PANC: Projeto e Análise de Algoritmos

Aula 10: Algoritmos de Ordenação I - Inserção:

Ordenação Direta, Binária e Shell Sort

Breno Lisi Romano

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Sumário

- Introdução
- Algoritmos de Ordenação por Inserção:
  - Ordenação Direta (*Insertion Sort*)
  - Ordenação Binária
  - Shell Sort
- Exemplos Práticos

Introdução (1)

- Em diversas aplicações, os dados devem ser armazenados obedecendo uma determinada ordem
- Alguns algoritmos podem explorar a ordenação dos dados para operar de maneira mais eficiente, do ponto de vista de desempenho computacional
- Para obtermos os dados ordenados, temos basicamente duas alternativas:
  - Inserimos os elementos na estrutura de dados respeitando a ordenação (dizemos que a ordenação é garantida por construção)
  - A partir de um conjunto de dados já criado, aplicamos um algoritmo para ordenar seus elementos

Introdução (2)

- Devido ao seu uso muito frequente, é importante ter à disposição algoritmos de ordenação (*sorting*) eficientes tanto em termos de tempo (devem ser rápidos) como em termos de espaço (devem ocupar pouca memória durante a execução) → Complexidade

- Vamos descrever os algoritmos de ordenação considerando o seguinte cenário:
  - a entrada é um array cujos elementos precisam ser ordenados
  - a saída é o mesmo array com seus elementos na ordem especificada
  - o espaço que pode ser utilizado é apenas o espaço do próprio array

- Portanto, vamos discutir ordenação de arrays, mas a lógica é aplicada a qualquer outra estrutura de dados

Introdução (3)

- Existem três grupos de algoritmos diferentes de ordenação e outros algoritmos que não pertencem a nenhum destes grupos:
  - Métodos de Ordenação por Inserção
    - Direta
    - Binária
    - Shell Sort
  - Métodos de Ordenação por Troca
    - Bubble Sort
    - Shake Sort
    - Comb Sort
    - Quick Sort
  - Métodos de Ordenação por Seleção
    - Select Sort
    - Heap Sort
  - Outros Métodos de Ordenação
    - Merge Sort
    - Radix Sort

Introdução (4)

- Antes de aprendermos sobre os algoritmos de ordenação, vamos pensar como ficaria o algoritmo para ordenar 3 variáveis: x, y e z

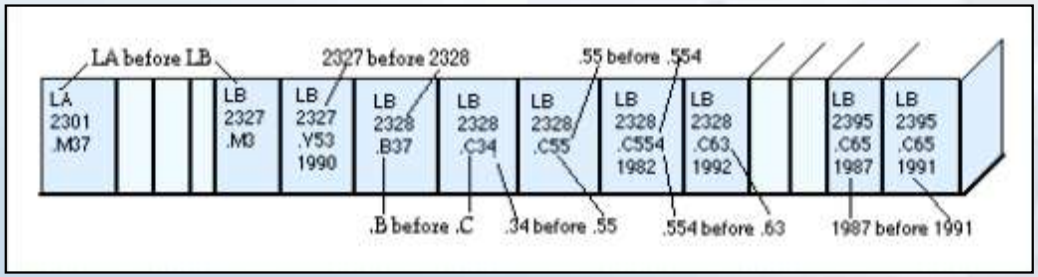
- Por exemplo:
  - Desordenado (x=10, y=30, z=20)
  - Ordenado (Imprime 10, 20, 30)
  - Desordenado (x=30, y=20, z=10)
  - Ordenado (Imprime 10, 20, 30)
  - Desordenado (x=20, y=10, z=30)
  - Ordenado (Imprime 10, 20, 30)
  - Desordenado (x=10, y=20, z=30)
  - Ordenado (Imprime 10, 20, 30)

```
void OrdenaTresVariaveis(int x, int y, int z)
{
    //x eh o menor
    if((x <= y)&&(x <= z)) {
        if(y <= z) printf("\n Ord = %d %d %d\n", x, y, z);
        else       printf("\n Ord = %d %d %d\n", x, z, y);
    }

    //y eh o menor
    if((y <= x)&&(y <= z)) {
        if(x <= z) printf("\n Ord = %d %d %d\n", y, x, z);
        else       printf("\n Ord = %d %d %d\n", y, z, x);
    }

    //z eh o menor
    if((z <= x)&&(z <= y)) {
        if(x <= y) printf("\n Ord = %d %d %d\n", z, x, y);
        else       printf("\n Ord = %d %d %d\n", z, y, x);
    }
}
```

