Estruturas de Dados I

Algoritmos de Ordenação

Prof. Dr. Lidio Mauro Lima de Campos
limadecampos@gmail.com

Universidade Federal do Pará – UFPA
ICEN
FACOMP

Agenda

- **Introdução (Sorting).**
- **Algoritmos Iterativos de Ordenação:**
  - **Ordenação por Trocas - Bolha (*Bubble Sort).***
  - **Ordenação por Inserção (*Insertion Sort).***
    - **Método da Inserção Direta.**
    - **Métodos de Incrementos Decrescentes - ShellSort.**
  - **Ordenação por Seleção (*Selection Sort).***
    - **Método da Seleção Direta.**
  - **Complexidade.**

Introdução

- **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.
- A coleção de elementos poderá ser 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.

Introdução

- 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.

Ordenação de Vetores

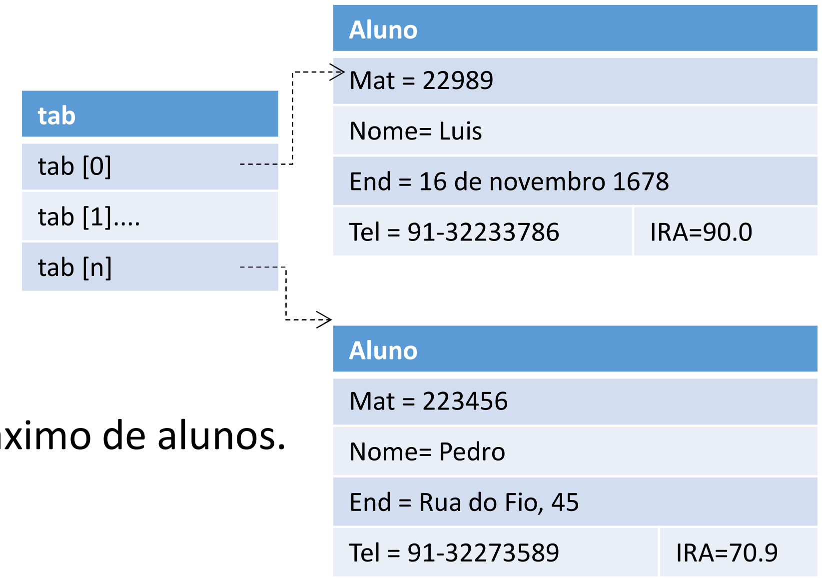
- **Entrada:** vetor com os elementos a serem ordenados.
- **Saída:** o mesmo vetor com os seus elementos na ordem especificada.
- **Ordenação:** pode ser aplicada a qualquer dado com ordem bem definida
  - Vetor com dados simples (*int, float, etc).*
  - Vetor de structs, objetos

Ordenação de Objetos

- **Ordenação:** pode ser aplicada a qualquer dado com ordem bem definida
- **Vetor com dados complexos** (*Structs, Objetos)*
  - Chave da ordenação escolhida entre os Campos.
  - Em geral, ineficiente pois toda a estrutura deve ser trocada.

- **Exemplo: Cadastro de Alunos.**
- Para estruturar esses dados, pode-se definir um tipo que representa os dados de um aluno.

```
struct aluno
{
  int mat; char nome[81]; char end [121]; char tel[21];
 float ira;
};
typedef struct aluno Aluno;
```



- Vamos montar a tabela de alunos usando um vetor com um número máximo de alunos.

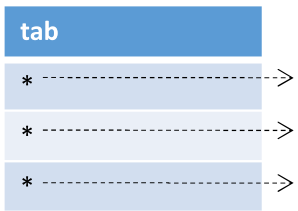
```
#define MAX 100
Aluno tab[MAX];
```

Ordenação de Objetos

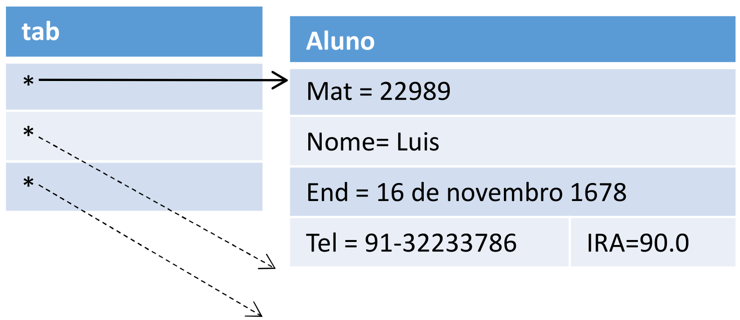
- **Ordenação:** pode ser aplicada a qualquer dado com ordem bem definida
- **Vetor de ponteiros para dados complexos**
  - Troca da Ordem dos Elementos = Troca de ponteiros.

