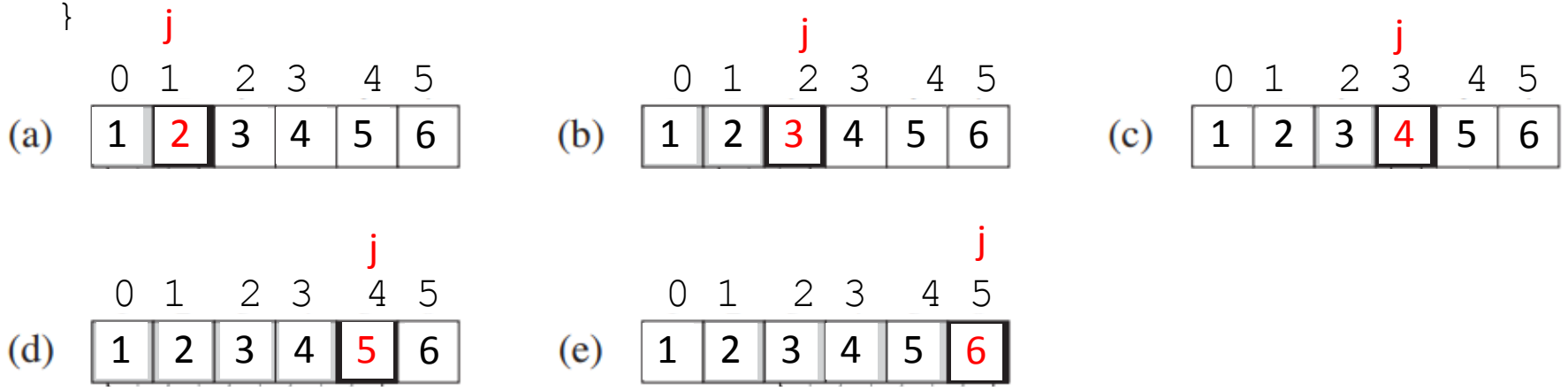
Algoritmos Estudados

- Bubble Sort
  - Consumo de Tempo no Pior Caso: $O(n^2)$
  - Consumo de Tempo no Melhor Caso: $O(n^2)$
- Selection Sort
  - Consumo de Tempo no Pior Caso: $O(n^2)$
  - Consumo de Tempo no Melhor Caso: $O(n^2)$
- Insertion Sort
  - Consumo de Tempo no Pior Caso: $O(n^2)$
  - Consumo de Tempo no Melhor Caso: $O(n)$

(vetor ordenado)

Ordenação por Inserção

```
OrdenaPorInsercao (vetor A, int n) {
1. for (j = 1; j < n; j++) {
2.      chave = A[j]
3.// Insere A[j] na seq ordenada A[0..j-1]
4.      i = j - 1
5.      while (i >= 0 && chave < A[i]) {
6.          // desloca para direita
7.          A[i+1] = A[i]
8.          i = i - 1
9.      }
10.     A[i+1] = chave
}
```



Conseguimos fazer melhor?

- Bubble Sort
  - Consumo de Tempo no Pior Caso: $O(n^2)$
- Selection Sort
  - Consumo de Tempo no Pior Caso: $O(n^2)$
- Insertion Sort
  - Consumo de Tempo no Pior Caso: $O(n^2)$

Conseguimos fazer melhor?

- Sim, O(n log n) seria o "melhor" algoritmo.

Ex. O(n^2) --> "Ó-grande" (upper bound)

(Problema Fundamental da Ordenação)

Ω(n log n) --> "ômega" (lower bound)

Lower Bound do Problema de Ordenação

- Até agora, apresentamos algoritmos que ordenam *n* números em tempo $O(n^2)$. Por enquanto, esse é o nosso *upper bound* para o problema da ordenação baseado em comparações.
- Seria possível calcular um *lower bound* para esse problema?
- Em outras palavras, desejamos encontrar um limite inferior teórico para esse problema, isto é, a mínima complexidade de tempo de quaisquer de suas resoluções algorítmicas.