Problema: Ordenação

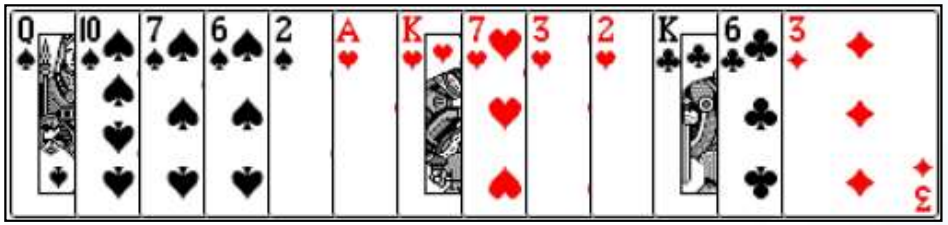
- **Problema de Ordenação:**
  - Ordenar uma sequência de números de maneira não decrescente
- **Entrada:**
  - Uma sequência de n números $<a_1, a_2, a_3, \ldots, a_n>$
- **Saída:**
  - Uma permutação $<a'_1, a'_2, a'_3, \ldots, a'_n>$ da sequência de entrada, tal que $a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_n$

Problema: Ordenação - Exemplos



Números de Registros em uma Biblioteca

Pacotes do FedEx



Jogo de Cartas

Contatos do Celular

Casas de Hogwarts – Chapéu Seletor

Detalhando...