- **Exemplo: Cadastro de Alunos.**
- Podemos trabalhar com um vetor de ponteiros que é mais eficiente.

```
#define MAX 100
Aluno* tab[MAX];
```



tab[i]=(Aluno*)malloc(sizeof(Aluno))



Ordenação Bolha (Bubble Sort)

- A idéia geral da ordenação bolha é colocar os elementos "**maiores**" nos seus lugares corretos.
- O algoritmo faz iterações repetindo os seguintes passos:
  - Se `lista[0] > lista[1]`, troque `lista[0]` com `lista[1]`.
  - Se `lista[1] > lista[2]`, troque `lista[1]` com `lista[2]`.
  - Se `lista[2] > lista[3]`, troque `lista[2]` com `lista[3]`.
  - ...
  - Se `lista[n-2] > lista[n-1]`, troque `lista[n-2]` com `lista[n-1]`.
- ***Quando dois elementos estão fora de ordem***, ***troque-os de posição*** até que o **i-ésimo elemento de maior valor do vetor seja levado para as posições finais do vetor**.

	i=3				i=2				i=1			
j=0	4	2	5	1	2	4	1	5	2	1	4	5
j=1	2	4	5	1	2	4	1	5	1	2	4	5
j=2	2	4	5	1	2	1	4	5				
	2	4	1	5								

i=controle do número de iterações, j=controle do número de comparações

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

[$\underline{57}$, $\underline{32}$, 25, 11, 90, 63]
[32, $\underline{57}$, $\underline{25}$, 11, 90, 63]
[32, 25, $\underline{57}$, $\underline{11}$, 90, 63]
[32, 25, 11, $\underline{57}$, $\underline{90}$, 63]
[32, 25, 11, 57, $\underline{90}$, $\underline{63}$]
[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.

Ordenação Bolha (Bubble Sort)

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

#i=n-1,....,1 incremento -1   #j=0,....,i-1 incremento 1

	i=3				i=2				i=1			
j=0	4	2	5	1	2	4	1	5	2	1	4	5
j=1	2	4	5	1	2	4	1	5	1	2	4	5
j=2	2	4	5	1	2	1	4	5				
	2	4	1	5								

https://www.youtube.com/watch?v=nmhjrl-aW5o#action=share

Ordenação Bolha (Bubble Sort)

- 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...

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

Ordenação Bolha (Bubble Sort) – Análise de Complexidade

- A complexidade de um algoritmo pode ser medida pelo esforço computacional.
- O esforço computacional pode ser determinado pelo **número de comparações entre elementos da lista**.

i=n-1,i>0,i-- , logo i=1 até n-1

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

j=0,1,...,i-1

- Número máximo de comparações entre elementos da lista:

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

$NT = LS - LI + 1 \quad \sum_{i=1}^{6} x_i \Rightarrow NT = 6 - 1 + 1 = 6$

$$\sum_{i=1}^{n} k = k + k + \ldots + k = nk$$

Seja $n \geq 1$ um inteiro positivo qualquer. Vale que

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Ordenação Bolha (Bubble Sort) – Análise de Complexidade

- A complexidade de um algoritmo pode ser medida pelo esforço computacional.
- O esforço computacional pode ser determinado pelo **número de comparações entre elementos da lista**.

Passada	Comparações
1	(n-1)
2	(n-2)
3	(n-3)
....	.....
n-2	2
n-1	1

- O tempo total gasto pelo algoritmo é proporcional a:

$Sn=(n-1)+(n-2)+.......+2+1=n*(n-1)/2=O(n^2)$.