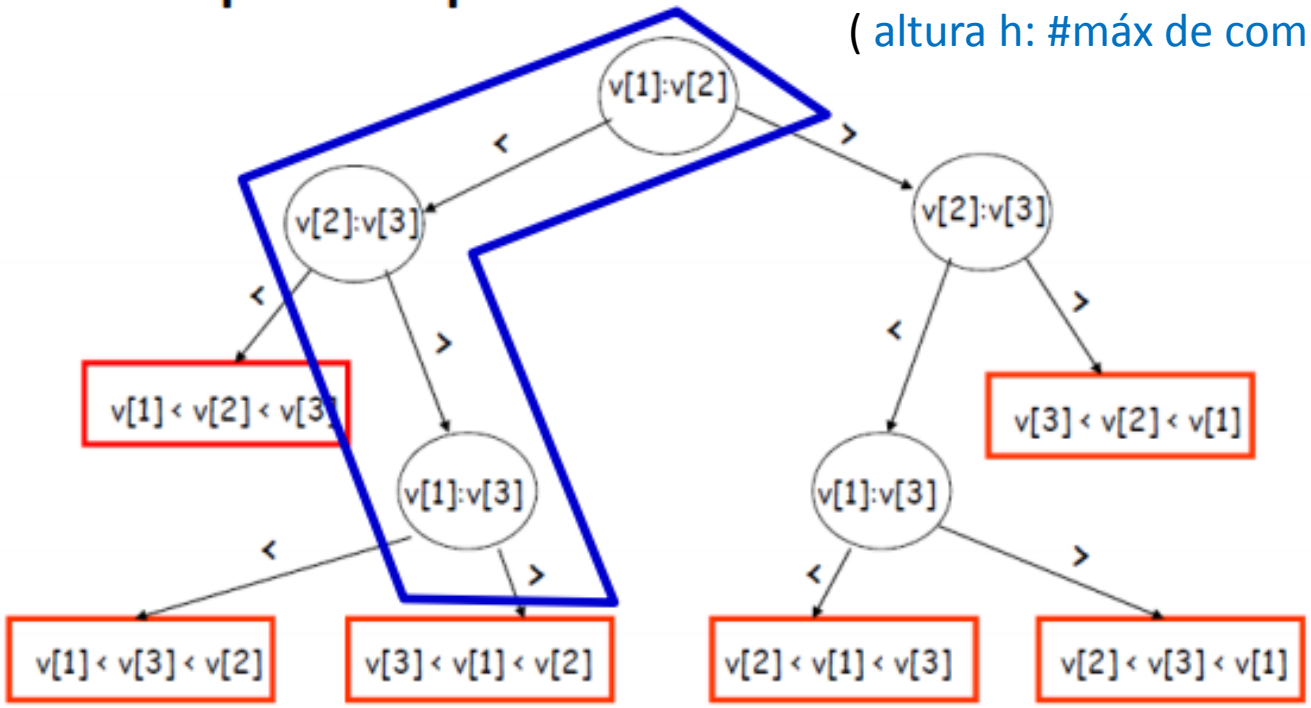
Árvore de Comparações

- Qualquer algoritmo de ordenação baseado em comparações pode ser representado em uma árvore binária.
- Na raiz fica a primeira comparação realizada entre dois elementos; nos filhos, as comparações subsequentes.
- Assim, as folhas dessa árvore representam as possíveis soluções do problema.

Árvore de Comparação

- A altura h da árvore é o número máximo de comparações que o algoritmo realiza, ou seja, o seu tempo de pior caso

(altura h: #máx de comparações)



(n! folhas)

Generalização

- Na ordenação de *n* elementos, há então *n!* possíveis resultados, que correspondem às permutações desses elementos.
- Portanto, qualquer árvore binária de comparações terá no mínimo *n!* folhas.
- A árvore mínima de comparações tem exatamente *n!* folhas.
- Supondo que a altura dessa árvore seja *h*, então *LB(n) = h*, onde *LB(n)* é o *lower bound* de tempo para a ordenação de *n* elementos.