ALGORITMOS DE ORDENAÇÃO POR INSERÇÃO

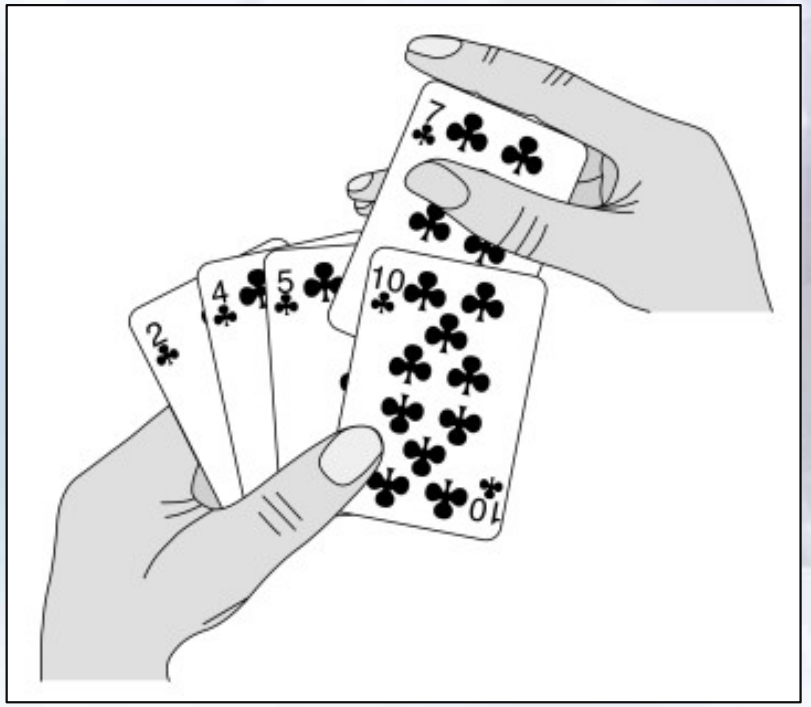
Ordenação Direta, Binária e Shell Sort

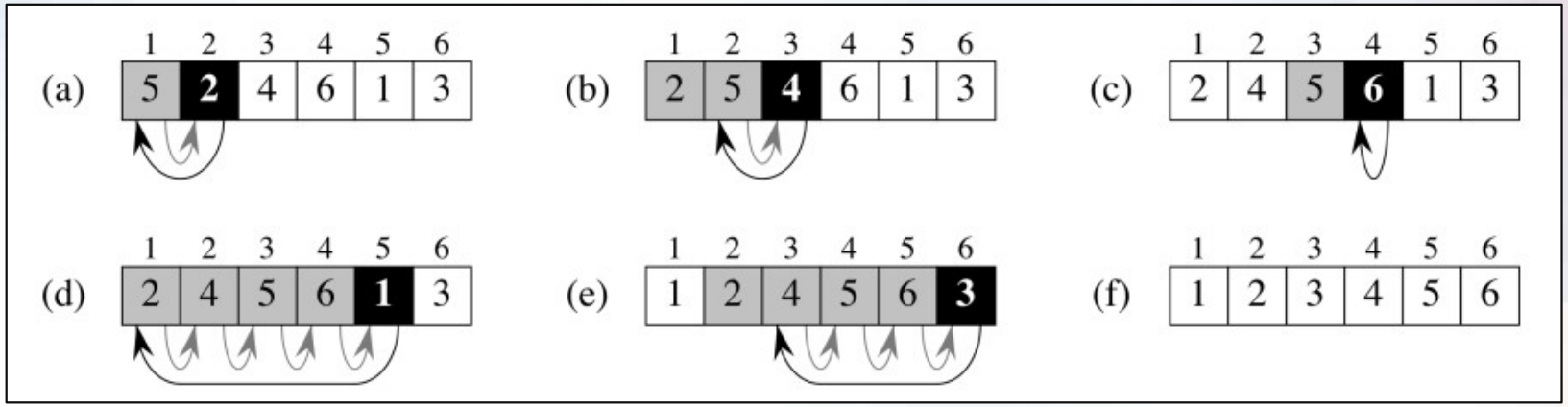
Introdução: Ordenação por Inserção

- Métodos caracterizados pelo princípio no qual se divide o array em dois segmentos, sendo um já ordenado e o outro a ser ordenado
  - Primeiro segmento formado por apenas um elemento → considerado ordenado
  - Segundo segmento contém n-1 elementos restantes
- A partir disto, iterações são desenvolvidas sendo que, em cada uma delas, um elemento do segmento não ordenado é transferido para o segmento ordenado, na sua posição correta
- Métodos de Ordenação por Inserção
  - Direta (*Insertion Sort*)
  - Binária
  - Shell Sort

Ordenação por Inserção: Direta

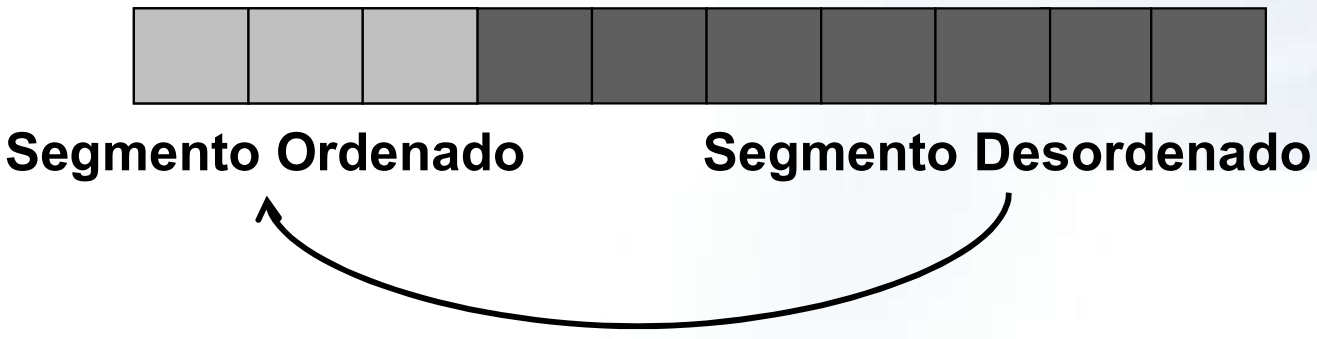
Ordenação por Inserção: Ilustração (1)





Ordenação por Inserção: Lógica (2)

- A **ordenação por inserção** é caracterizada pelo **princípio** no qual se **divide** o **array** em **dois segmentos**: um já **ordenado** e o outro **não ordenado**
- Inicialmente, o **primeiro segmento** é formado **apenas** por **um elemento** (já considerado ordenado)
- O **segundo segmento** contém **n-1 elementos** restantes **não ordenados**
- O **progresso** se desenvolve em **n-1 interações** sendo que, em cada uma delas, um **elemento do segmento não ordenado** é **transferido** para o **primeiro segmento**, e **inserido** na **posição correta** em relação aos demais elementos já existentes