$Sn=(a1+an)*nt/2$

$a1=(n-1)$, $an=1$, $nt=(n-1)$

Ordenação Bolha (Bubble Sort) – Análise de Complexidade

- A complexidade de um algoritmo pode ser medida pelo esforço computacional.
- O esforço computacional pode ser determinado pelo **número de trocas entre elementos da lista**.

i=n-1,i>0,i-- , logo i=1 até n-1

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

j=0,1,...,i-1

- Número máximo de trocas entre elementos da lista:

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

Ordenação Bolha (Bubble Sort) – Análise de Complexidade

- A complexidade de um algoritmo pode ser medida pelo esforço computacional.
- O esforço computacional pode ser determinado pelo **número de comparações entre elementos da lista**.

i=n-1,i>0,i-- , logo i=1 até n-1

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

j=0,1,...,i-1

- Número mínimo de comparações entre elementos da lista:

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

Ordenação Bolha (Bubble Sort) – Análise de Complexidade

- A complexidade de um algoritmo pode ser medida pelo esforço computacional.
- O esforço computacional pode ser determinado pelo **número de trocas entre elementos da lista**.

i=n-1,i>0,i-- , logo i=1 até n-1

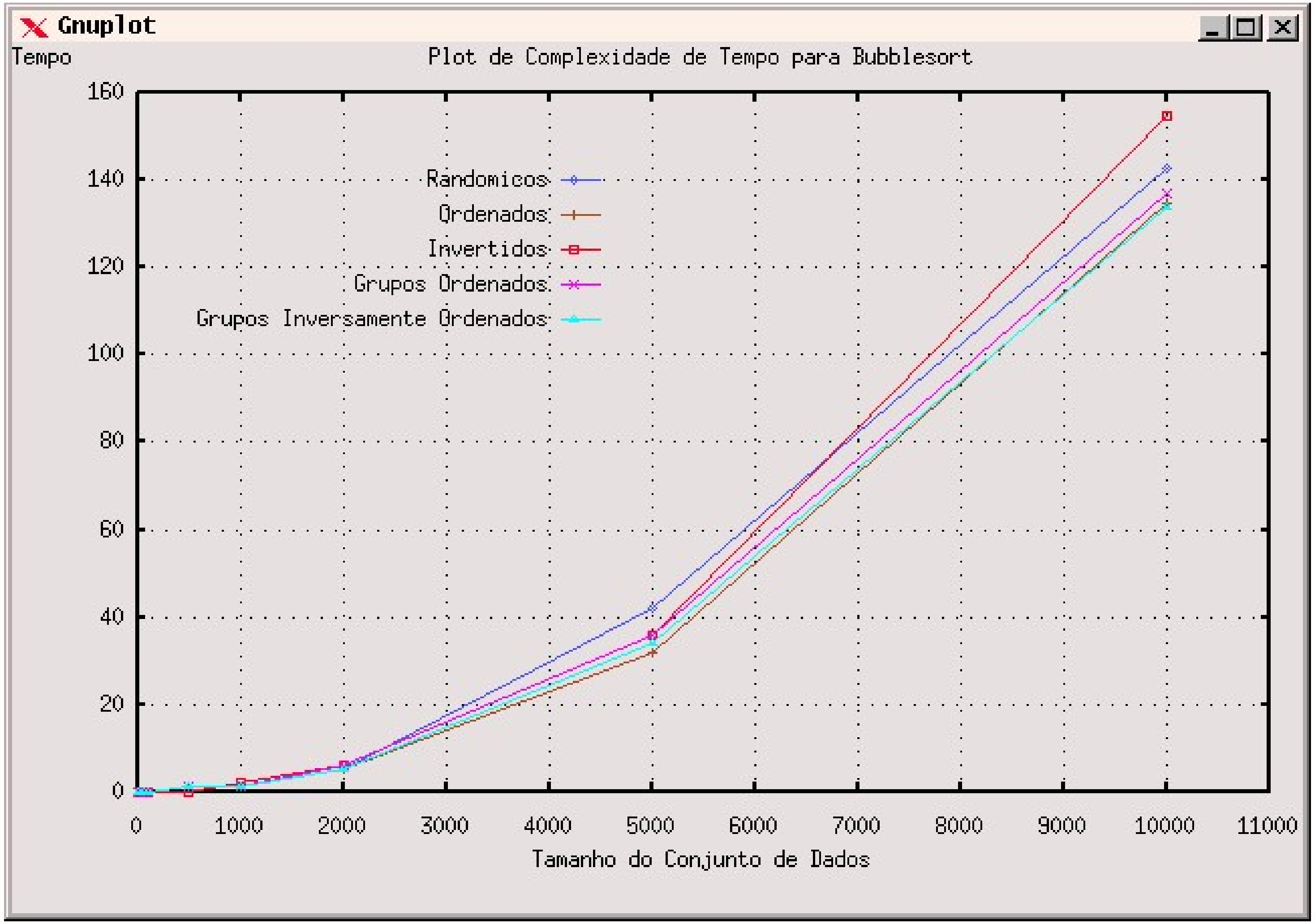
```
def bubbleSort(lista):
  n = len(lista)
  for i in range(n - 1, 0, -1):
    for j in range(i):
      if lista[j] > lista[j + 1]:
        (lista[j], lista[j + 1]) = (lista[j + 1], lista[j])
```

j=0,1,...,i-1

- Número mínimo de trocas entre elementos da lista:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 0 = 0$$

