Ordenação de estruturas lineares

Definições e algoritmos

Prof. Marcelo de Souza

45RPE – Resolução de Problemas com Estruturas de Dados
Universidade do Estado de Santa Catarina

Ordenação de estruturas lineares

Conceitos básicos

Ordenar uma estrutura consiste em rearranjar seus elementos, respeitando uma dada ordem.

- Geralmente, usamos ordem crescente ou ordem decrescente.

Ordenar uma estrutura consiste em rearranjar seus elementos, respeitando uma dada ordem.

- Geralmente, usamos ordem crescente ou ordem decrescente.

Podemos ordenar qualquer coleção de itens, desde que sejam comparáveis uns aos outros.

- **Números** (ordem crescente/decrescente) ou **strings** (ordem alfabética), já comparáveis;
- **Livros** (ordem dada pelo título, autor, ano ou páginas, por exemplo).
  - Necessário implementar a interface `Comparable` e o método `compareTo`.

Ordenação de estruturas lineares

Conceitos básicos

Ordenar uma estrutura consiste em rearranjar seus elementos, respeitando uma dada ordem.

- Geralmente, usamos ordem crescente ou ordem decrescente.

Podemos ordenar qualquer coleção de itens, desde que sejam comparáveis uns aos outros.

- **Números** (ordem crescente/decrescente) ou **strings** (ordem alfabética), já comparáveis;
- **Livros** (ordem dada pelo título, autor, ano ou páginas, por exemplo).
  - Necessário implementar a interface `Comparable` e o método `compareTo`.

A eficiência de um algoritmo de ordenação é muito importante, especialmente quando tratamos estruturas com grandes volumes de dados.

Algoritmos de ordenação

Alguns dos muitos algoritmos de ordenação, com suas complexidades assintóticas de tempo.

Algoritmo	**Caso médio**	**Melhor caso**	**Pior caso**
Insertion sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Selection sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Shell sort	$\mathcal{O}(n^{1,5})$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Merge sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Radix sort	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Algumas observações:

- Apesar do método radix sort ter complexidade linear, ele não é aplicável em muitos casos. Logo, o melhor desempenho para o caso geral é $\mathcal{O}(n \log n)$.
- O pior caso do método quick sort é facilmente evitado escolhendo pivôs apropriados. Por ser mais rápido que o merge sort na prática, esse é o algoritmo de ordenação mais usado.

Insertion sort

Ideia geral: seleciona o segundo elemento e o insere na sub-lista à sua esquerda, na posição que mantém a ordenação desejada; repete para cada elemento seguinte.

Insertion sort

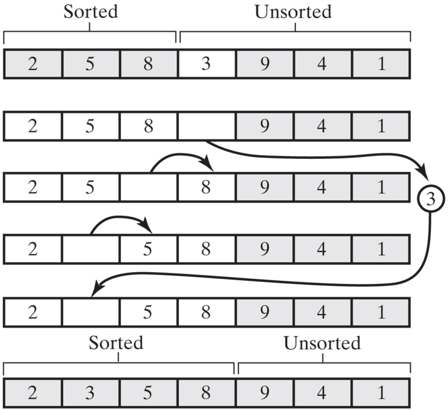
```
Entrada:  vetor A com n elementos.
Saída:    vetor A ordenado [de forma crescente].

PARA i ← 1 até n − 1 FAÇA
  seleciona o elemento A[i]
  insere na sub-lista A[0 ... i] de forma ordenada
RETORNA A
```

Insertion sort

Exemplo de funcionamento

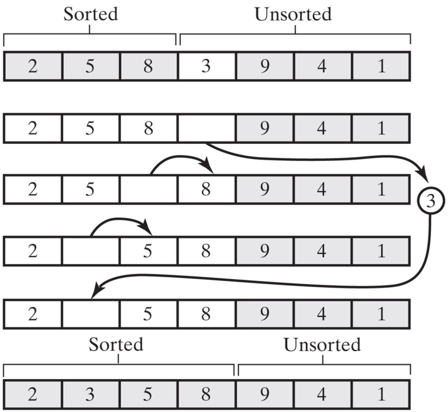
A cada iteração, parte da estrutura já está ordenada, enquanto o restante será ordenado.



Insertion sort

Exemplo de funcionamento

A cada iteração, parte da estrutura já está ordenada, enquanto o restante será ordenado.



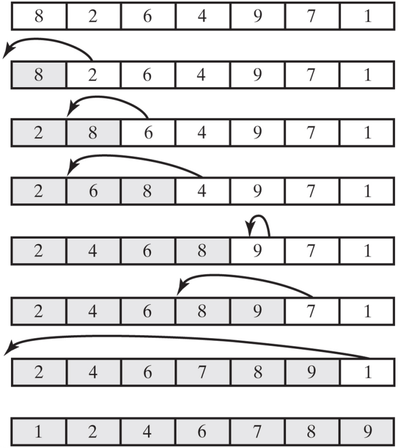
Exemplo: processamento do elemento 3.