Ordenação por Inserção: Lógica (3)

- Veja os passos utilizados para se ordenar valores pelo método da inserção direta:
  1. Considere o primeiro elemento como pertencente ao segmento ordenado S1
  2. Considere os demais elementos como pertencentes ao segmento desordenado S2
  3. Toma-se um dos elementos não ordenados do segmento S2, a partir do primeiro, e localiza-se a sua posição relativa correta em S1
  4. A cada comparação realizada entre o elemento do segmento S2 e os que já estão no segmento S1, podemos obter um dos seguintes resultados:
     - O elemento a ser inserido é menor do que aquele com o qual se está comparando. Neste caso, este é movido uma posição para a direita, deixando vaga a posição que anteriormente ocupava
     - O elemento a ser inserido é maior ou igual àquele que se está comparando. Neste caso, fazemos a inserção do elemento na posição vaga, a qual corresponde à sua posição correta no segmento S1
     - Se o elemento a ser inserido é maior que todos do segmento S1, a inserção corresponde a deixá-lo na posição que já ocupava em S2
     - Após cada inserção, a fronteira entre os dois segmentos é deslocado uma posição para a direita, indicando, com isto, que o segmento ordenado ganhou um elemento e o não ordenado perdeu um
  5. O processo prossegue até que todos os elementos de S2 tenham sido transferidos para S1

Ordenação por Inserção: Exemplo (4)

Exemplo Ilustrativo:

i	0	1	2	3	4	5
Vet[i]	60	30	40	50	90	80
	S1			**S2**		

Primeira Iteração:

i	0	1	2	3	4	5
Vet[i]	60	30	40	50	90	80
	S1			**S2**		

i	0	1	2	3	4	5
Vet[i]	30	60	40	50	90	80
	S1			**S2**		

Ordenação por Inserção: Exemplo (5)

Segunda Iteração:

i	0	1	2	3	4	5
Vet[i]	30	40	60	50	90	80

S1 S2

Terceira Iteração:

i	0	1	2	3	4	5
Vet[i]	30	40	50	60	90	80

S1 S2

Quarta Iteração:

i	0	1	2	3	4	5
Vet[i]	30	40	50	60	90	80

S1 S2

Quinta Iteração:

i	0	1	2	3	4	5
Vet[i]	30	40	50	60	80	90

Ordenação por Inserção: Mais um Exemplo (6)

Segundo Exemplo Ilustrativo:

i	0	1	2	3	4	5	6	7
Vet[i]	44	55	12	42	94	18	06	67

Solução:

i	0	1	2	3	4	5	6	7	Iteração
Vet[i]	44	55	12	42	94	18	06	67	-
Vet[i]	44	55	12	42	94	18	06	67	1
Vet[i]	12	44	55	42	94	18	06	67	2
Vet[i]	12	42	44	55	94	18	06	67	3
Vet[i]	12	42	44	55	94	18	06	67	4
Vet[i]	12	18	42	44	55	94	06	67	5
Vet[i]	06	12	18	42	44	55	94	67	6
Vet[i]	06	12	18	42	44	55	67	94	7

Ordenação por Inserção: Pseudocódigo (7)

```
ORDENA
1   para j ← 2 até n faça
2       chave ← A[j]
3       ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
4       i ← j − 1
5       enquanto i ≥ 1  e A[i] > chave faça
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← chave
```

Ordenação por Inserção: Complexidade de Tempo (8)

ORDENA	Custo	# execuções
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	?
2 $chave \leftarrow A[j]$	c_2	?
3 $\triangleright$ Insere $A[j]$ em $A[1 \ldots j-1]$	0	?
4 $i \leftarrow j-1$	c_4	?
5 **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_5	?
6 $A[i+1] \leftarrow A[i]$	c_6	?
7 $i \leftarrow i-1$	c_7	?
8 $A[i+1] \leftarrow chave$	c_8	?