Ordenação Bolha (Bubble Sort)- Análise de Complexidade



Ordenação por Inserção (*Insertion Sort).*

- A ideia do algoritmo **Insertion Sort** é a seguinte:
- **A cada iteração i**, **os elementos das posições 0 até i-1 (SO)** da lista estão ordenados (lista ordenada entre as posições 0 e 0 para i=1, lista ordenada entre as posições 0 e 1 para i=2).
- Então, **precisamos inserir o elemento da posição i (primeiro elemento do SNO)**, **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**.

Ordenação por Inserção (*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. i=1
[25, 57, 32, 11, 90, 63]: lista ordenada entre as posições 0 e 1. i=2
[25, 32, 57, 11, 90, 63]: lista ordenada entre as posições 0 e 2. i=3
[11, 25, 32, 57, 90, 63]: lista ordenada entre as posições 0 e 3. i=4
[11, 25, 32, 57, 90, 63]: lista ordenada entre as posições 0 e 4. i=5
[11, 25, 32, 57, 63, 90]: lista ordenada entre as posições 0 e 5. i=6

Ordenação por Inserção (*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:

```
def insertion(lista, i):
  aux = lista[i] #aux = recebe elemento do SNO
  j = i - 1
  while (j >= 0) and (lista[j] > aux):
    lista[j + 1] = lista[j] #troca primeiro elemento do SNO, com o
                            elemento j do SO.
    j = j - 1
  lista[j + 1] = aux  #posição j+1 do segmento O recebe primeiro elemento do SNO

def insertionSort(lista):
  n = len(lista)
  for i in range(1, n):  #elemento atual do SNO
    insertion(lista, i)


arr = [18, 15, 7]
insertionSort(arr)

for i in range(len(arr)):
        print ("%d" %arr[i])
#lidio
```

Ordenação por Inserção (*Insertion Sort)*

```python
def insertion(lista, i):
 aux = lista[i]
 j = i - 1
 while (j >= 0) and (lista[j] > aux):
   lista[j + 1] = lista[j]
   j = j - 1
 lista[j + 1] = aux

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


arr = [18, 15, 7]
insertionSort(arr)

for i in range(len(arr)):
        print ("%d" %arr[i])
#lidio
```

lista i=1

18	**15**	**7**

aux=lista[1]=15

j=0

while(j>=0 (V) && lista[0]=18>15 (V)

lista[1]=lista[0]=18

j=0-1=-1

18	**18**	**7**

while(j>=0 (F) &&....

lista[0]=15

15	**18**	**7**

Ordenação por Inserção (*Insertion Sort)*

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

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


arr = [18, 15, 7]
insertionSort(arr)

for i in range(len(arr)):
        print ("%d" %arr[i])
#lidio
```

Lista i=2

15	**18**	**7**

aux=lista[2]=7

j=1

while(j>=0 (V) && lista[1]=18>7 (V)

lista[2]=lista[1] =18

j=1-1=0

15	**18**	**18**

while(j>=0 (V) && lista[0]=15>7 (V)

lista[1]=lista[0] =15

j=0-1=-1

15	**15**	**18**

while(j>=0 (F) && lista[0]=15>7 (V)

lista[0]=aux=7

7	**15**	**18**

Ordenação por Inserção (*Insertion Sort).*

Exemplo:12, 11, 13, 5, 6

Vamos fazer um loop para **i = 1** (segundo elemento do vetor) a 5 (tamanho do vetor)

i = 1. Como 11 é menor que aux**=12**, mova 12 e insira 11 antes de 12

11, 12, 13, 5, 6

i = 2. aux=13 permanecerá na sua posição, pois todos os elementos em A [0..i-1] são menores que 13

11, 12, 13, 5, 6

i = 3. aux=5 se moverá para o começo e todos os outros elementos de 11 a 13 se moverão uma posição à frente de sua posição atual.

5, 11, 12, 13, 6

i = 4. aux=6 se moverá para a posição após 5, e os elementos de 11 a 13 se moverão uma posição à frente de sua posição atual.

5, 6, 11, 12, 13

Ordenação por Inserção (*Insertion Sort) – Análise de Complexidade*

