

Algoritmos de Ordenação

O Problema da Ordenação

Bubble Sort

Selection Sort

Insertion Sort

Exercícios

O Problema da Ordenação

O Problema da Ordenação

- Vamos estudar alguns algoritmos para o seguinte problema:

Definição do Problema

Dada uma coleção de elementos, com uma relação de ordem entre eles, ordenar os elementos da coleção de forma crescente.

- Nos nossos exemplos, a coleção de elementos será representada por uma lista de inteiros.
 - Números inteiros possuem uma relação de ordem entre eles.
- Apesar de usarmos números inteiros, os algoritmos que estudaremos servem para ordenar qualquer coleção de elementos que possam ser comparados entre si.

O Problema da Ordenação

- O problema da ordenação é um dos mais básicos em computação.
- Muito provavelmente este é um dos problemas com maior número de aplicações diretas ou indiretas (como parte da solução para um problema maior).
- Exemplos de aplicações diretas:
 - Criação de *rankings*.
 - Definição de preferências em atendimentos por prioridade.
- Exemplos de aplicações indiretas:
 - Otimização de sistemas de busca.
 - Manutenção de estruturas de bancos de dados.

Bubble Sort

Bubble Sort

- A ideia do algoritmo Bubble Sort é a seguinte:
- O algoritmo faz iterações repetindo os seguintes passos:
 - Se $\text{lista}[0] > \text{lista}[1]$, troque $\text{lista}[0]$ com $\text{lista}[1]$.
 - Se $\text{lista}[1] > \text{lista}[2]$, troque $\text{lista}[1]$ com $\text{lista}[2]$.
 - Se $\text{lista}[2] > \text{lista}[3]$, troque $\text{lista}[2]$ com $\text{lista}[3]$.
 - ...
 - Se $\text{lista}[n-2] > \text{lista}[n-1]$, troque $\text{lista}[n-2]$ com $\text{lista}[n-1]$.
- Após uma iteração executando os passos acima, o que podemos garantir?
 - O maior elemento estará na posição correta (a última da lista).

Bubble Sort

- Após a primeira iteração de trocas, o maior elemento estará na posição correta.
- Após a segunda iteração de trocas, o segundo maior elemento estará na posição correta.
- E assim sucessivamente...
- Quantas iterações são necessárias para deixar a lista completamente ordenada?

Bubble Sort

- No exemplo abaixo, os elementos sublinhados estão sendo comparados (e, eventualmente, serão trocados):

[57, 32, 25, 11, 90, 63]

[32, 57, 25, 11, 90, 63]

[32, 25, 57, 11, 90, 63]

[32, 25, 11, 57, 90, 63]

[32, 25, 11, 57, 90, 63]

[32, 25, 11, 57, 63, 90]

- Isto termina a primeira iteração de trocas.
- Como a lista possui 6 elementos, temos que realizar 5 iterações.
- Note que, após a primeira iteração, não precisamos mais avaliar a última posição da lista.

Trocando Elementos em uma Lista

- Podemos trocar os elementos das posições i e j de uma lista da seguinte forma:

```
1 lista = [1, 2, 3, 4, 5]
2 i = 0 # lista[0] = 1
3 j = 2 # lista[2] = 3
4
5 aux = lista[i] lista[i] =
6 lista[j] lista[j] = aux
7
8 print(lista)
9 # [3, 2, 1, 4, 5]
10
```

Trocando Elementos em uma Lista

- Podemos trocar os elementos das posições i e j de uma lista da seguinte forma:

```
1 lista = [1, 2, 3, 4, 5]
2 i = 0 # lista[0] = 1
3 j = 2 # lista[2] = 3
4
5 (lista[i], lista[j]) = (lista[j], lista[i])
6
7
8 print(lista)
9 # [3, 2, 1, 4, 5]
10
```