- A constante $\mathbf{c_k}$ representa o **custo (tempo)** de cada execução da linha **k**
- Denote por $\mathbf{t_j}$ o número de vezes que o teste no laço **enquanto** (linha 5) é feito para aquele valor de **j**

Ordenação por Inserção: Complexidade de Tempo (9)

ORDENA	Custo	Vezes
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $chave \leftarrow A[j]$	c_2	$n-1$
3 $\triangleright$ Insere $A[j]$ em $A[1 \ldots j-1]$	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] \leftarrow chave$	c_8	$n-1$

- A constante $\mathbf{c_k}$ representa o **custo (tempo)** de cada execução da linha **k**
- Denote por $\mathbf{t_j}$ o número de vezes que o teste no laço **enquanto** (linha 5) é feito para aquele valor de **j**

Ordenação por Inserção: Complexidade de Tempo (10)

- **Tempo de Execução Total T(n) da Ordenação por Inserção:**
  - Soma dos tempos de execução de cada uma das linhas do algoritmo, ou seja:

$$\begin{aligned} T(n) = \; & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ & + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) \\ & + c_8(n-1) \end{aligned}$$

  - Como se vê, entradas de **tamanho igual** (i.e., mesmo valor de n), podem apresentar **tempos de execução diferentes** já que o valor de T(n) depende dos valores dos t_j

Ordenação por Inserção: Complexidade de Tempo (11)

- **T(n) no Melhor Caso da Ordenação por Inserção:**
  - O array já está ordenado
  - Para j = 2, ... , n temos A[i] ≤ chave na linha 5 quando i=j−1. Assim, $t_j = 1$ para j = 2, ... , n

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

  - Este tempo de execução é da forma **an + b** para constantes **a** e **b** que dependem apenas dos $\mathbf{c_i}$.
  - Portanto, no **melhor caso**, o T(n) é uma **função linear**

Ordenação por Inserção: Complexidade de Tempo (12)

- **T(n) no Pior Caso da Ordenação por Inserção:**
  - O array está em ordem decrescente
  - Para inserir a chave em A[1 ... j−1], temos que compará-la com todos os elementos neste sub(array). Assim, $t_j = j$ para $j = 2, \ldots, n$
  - Lembrem-se que:

Soma dos Termos de uma P.A Finita

$$s_n = \frac{(a_1 + a_n).n}{2}$$

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

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

Ordenação por Inserção: Complexidade de Tempo (13)

- **T(n) no Pior Caso da Ordenação por Inserção:**

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$$

- Este tempo de execução é da forma **$an^2 + bn + c$**, onde **a, b** e **c** são constantes que dependem apenas dos **c_i**.
- Portanto, no **pior caso**, o T(n) é uma **função quadrática**

Ordenação por Inserção: Análise da Complexidade

- Principais pontos a se destacar:
  - É de simples implementação, leitura e manutenção
  - In-place: apenas requer uma quantidade constante de O(1) espaço de memória adicional
  - Estável: Não muda a ordem relativa de elementos com valores iguais
  - Útil para pequenas entradas
  - É um bom método quando se desejar adicionar poucos elementos em um arquivo já ordenado, pois seu custo é linear
  - É o método a ser utilizado quando o arquivo está "quase" ordenado
  - Alto custo de movimentação de elementos no array

- Análise da Complexidade do Algoritmo:
  - **Pior Caso:**
    - T(n): $O(n^2)$
    - Justificativa: Array está em ordem decrescente
  - **Melhor Caso:**
    - T(n): O(n)
    - Justificativa: Array já encontra-se ordenado

Ordenação por Inserção: Implementação 01 (14)

```
/*InsertionSort01(): Função que ordena um array
considerando o método de ordenação por inserção */
void InsertionSort01(int Vet[])
{
   int i, j, chave;

  for(j =1; j < n; j++)
  {
    chave = Vet[j];
     i = j -1;
     while ((i >= 0) && (Vet[i] > chave))
     {
              Vet[i +1] = Vet[i];
               i = i -1;
     }
     Vet[i+1] = chave;
  }
}
```

```
ORDENA
1   para j ← 2 até n faça
2       chave ← A[j]
3       ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
4       i ← j − 1
5       enquanto i ≥ 1  e A[i] > chave faça
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← chave
```

j: índice do segmento ordenado
i: índice para encontrar a posição de inserção
chave: elemento chave analisado

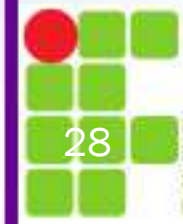
Algoritmo Importante!!