```
def insertion(lista, i):
 aux = lista[i]
 j = i - 1
 while (j >= 0) and (lista[j] > aux):  #j=0...i-1
   lista[j + 1] = lista[j]
   j = j - 1
 lista[j + 1] = aux

def insertionSort(lista):
 n = len(lista)
 for i in range(1, n): #i=1..n-1
   insertion(lista, i)


arr = [18, 15, 7]
insertionSort(arr)

for i in range(len(arr)):
        print ("%d" %arr[i])
#lidio
```

PIOR CASO: Vetor Desordenado

- Número máximo de comparações entre elementos da lista:

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

MELHOR CASO: Vetor de Entrada fornecido já se encontra ordenado.

- Número mínimo de comparações entre elementos da lista:

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

#para cada i (lista[j]>aux) é realizado um única vez . E lista[j+1]=aux é apenas uma permutação.

$NT = LS - LI + 1 \quad \sum_{i=1}^{6} x_i \Rightarrow NT = 6 - 1 + 1 = 6$

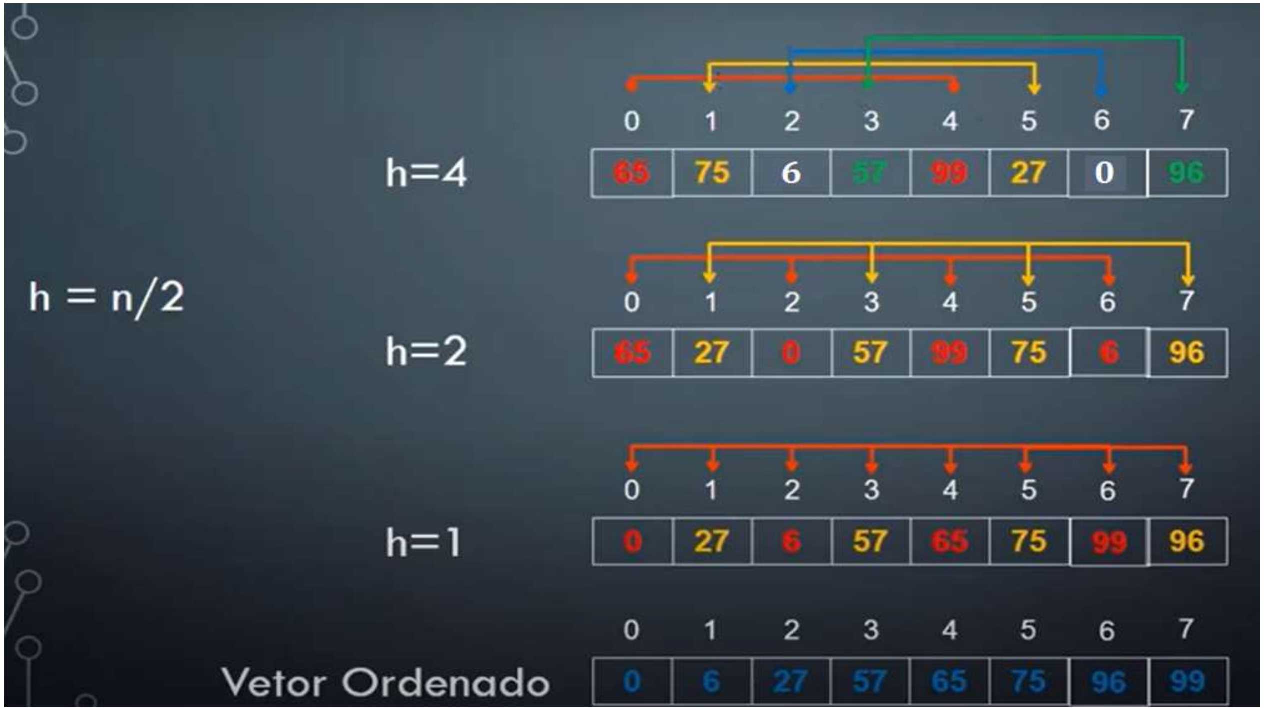
Método dos Incrementos Decrescente - ShellSort

- Proposto por Donald Shell em 1959.
- É uma extensão do InsertionSort.
- Problemas com o algoritmo de ordenação por inserção: Troca itens adjacentes para determinar o ponto de inserção.
- São efetuadas n − 1 comparações e movimentações quando o menor item está na posição mais à direita do vetor.
- Ineficiente para n grande

- https://www.youtube.com/watch?v=qzXAVXddcPU

Método dos Incrementos Decrescente - ShellSort

- Os itens separados de h posições são rearranjados, gerando sequências ordenadas.
- É dito que cada sequência está h-ordenada.
- Depois, o valor de h é reduzido progressivamente, até atingir o valor 1, que resultará no vetor completamente ordenado.



- A medida que h decresce, o vetor vai ficando cada vez mais próximo da ordenação.
- Com h = 1, o algoritmo se comporta exatamente igual ao InsertionSort.

Método dos Incrementos Decrescente - ShellSort

h = 4

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

h = 2

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

h = 1 (inserção)

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