Generalização

- Sabemos que a quantidade de folhas de uma árvore binária de altura h é $\leq 2^h$.
- Portando, $n! \leq 2^h$.
- Ou seja, $h \geq log_2\ n!$
- Conclui-se que $LB(n) \geq log_2\ n!$

Aproximação de Stirling

- O valor numérico de $n!$ pode ser calculado por multiplicação repetida se n não for grande demais. É isto que as calculadoras fazem. O maior fatorial, que a maioria das calculadoras suportam é 69!, porque $70! > 10^{100}$.
- Quando n é grande demais, $n!$ pode ser calculado com uma boa precisão usando a **aproximação de Stirling**:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Cálculo do Lower Bound

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$$n! \approx (2\pi n)^{\frac{1}{2}} n^n e^{-n}$$

$$\log_2 n! \approx \log_2 (2\pi)^{\frac{1}{2}} + \log_2 (n)^{\frac{1}{2}} + \log_2 n^n + \log_2 e^{-n}$$

$$\log_2 n! \approx \left(\frac{\log_2 (2\pi)}{2}\right) + \left(\frac{\log_2 (n)}{2}\right) + \log_2 n^n + \log_2 e^{-n}$$

$$\log_2 n! \approx \left(\frac{\log_2 (2\pi)}{2}\right) + \left(\frac{\log_2 (n)}{2}\right) + (n \cdot \log_2 n) - n \cdot \log_2 e$$

$$\log_2 n! \approx O(1) + O(\log_2 n) + O(n \cdot \log_2 n) - O(n)$$

Cálculo do Lower Bound

$$\log_2 n! \approx O(1) + O(\log_2 n) + O(n \cdot \log_2 n) - O(n)$$

$$LB(n) \geq log_2\, n!$$

então

$$LB(n) = \Omega(n \cdot \log_2 n)$$

Desta forma, se encontrarmos um algoritmo que resolva a ordenação em tempo ***O(n·log$_2$n)***, ele será ótimo, e esse problema estará computacionalmente resolvido.

Alguns Algoritmos de Ordenação

- Exemplos de algoritmos O(n log n):
  - Merge Sort
    - "Intercalação"
  - Quick Sort
    - "Particionamento"
  - Heap Sort
    - estrutura de dados "Heap"

Ordenação por Seleção

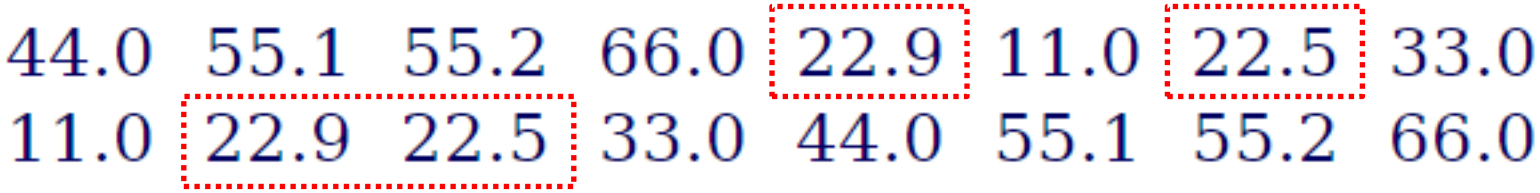
//Este algoritmo usa a seguinte estratégia: seleciona o menor elemento do vetor,
// depois o segundo menor, depois o terceiro menor, e assim por diante.

```
OrdenaPorSelecao (vetor A, int n) {
1. for (i = 0; i < n-1; i++) {
2.      min = i
3.      for (j = i+1; j < n; j++) {
4.          if (v[j] < v[min]) {
5.              min = j
6.      }
7.      tmp = v[i]
8.      v[i] = v[min]
9.      v[min] = tmp
10. }
```

(Fonte: https://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html)

Algoritmos "Estáveis"

- Um algoritmo de ordenação é estável se não altera a posição relativa dos elementos que têm o mesmo valor.
  - Em outras palavras, um algoritmo estável de ordenação mantém a ordem de inserção dos dados no caso de empates.
- Exemplo: ordenação estável da parte inteira



(Fonte: https://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html)

Algoritmos "Estáveis"

- Verifique você mesmo:
  - 1) O algoritmo de inserção é estável?
  - 2) O algoritmo de seleção é estável?

(Dica: verifique os dois algoritmos para o problema do Balizamento com a "entrada 3": Marcia, Ligia e Graciete.)