1. Os três primeiros elementos estão ordenados.
2. Remove o quarto elemento (3) da sua posição.
3. Desloca o 8, uma vez que 8 > 3.
4. Desloca o 5, uma vez que 5 > 3.
5. Insere na posição vaga, uma vez que 3 > 2.
6. Os quatro primeiros elementos estão ordenados.

Insertion sort

Exemplo de funcionamento

A cada iteração, parte da estrutura já está ordenada, enquanto o restante será ordenado.



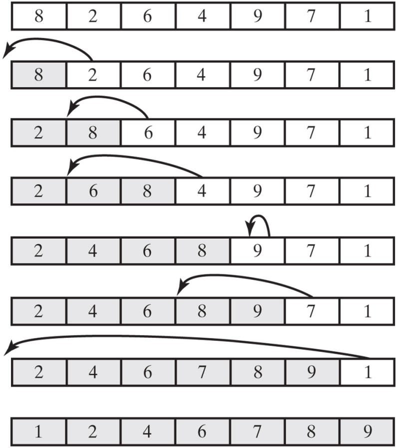
Insertion sort

Exemplo de funcionamento

A cada iteração, parte da estrutura já está ordenada, enquanto o restante será ordenado.

Exemplo: execução completa do algoritmo.

- A cada iteração, o primeiro elemento é removido da sub-lista não ordenada e inserido na posição correta da sub-lista ordenada.
- Iterações:
  1. Insere o elemento 2 na primeira posição;
  2. Insere o elemento 6 na segunda posição;
  3. Insere o elemento 4 na segunda posição;
  4. Insere o elemento 9 na quinta posição;
  5. Insere o elemento 7 na quarta posição;
  6. Insere o elemento 1 na primeira posição.
- Não é usada nenhuma estrutura auxiliar.



Insertion sort

Análise de complexidade

Considerando a ordenação de *arrays*:

- O laço de repetição principal percorre todos os elementos da estrutura, com exceção do primeiro. Logo, esse laço executa n vezes.
- A inserção na sub-lista ordenada exige
  1. a comparação com os elementos dessa sub-lista em busca da posição de inserção;
  2. a inserção na posição correta.
- No **pior caso** (estrutura em ordem decrescente), cada elemento é comparado com todos os anteriores e inserido no início da estrutura, executando $1+2+...+(n-1) = n(n-1)/2$ operações de comparação e *shift* de elementos.
- No **melhor caso** (estrutura já ordenada), o elemento selecionado somente é comparado com o último elemento da sub-lista ordenada e permanece em sua posição.

Insertion sort

Análise de complexidade

Considerando a ordenação de *arrays*:

- O laço de repetição principal percorre todos os elementos da estrutura, com exceção do primeiro. Logo, esse laço executa n vezes.
- A inserção na sub-lista ordenada exige
  1. a comparação com os elementos dessa sub-lista em busca da posição de inserção;
  2. a inserção na posição correta.
- No **pior caso** (estrutura em ordem decrescente), cada elemento é comparado com todos os anteriores e inserido no início da estrutura, executando $1+2+\ldots+(n-1) = n(n-1)/2$ operações de comparação e *shift* de elementos.
- No **melhor caso** (estrutura já ordenada), o elemento selecionado somente é comparado com o último elemento da sub-lista ordenada e permanece em sua posição.

Portanto:

- Pior caso: $\mathcal{O}(n^2)$.
- Melhor caso: $\mathcal{O}(n)$.
- Quanto mais ordenada a estrutura estiver, menor a complexidade do método na prática.

Insertion sort

Implementação

Insertion sort para *arrays*:

```python
def insertion_sort(array):
  n = len(array)

  for i in range(1, n):
    key = array[i]
    j = i - 1
    while j >= 0 and key < array[j]:
      array[j + 1] = array[j]
      j -= 1
    array[j + 1] = key
```

Recursos adicionais

Veja o funcionamento de vários algoritmos de ordenação usando recursos de visualização:

- https://www.cs.usfca.edu/~galles/visualization/Algorithms.html.
- https://visualgo.net/en/sorting.
- https://csvistool.com/?q=sort.

Veja a documentação do Python sobre ordenação de *arrays*:

- https://docs.python.org/pt-br/3.13/howto/sorting.html.

45RPE – Resolução de Problemas com Estruturas de Dados
Prof. Marcelo de Souza