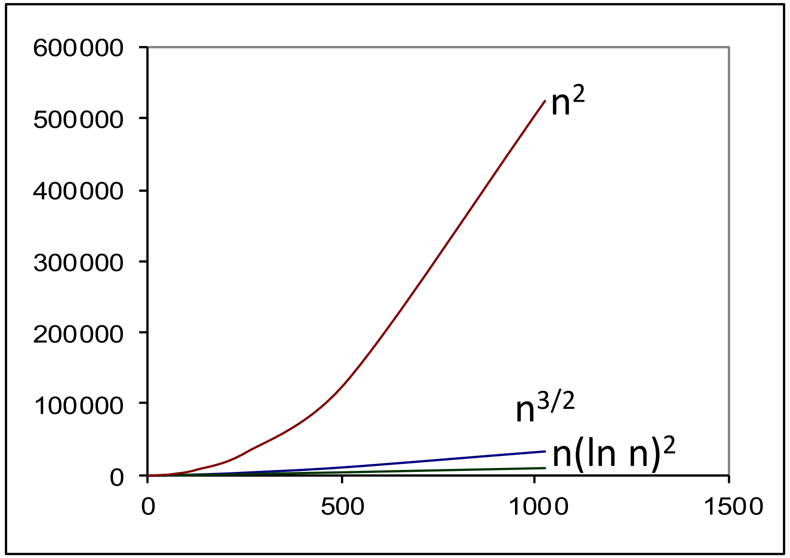
Método dos Incrementos Decrescente – ShellSort – Análise de Complexidade

- **Complexidade**
- A razão pela qual o método é eficiente ainda não é conhecida, porque ninguém foi capaz de analisar o algoritmo.
  - Segundo [KNU73] e [WIR89] a análise de desempenho do método é muito complexa, pois identifica alguns problemas matemáticos bastante difíceis, alguns deles ainda não resolvidos.
  - Um dos problemas é determinar o efeito que a ordenação dos segmentos em um passo produz nos passos subsequentes.
  - Também não se conhece a sequência de incrementos que produz o melhor resultado.

Conjetura 1	Conjetura 2
$O(n)=O(n^{1,25})$ [KNU73] e [WIR89]	$O(n)=O(n (\ln n)^2)$ [ZIV99], (pág,77)

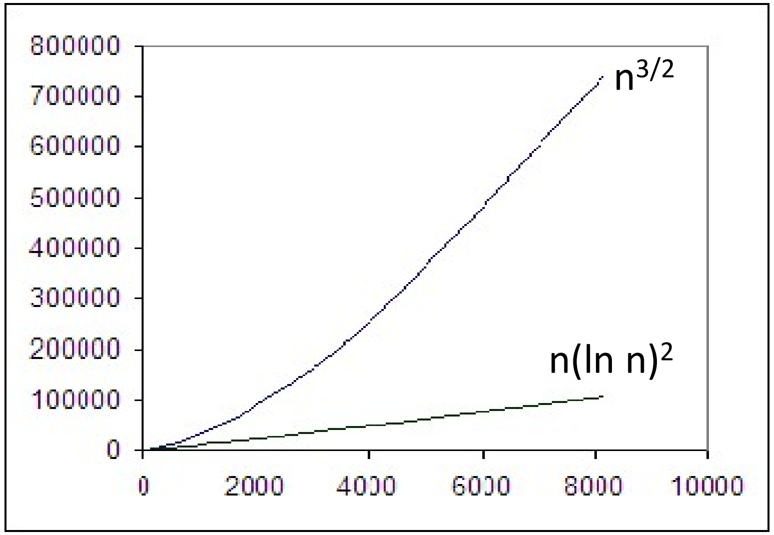
Método dos Incrementos Decrescente – ShellSort – Análise de Complexidade

- **Complexidade**
- Em termos de tempo de execução, o **Shell Sort** apresenta desempenho bem superior àqueles que têm esse tempo regido por funções quadráticas n^2 : seu tempo, para N elementos, é proporcional a $n^{3/2}$.
- Veja, no gráfico abaixo (a curva intermediária), O gráfico da função indicativa de desempenho do ShellSort ($n^{3/2}$).



Método dos Incrementos Decrescente - ShellSort

- **Complexidade**
- É importante notar que embora as duas curvas mais baixas (relativa a $n^{3/2}$ e a $n(\ln n)^2$ aparentem ter, no gráfico a seguir, um comportamento muito próximo, isto não será verdade para valores maiores de n, como mostra o segundo gráfico a seguir, que compara o comportamento apenas dessas duas curvas, para valores de n até 10 vezes maiores:



Ordenação por Seleção (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...

Ordenação por Seleção (Selection *Sort).*

- No exemplo abaixo, os elementos sublinhados representam os elementos que serão trocados na iteração i 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]

Ordenação por Seleção (Selection *Sort).*

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