Ordenação por Inserção: Implementação 01 (15)

```
/*InsertionSort02(): Função que ordena um array considerando o método de ordenação por
   inserção */
void InsertionSort(int Vet[])
{
    int i, j, aux;

   for(i=1; i<n; i++) {
      for(j=i; j>0; j--){
         if(Vet[j] < Vet[j-1]) {
            aux = Vet[j-1];
            Vet[j-1] = Vet[j];
            Vet[j] = aux;
         }
      }
   }
}
```

i: índice do segmento ordenado
j: índice do segmento não ordenado
aux: variável auxiliar para troca

Ordenação por Inserção: Binária



Met. Ord. Inserção: Ordenação Binária (1)

- O algoritmo de inserção direta é facilmente aperfeiçoado, observando-se o segmento S1. Portanto, pode-se utilizar um método mais rápido para determinar o ponto correto de inserção

- **Busca Binária:**
  - Considere o array a seguir, com oito valores, e a sua divisão inicial em dois segmentos S1 (ordenado) e S2 (desordenado):

i	0	1	2	3	4	5	6	7
Vet[i]	06	12	18	42	55	67	30	11

L R

  - Podemos, então, efetuar a busca de um dado valor x no S1. Observe que o segmento S1 está indicado pelos limites L e R, inferior e superior, respectivamente. As operações objetivam buscar a posição do ponto de inserção mais rapidamente

Met. Ord. Inserção: Ordenação Binária (2)

S1 - Ordenado | **S2 - Desordenado**

i	0	1	2	3	4	5	6	7
Vet[i]	06	12	18	42	55	67	30	11
	L						R	

```
//Busca  Binária
    L = 0;
    R = i;
    while (L < R)
    {
      m = piso(L + R)/2;
      if (x >= Vet[m]) L = m+1;
      else R = m;
    }
```

Algoritmo Importante!!

Encontrar a posição que x = 30 deve ser inserido no Segmento S1

L	R	m
0	6	3
0	3	1
2	3	2
3	3	-

x deve ser inserido na posição de R ou L (iguais). Ou seja, x deve ser inserido na posição 3.

– Os demais passos do algoritmo, obviamente, não se modificam em sua essência. Desloca-se todos para direita, e insere 30 na posição 3, ficando:

i	0	1	2	3	4	5	6	7
Vet[i]	06	12	18	30	42	55	67	11

Met. Ord. Inserção: Ordenação Binária (3)

S1 - Ordenado | **S2 - Desordenado**

i	0	1	2	3	4	5	6	7
Vet[i]	06	12	18	30	42	55	67	11

L = 0; R = 7

```
//Busca  Binária
    L = 0;
    R = i;
    while (L < R)
    {
      m = piso(L + R)/2;
      if (x >= Vet[m]) L = m+1;
      else R = m;
    }
```

Encontrar a posição que x = 11 deve ser inserido no Segmento S1

L	R	m
0	7	3
0	3	1
0	1	0
1	1	-

x deve ser inserido na posição de R ou L (iguais). Ou seja, x deve ser inserido na posição 1.

– Desloca todos para direita, e insere 11 na posição 1, ficando:

i	0	1	2	3	4	5	6	7
Vet[i]	06	11	12	18	30	42	55	67

Met. Ord. Inserção: Ordenação Binária (4)

```
/*OrdenaBinaria(): Insere os numeros em ordem no Vet[] por pesquisa binaria. */
void OrdenaBinaria(int Vet[])
{
    //Variaveis Locais
  int i, j;
  int m, x;
  int L, R;

  for(i=1; i< N; i++)
  {
    x = Vet[i];
    //Busca do ponto R, para insercao de x
    L = 0;
    R = i;
    while (L < R)
    {
      m = (L + R)/2;
      if (x >= Vet[m]) L = m+1;
      else R = m;
    }

    //Fazendo a Movimentacao para a Insercao no ponto R
    for(j=i; j>L; j=j-1)
        Vet[j] = Vet[j-1];

    //Inserindo x na posicao de R de Vet[]
    Vet[R] = x;
  }
}
```

Entender o Funcionamento é Mais Importante que Decorar o Algoritmo!!!