- O código abaixo realiza as trocas de uma iteração do algoritmo.
- Os pares de elementos das posições 0 e 1, 1 e 2, ..., i-1 e i são comparados e, eventualmente, trocados.
- Assumimos que, das posições i+1 até n-1, a lista já possui os maiores elementos ordenados.

```
1 for j in range(i):  
2     if lista[j] > lista[j + 1]:  
3         ( lista[j], lista[j + 1]) = ( lista[j + 1], lista[j])
```

Bubble Sort

```
1 def bubbleSort(lista):  
2     n = len(lista)  
3     for i in range(n - 1, 0, -1):  
4         for j in range(i):  
5             if lista[j] > lista[j + 1]:  
6                 (lista[j], lista[j + 1]) = (lista[j + 1], lista[j])
```

- Note que as comparações na primeira iteração ocorrem até a última posição da lista.
- Na segunda iteração, elas ocorrem até a penúltima posição.
- E assim sucessivamente...

Selection Sort

Selection Sort

- Dada uma lista contendo n números inteiros, desejamos ordenar essa lista de forma crescente.
- A ideia do algoritmo é a seguinte:
 - Encontre o menor elemento a partir da posição 0. Troque este elemento com o elemento da posição 0.
 - Encontre o menor elemento a partir da posição 1. Troque este elemento com o elemento da posição 1.
 - Encontre o menor elemento a partir da posição 2. Troque este elemento com o elemento da posição 2.
 - E assim sucessivamente...

Selection Sort

- No exemplo abaixo, os elementos sublinhados representam os elementos que serão trocados na iteração do Selection Sort:

Iteração 0: [57, 32, 25, 11, 90, 63]

Iteração 1: [11, 32, 25, 57, 90, 63]

Iteração 2: [11, 25, 32, 57, 90, 63]

Iteração 3: [11, 25, 32, 57, 90, 63]

Iteração 4: [11, 25, 32, 57, 90, 63]

Iteração 5: [11, 25, 32, 57, 63, 90]

Selection Sort

- Podemos criar uma função que retorna o índice do menor elemento de uma lista (formada por números inteiros) a partir de uma posição inicial dada:

```
1 def indiceMenor(lista, inicio): minimo =  
2     inicio  
3     n = len(lista)  
4     for j in range(inicio + 1, n):  
5         if lista[minimo] > lista[j]: minimo = j  
6     return minimo  
7
```

Selection Sort

- Dada a função anterior, que encontra o índice do menor elemento de uma lista a partir de uma dada posição, como implementar o algoritmo de ordenação?
 - Encontre o menor elemento a partir da posição 0 e troque-o com o elemento da posição 0.
 - Encontre o menor elemento a partir da posição 1 e troque-o com o elemento da posição 1.
 - Encontre o menor elemento a partir da posição 2 e troque-o com o elemento da posição 2.
 - E assim sucessivamente...

- Usando a função auxiliar `indiceMenor`, podemos implementar o Selection Sort da seguinte forma:

```
1 def selectionSort(lista):  
2     n = len(lista)  
3     for i in range(n):  
4         minimo = indiceMenor(lista, i)  
5         (lista[i], lista[minimo]) = (lista[minimo], lista[i])
```

Selection Sort

- Usando a função auxiliar `indiceMenor`, podemos implementar o Selection Sort da seguinte forma:

```
1 def selectionSort(lista):  
2     n = len(lista)  
3     for i in range(n - 1):  
4         minimo = indiceMenor(lista, i)  
5         (lista[i], lista[minimo]) = (lista[minimo], lista[i])
```

- É possível diminuir o número de trocas no melhor caso?
- Vale a pena testar se `lista[i] != lista[minimo]` antes de realizar a troca?

Insertion Sort

Insertion Sort

- A ideia do algoritmo Insertion Sort é a seguinte:
 - A cada iteração i , os elementos das posições 0 até $i-1$ da lista estão ordenados.
 - Então, precisamos inserir o elemento da posição i , entre as posições 0 e i , de forma a deixar a lista ordenada até a posição i .
 - Na iteração seguinte, consideramos que a lista está ordenada até a posição i e repetimos o processo até que a lista esteja completamente ordenada.