```python
def indiceMenor(lista, inicio):
  minimo = inicio
  n = len(lista)
  for j in range(inicio + 1, n):
    if lista[minimo] > lista[j]:
      minimo = j
  return minimo
```

Ordenação por Seleção (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?
- Usando a função auxiliar indiceMenor podemos implementar o Selection Sort da seguinte forma:

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

Ordenação por Seleção (Selection *Sort) – Análise de Complexidade.*

```
def indiceMenor(lista, inicio):
  minimo = inicio
  n = len(lista)
  for j in range(inicio + 1, n): #j=i+1...n-1
    if lista[minimo] > lista[j]:
      minimo = j
  return minimo
```

```
def selectionSort(lista):
  n = len(lista)
  for i in range(n): #i=0...n-1
    minimo = indiceMenor(lista, i)
    (lista[i], lista[minimo]) = (lista[minimo], lista[i])
```

Número máximo e mínimo de Comparações entre elementos da lista

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n - i - 1 = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

n-1 n-2 n-3 n-(n-1)-1

PA: n-1 n-2 n-3 0

a1=n-1

an=0

n termos

$NT = LS - LI + 1$ $\quad \sum_{i=1}^{6} x_i \Rightarrow NT = 6 - 1 + 1 = 6$ $\quad \sum_{i=1}^{n} k = k + k + ... + k = nk$

S=(a1+an)n/2

Ordenação por Seleção (Selection *Sort) – Análise de Complexidade.*

```
def selectionSort(lista):
  n = len(lista)
  for i in range(n - 1): #0..n-1
    minimo = indiceMenor(lista, i)
    (lista[i], lista[minimo]) = (lista[minimo], lista[i])
```

mínimo

- Número máximo de trocas entre elementos da lista:

$$f(n) = \sum_{i=0}^{n-1} 1 = n - 1$$

Teste de Tempo dos Algoritmos

- SIMULADOR
- https://replit.com/@sandrooliveira/testetempo
- https://math.hws.edu/eck/js/sorting/xSortLab.html

Algoritmo	Comparações			Movimentações			Espaço
	Melhor	Médio	Pior	Melhor	Médio	Pior	
Bubble	$O(n^2)$			$O(n^2)$			$O(1)$
Selection	$O(n^2)$			$O(n)$			$O(1)$
Insertion	$O(n)$	$O(n^2)$		$O(n)$	$O(n^2)$		$O(1)$
Merge	$O(n \log n)$			–			$O(n)$
Quick	$O(n \log n)$		$O(n^2)$	–			$O(n)$
Shell	$O(n^{1.25})$ ou $O(n (\ln n)^2)$			–			$O(1)$

Referências

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Algoritmos – Teoria e Prática, Tradução da Segunda Edição. Campus, 2016.
- Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Pioneira Thomson Learning, 4ed. 2009.