Ordenação por Ordenação Binária: Análise da Complexidade

- Principais pontos a se destacar:
  - A posição correta para inserção é encontrada quando a condição L = R é satisfeita
  - Sabemos que o número de comparações necessárias para localizar a posição de um elemento utilizando-se do algoritmo de Busca Binária para um array com i elementos é lg i
  - Para a Ordenação Binária, as operações são repetidas para valores de i variando de i= (n-1), (n-2), (n-3), ..., 3, 2, 1 (L=R)

- Análise da Complexidade do Algoritmo:
  - Considerando as comparações:
  - T(n) = $\sum_{i=1}^{n-1} \lg i = \lg 1 + lg\ 2 + \lg 3 + \cdots + \lg(n-1) = \lg(n-1)!$
  - Existe uma melhora, mas em relação a apenas o número de comparações e não movimentações

Ordenação por Inserção: Shell Sort

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Met. Ord. Inserção: Shell Sort (1)

- Refinamento do Inserção Direta proposto por D. L. Shell
- A diferença é a quantidade de segmentos do array utilizados para o processo de ordenação, por este motivo o método é também conhecido como incrementos decrescentes
- Algoritmo:
  1. Dividir o array em k segmentos S_1, S_2, ... , S_k, de tal forma que cada um dos segmentos S_i, i = 1,2,...,k possua aproximadamente n/i elementos
     1. Segmento S_1: Vet[0], Vet[0+i], Vet[0+2i], ...
     2. Segmento S_2: Vet[1], Vet[1+i], Vet[1+2i], ...
     3. ....
     4. Segmento S_k: Vet[k-1], Vet[k-1+i], Vet[k-1+2i], ...
  2. Em cada passo i, realiza-se uma ordenação isolada de cada segmento, usando a inserção direta
  3. Repete-se o processo no término de cada passo, alterando o valor do incremento i para metade do valor anterior, até a execução de um certo passo com incremento i=1
- O valor do incremento i deve ser uma potência inteira de 2. Logo, o valor inicial é dado por 2^{np}, sendo np fornecido pelo usuário. Para np=3, temos:
  - Passo 1: $i = 2^{np} = 2^3 = 8$ segmentos
  - Passo 2: $i = 2^{np-1} = 2^2 = 4$ segmentos
  - Passo 3: $i = 2^{np-2} = 2^1 = 2$ segmentos
  - Passo 4: $i = 2^{np-3} = 2^0 = 1$ segmento

Met. Ord. Inserção: Shell Sort (2)

- Considere o seguinte array desordenado e o valor de np = 2. Logo, teremos de efetuar os seguintes passos:
  - Passo 1: i = 2^{np} = 2^2 = 4 segmentos
  - Passo 2: i = 2^{np-1} = 2^1 = 2 segmentos
  - Passo 3: i = 2^{np-2} = 2^0 = 1 segmento

i	0	1	2	3	4	5	6	7	8	9	10	11
Vet[i]	17	24	42	15	21	22	47	37	52	43	27	12

- **Passo 01: i = 2^{np} = 2^2 = 4 segmentos**

S1	**0**	**4**	**8**	**S2**	**1**	**5**	**9**	**S3**	**2**	**6**	**10**	**S4**	**3**	**7**	**11**
	17	21	52		24	22	43		42	47	27		15	37	12

S1	**0**	**4**	**8**	**S2**	**1**	**5**	**9**	**S3**	**2**	**6**	**10**	**S4**	**3**	**7**	**11**
	17	21	52		22	24	43		27	42	47		12	15	37

i	0	1	2	3	4	5	6	7	8	9	10	11
Vet[i]	17	22	27	12	21	24	42	15	52	43	47	37

Met. Ord. Inserção: Shell Sort (3)

i	0	1	2	3	4	5	6	7	8	9	10	11
Vet[i]	17	22	27	12	21	24	42	15	52	43	47	37

- **Passo 02: $i = 2^{np-1} = 2^1 = 2$ segmentos**

S1

0	2	4	6	8	10
17	27	21	42	52	47

S2

1	3	5	7	9	11
22	12	24	15	43	37

S1

0	2	4	6	8	10
17	21	27	42	47	52

S2

1	3	5	7	9	11
12	15	22	24	37	43

i	0	1	2	3	4	5	6	7	8	9	10	11
Vet[i]	17	12	21	15	27	22	42	24	47	37	52	43

Met. Ord. Inserção: Shell Sort (4)

i	0	1	2	3	4	5	6	7	8	9	10	11
Vet[i]	17	12	21	15	27	22	42	24	47	37	52	43

- **Passo 03:** $i = 2^{np-2} = 2^0 = 1$ **segmento**
  - Neste último passo, os elementos já estão próximos de suas reais posições, o que leva a um número menor de trocas.