Insertion Sort

- No exemplo abaixo, o elemento sublinhado representa o elemento que será inserido na i -ésima iteração do Insertion Sort:

[57, 25, 32, 11, 90, 63]: lista ordenada entre as posições 0 e 0.

[25, 57, 32, 11, 90, 63]: lista ordenada entre as posições 0 e 1.

[25, 32, 57, 11, 90, 63]: lista ordenada entre as posições 0 e 2.

[11, 25, 32, 57, 90, 63]: lista ordenada entre as posições 0 e 3.

[11, 25, 32, 57, 90, 63]: lista ordenada entre as posições 0 e 4.

[11, 25, 32, 57, 63, 90]: lista ordenada entre as posições 0 e 5.

Insertion Sort

- Podemos criar uma função que, dados uma lista e um índice i , insere o elemento de índice i entre os elementos das posições 0 e $i-1$ (pré-ordenados), de forma que todos os elementos entre as posições 0 e i fiquem ordenados:

```
1 def insertion(lista, i):  
2     aux = lista[i]  
3     j = i - 1  
4     while (j >= 0) and (lista[j] > aux):  
5         lista[j + 1] = lista[j]  
6     j = j - 1  
7     lista[j + 1] = aux
```

Insertion Sort

- Exemplo de execução da função insertion:
 - Configuração inicial:
[11, 31, 54, 58, 66, 12, 47], $i = 5$, $aux = 12$
 - Iterações:
[11, 31, 54, 58, 66, 12, 47], $j = 4$
[11, 31, 54, 58, 66, 66, 47], $j = 3$
[11, 31, 54, 58, 58, 66, 47], $j = 2$
[11, 31, 54, 54, 58, 66, 47], $j = 1$
[11, 31, 31, 54, 58, 66, 47], $j = 0$
 - Neste ponto, temos que $lista[j] < aux$, logo, o loop while é encerrado e a atribuição $lista[j + 1] = aux$ é executada: [11, 12, 31, 54, 58, 66, 47]

Insertion Sort

- Em Python, podemos implementar a função insertion de forma ainda mais simples, inserindo o elemento na posição desejada com um único comando.

```
1 def insertion(lista, i):  
2     j = i - 1  
3     while (j >= 0) and (lista[j] > lista[i]): j = j - 1  
4     lista[j + 1:i + 1] = [lista[i]] + lista[j + 1:i]  
5
```

Insertion Sort

- Usando a função auxiliar insertion, podemos implementar o Insertion Sort da seguinte forma:

```
1 def insertionSort(lista): n = len(  
2     lista)  
3     for i in range(1, n):  
4         insertion(lista, i)
```

Resumo

- Não existe um algoritmo de ordenação que seja o melhor em todas as possíveis situações.
- Para escolher o algoritmo mais adequado para uma dada situação, precisamos verificar as características específicas dos elementos que devem ser ordenados.
- Por exemplo:
 - Se os elementos a serem ordenados forem grandes, por exemplo, registros acadêmicos de alunos, o Selection Sort pode ser uma boa escolha, já que ele efetuará, no pior caso, muito menos trocas que o Insertion Sort ou o Bubble Sort.
 - Se os elementos a serem ordenados estiverem quase ordenados (situação relativamente comum), o Insertion Sort realizará muito menos operações (comparações e trocas) do que o Selection Sort ou o Bubble Sort.
- Teste de tempo de execução dos algoritmos de ordenação:
 - <https://repl.it/@sandrooliveira/testetempo>

Exercícios

Exercícios

1. Altere o Bubble Sort para que o algoritmo pare assim que for possível perceber que a lista está ordenada. Qual o custo deste novo algoritmo em termos do número de comparações entre elementos da lista (tanto no melhor, quanto no pior caso)?
2. Escreva uma função k -ésimoque, dada uma lista de tamanho n e um inteiro k (tal que $1 \leq k \leq n$), determine o k -ésimomenor elemento da lista. Analise o custo da sua função em termos do número de comparações realizadas entre elementos da lista.