i	0	1	2	3	4	5	6	7	8	9	10	11
Vet[i]	12	15	17	21	22	22	24	37	42	43	47	52

Met. Ord. Inserção: Shell Sort (5)

```
/*OrdenaShellSort(): Insere os numeros em ordem no Vet[] pelo metodo de insercao
                     através de incrementos decrescentes*/
void OrdenaShellSort(int Vet[], int Inc, int SegCorrente)
{
   //Variaveis Locais
  int i, j, x, k;

  for(i=(SegCorrente+Inc); i<N; i+=Inc)
  {
    k = SegCorrente;
    j = i - Inc;
    x = Vet[i];
    while((j>=SegCorrente)&&(k==SegCorrente))
    {
      if(x < Vet[j])
      {
        Vet[j+Inc] = Vet[j];
        j = j-Inc;
      }
      else k=j+Inc;
    }
    Vet[k] = x;
  }
}
```

```
//Chamada da Função
Np =2;
for(i=Np; i>=0; i--)
{
        Inc = (int)pow(2.0, i);
        for(SegCorrente=0; SegCorrente<Inc; SegCorrente++)
                OrdenaShellSort(Vet, Inc, SegCorrente);
}
```

Entender o Funcionamento é Mais Importante que Decorar o Algoritmo!!!

Met. Ord. Inserção: Shell Sort (6)

- Considere o seguinte vetor desordenado e o valor de np = 2. Simular a ordenação para o vetor abaixo.

- **Vetor Desordenado:**

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Vet[i]	26	17	8	15	17	3	7	52	48	19	59	69	67	95	0

- **Passo 1**: i = 2^{np} = 2^2 = 4 segmentos
  - **S1 =** Vet[0], Vet[4], Vet[8], Vet[12]
  - **S2 =** Vet[1], Vet[5], Vet[9], Vet[13]
  - **S3 =** Vet[2], Vet[6], Vet[10], Vet[14]
  - **S4 =** Vet[3], Vet[7], Vet[11]

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Vet[i]	17	3	0	15	26	17	7	52	48	19	8	69	67	95	59

Met. Ord. Inserção: Shell Sort (7)

- **Passo 2**: i = 2^{np-1} = 2^1 = 2 segmentos
  - **S1 =** Vet[0], Vet[2], Vet[4], Vet[6], Vet[8], Vet[10], Vet[12], Vet[14]
  - **S2 =** Vet[1], Vet[3], Vet[5], Vet[7], Vet[9], Vet[11], Vet[13]

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Vet[i]	0	3	7	15	8	17	17	19	26	52	48	69	59	95	67

- **Passo 3:** i = 2^{np-2} = 2^0 = 1 segmento

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Vet[i]	0	3	7	8	15	17	17	19	26	48	52	59	67	69	95

Ordenação por Shell Sort: Análise da Complexidade

- Principais pontos a se destacar:
  - A complexidade do algoritmo ainda não é conhecida
  - Ninguém foi capaz de encontrar uma fórmula fechada para a sua função de complexidade
    - A Análise contém alguns problemas matemáticos difíceis
    - Exemplo: escolher a sequência de incrementos (adotamos um padrão mas poderia ser qualquer sequencia)
  - É uma boa opção para arrays de tamanho moderado, com implementação simples
  - Não é estável
- Análise da Complexidade do Algoritmo:
  - Conjecturas referentes ao número de comparações para a sequência de Knuth:
    - Conjectura 1: $O(n^{1,25})$
    - Conjectura 2: $O(n\ (\lg n)^2)$

Simuladores de Funcionamento

- Simuladores do Funcionamento de Alguns Algoritmos de Ordenação:
  - USFCA: https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
  - VisualGo: https://visualgo.net/pt/sorting?slide=1

PANC: Projeto e Análise de Algoritmos

Aula 10: Algoritmos de Ordenação I - Inserção:
Ordenação Direta, Binária e Shell Sort

Breno Lisi Romano

Dúvidas???

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista