PANC: Projeto e Análise de Algoritmos

Aula 03: Análise do Tempo de Execução – Ordenação por Inserção e outros exemplos

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

- Revisão de Conteúdo
- Cálculo do Tempo de Execução
  - Limitantes Inferiores e Superiores
  - Custos
  - Otimalidade de um Algoritmo
  - Cálculo do Tempo de Execução e Perspectiva
- Comparando Algoritmos
- Ordenação por Inserção (*Insertion Sort*)

Recapitulando... (1)

- *"Algorithm analysis usually means 'give a big-O figure for the running time of an algorithm (Of course, a big-Θ would be even better). This can be done by getting a big-O figure for parts of the algorithm and then combining these figures using the sum and product rules for big-O.*
- *Another useful technique is to pick an elementary operation, such as additions, multiplications or comparisons, and observing that the running time of the algorithm is big-O of the number of elementary operations. Then, you can analyze the exact number of operations as function of n in the worst case."*
- **- Ian Parberry, *Problems on Algorithms***

Recapitulando... (2)

- **Importância:**
  - Os **algoritmos permeiam** toda a ciência da computação (entre outras ciências), independente da área de concentração
  - O **projeto de algoritmos** é fortemente influenciado pela **estimativa** de seu **comportamento**
  - Estamos interessados em **algoritmos eficientes**, ou pelo menos, **“bem comportados”**

- **Projeto:**
  - Antes de projetarmos um algoritmo, **analisa-se o problema**:
    - suas **características**
    - sua **complexidade**
    - **contexto** em que se encontra – o que **determinará** a exigência sobre **tempo de execução** e qualidade das soluções
  - Depois desta análise, as **decisões** se concentram em **qual tipo de algoritmo** será utilizado, quais estruturas de dados e outros detalhes de implementação

Recapitulando... (3)

- **Implementação:**
  - Uma vez tomadas todas as decisões anteriores, **implementa-se o algoritmo**
  - Qual é o **custo** de se utilizar uma **determinada implementação** específica?
  - Devemos levar em **consideração** a **memória** necessária para **armazenar** as **estruturas de dados** e os **trechos do código** – quantas vezes cada um será executado?

- **Analisando Algoritmos:**
  - Analisar o **comportamento assintótico** de um algoritmo (ou implementação) de acordo com o **tamanho da entrada**
  - Se **aplicarmos** a mesma **métrica** a **diferentes algoritmos** para um **mesmo problema**, **podemos compará-los** de uma maneira adequada

Recapitulando... (4)

- **Analisando Algoritmos:**
  - Análise **Teórica**
  - Na prática, **outros fatores** podem influenciar o **desempenho** de uma implementação:
    - **Otimizações** realizadas pelo **compilador**
    - **Características** do **sistema operacional**
    - **Características** de **hardware**
  - Algumas **simplificações** são feitas nesta análise teórica, como veremos a seguir

- **Comparando Algoritmos:**
  - Recomenda-se **comparar algoritmos** com **complexidade** dentro de uma mesma **ordem de grandeza** por meio de experimentos computacionais
  - Desta forma, os **custos reais** e outros não aparentes se **tornam claros**

Recapitulando... (5)

- **Perspectivas - Definição:**
  - Além do ambiente computacional, o comportamento de um algoritmo pode variar de acordo com o comportamento da entrada (tamanho, estrutura, etc.), o que gera diferentes **Perspectivas**
- **Melhor Caso:**
  - A **entrada** está organizada de maneira que o algoritmo levará o **tempo mínimo** para resolver o problema
- **Pior Caso:** ← **Foco da Análise**
  - A **entrada** está organizada de maneira que o algoritmo levará o **tempo máximo** para resolver o problema
- **Caso Médio:**
  - A **entrada** está organizada de maneira que o algoritmo levará um **tempo médio** para resolver o problema

Como Medir?

- Consideramos um **conjunto de instruções** com **custos** especificados, normalmente, **só as instruções mais significativas**
- Definimos uma **função de custo** ou **função de complexidade T**
- **T(n)** é a **medida de custo da execução** de um **algoritmo** para uma instância de **tamanho n**:
  - A função de **complexidade de tempo T(n)** mede o tempo necessário para executar um algoritmo **(número de instruções)**
    - **Não o tempo medido no relógio, mas quantas vezes operações relevantes serão executadas**
  - A função de **complexidade de espaço T(n)** mede a **quantidade de memória** necessária para executar um algoritmo

Cálculo do Tempo de Execução (1)

- **Limitante Inferior:**
  - Dado um determinado problema P, chamamos **de limite inferior** (ou ***lower bound***) **LB(P)** a **complexidade mínima** necessária para resolvê-lo

- **Limitante Superior:**
  - Dado um determinado problema P, chamamos de **limite superior** (ou **upper bound**) **UB(P)** a **complexidade** do **melhor algoritmo** conhecido que o resolve

- **Limitantes Inferiores e Superiores:**
  - Um determinado **problema** é considerado **computacionalmente resolvido** se:
    - **UB(P) pertence** ao mesmo **domínio** de **LB(P)**

Cálculo do Tempo de Execução (2)

- **Problema computacional:** Dado um array A de n números inteiros, determine o maior valor entre eles
- **Como calcular, em função de n, o número máximo de operações** realizadas por este algoritmo?

```
int arrayMax(int A[ ], int n)
{
    int currentMax = A[0];
    for(int i=1; i<n; i++){
        if(A[i] > currentMax)
            currentMax = A[i];
    }
    return currentMax;
}
```

Cálculo do Tempo de Execução (3)

- **Custos:**
  - A **contribuição** de cada **instrução** para o **tempo de execução** é o **produto** de seu **custo individual** e o **número de vezes** que é **executada**:
    - Uma operação com custo c_1 executada uma vez contribui com c_1
    - Uma operação com custo c_2 executada n vezes contribui com $c_2.n$
    - Um **laço** (for, while) que termina de maneira usual contribui com o **produto** de uma **constante** e a **quantidade de vezes** que foi executado
      - Operações de **atribuição**, **incremento** e **comparação** são contados como uma **única constante**
      - A **comparação** do **laço** é sempre **executada uma vez mais**, para determinar o seu fim
        - Por exemplo, um laço de 0 até n-1 é executado n vezes

Cálculo do Tempo de Execução (4)

- **Custos:**
  - Um **desvio condicional** (if, switch) fora do cabeçalho de um laço é contado como **tempo constante**
  - Uma **chamada** para uma **função** tem **complexidade correspondente** à **complexidade da execução da função**
  - Uma **função recursiva** tem sua **complexidade** definida em termos da **recorrência associada**
  - **Instruções** contidas **dentro de laços** são **executadas repetidas vezes**, o que deve ser levado em consideração
  - O **tempo de execução T(n)** é, portanto, a **soma destes produtos** referentes a **cada instrução** do algoritmo

- Obs: Basicamente entendemos Ian Parberry agora!!!

Cálculo do Tempo de Execução (5)

```
int arrayMax(int A[ ], int n)
{
    int currentMax = A[0];              1
    for(int i=1; i<n; i++){1 + 2.(n-1) + 1 *
        if(A[i] > currentMax)           (n-1)
            currentMax = A[i];          (n-1)
    }
    return currentMax;                  1
}
```

* 01 atribuição inicial e repete n-1 vezes (01 comparação, 01 incremento), no máximo. Adicionalmente, 01 última comparação

- **Análise Assintótica:**
  - No **pior caso**, são realizadas **4+4.(n-1)** operações (para n ≥ 1), logo, o **algoritmo** é **Linear (4.n)**

Cálculo do Tempo de Execução (6)

- **Considerações:**
  - O algoritmo **arrayMax** executa **4n operações primitivas**, excluindo os termos de mais baixa ordem
  - Sejam a e b **os tempos de execução** das instruções **mais rápida** e **mais lenta** da arquitetura utilizada, respectivamente
  - Seja **T(n)** o **tempo real** de execução do **pior caso** de **arrayMax**
  - Temos que:
    - $a \cdot 4n \leq T(n) \leq b \cdot 4n \rightarrow$ T(n) é delimitada por duas funções lineares
  - A **linearidade** de **T(n)** é uma **propriedade intrínseca** de **arrayMax**
    - Por exemplo, o **ambiente de hardware** ou **software** apenas **alterariam** T(n) por uma **constante**, porém, a **linearidade se manteria**
    - **Não existe um algoritmo melhor que linear para arrayMax()**
- **Propriedade**:
  - Cada **algoritmo** possui uma **taxa de crescimento** que lhe é **intrínseca**

Otimalidade de um Algoritmo

- **Teorema - Limitante Inferior para Encontrar o Maior Elemento:**
  - Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos ($n \geq 1$), faz pelo menos n-1 comparações

- **Prova:**
  - Cada um dos n-1 elementos deve ser mostrado, por meio de comparações, ser menor do que algum outro elemento, logo, n-1 comparações são necessárias

- **Otimalidade:**
  - **Se o limitante inferior para encontrar o menor elemento é igual ao limitante superior → o problema é computacionalmente resolvido e o algoritmo é ótimo**

Outro Exemplo: Cálculo do Tempo de Execução – Maior e Menor Elemento (1)

- **Problema computacional:** encontrar o maior e o menor elemento de um array de inteiros A, de tamanho n, com n ≥ 1

```
int maxMin1(int A[ ], int n, int max, int min)
{
    max = A[0];
    min = A[0];
    for(int i=1; i<n; i++){
        if(A[i] > max)
            max = A[i];
        if(A[i] < min)
            min = A[i];
    }
}
```

Outro Exemplo: Cálculo do Tempo de Execução – Maior e Menor Elemento (2)

- Vamos analisar o algoritmo para **determinar o T(n)** baseando-se apenas no número de **comparações** entre os elementos de A

```
int maxMin1(int A[ ], int n, int max, int min)
{
    max = A[0];
    min = A[0];
    for(int i=1; i<n; i++){
        if(A[i] > max)
            max = A[i];
        if(A[i] < min)
            min = A[i];
    }
}
```

2.(n-1) comparações

- **Análise:** O número de comparações é T(n) = 2.(n-1) para n>0, para o melhor caso, pior caso e caso médio → este algoritmo pode ser melhorado

Outro Exemplo: Cálculo do Tempo de Execução – Maior e Menor Elemento (3)

- **Conseguimos melhorar?**

```
int maxMin2(int A[ ], int n, int max, int min)
{
    max = A[0];
    min = A[0];
    for(int i=1; i<n; i++){
        if(A[i] > max)
            max = A[i];
        else if(A[i] < min)
            min = A[i];
    }
}
```

? comparações

- **Para esta nova versão, as perspectivas mudaram?**

Outro Exemplo: Cálculo do Tempo de Execução – Maior e Menor Elemento (4)

```
int maxMin2(int A[ ], int n, int max, int min)
{
    max = A[0];
    min = A[0];
    for(int i=1; i<n; i++){
        if(A[i] > max)
            max = A[i];
        else if(A[i] < min)
            min = A[i];
    }
}
```

- **Melhor Caso:**
  - Os elementos estão em ordem crescente, logo, o número de comparações é **T(n) = n-1**
- **Pior Caso:**
  - Os elementos estão em ordem decrescente, logo, o número de comparações é **T(n) = 2.(n-1)**

Outro Exemplo: Cálculo do Tempo de Execução – Maior e Menor Elemento (5)

- **Caso Médio:**
  - Supõe-se uma **distribuição de probabilidades** sobre o conjunto de **entradas** de tamanho n
    - É comum supor uma distribuição em que quaisquer entradas são **igualmente prováveis**, embora isso não seja sempre verdade
  - Esta análise geralmente é mais elaborada do que as duas anteriores
  - Podemos considerar uma distribuição dos elementos de A de maneira que A[i] será maior do que a variável max na metade dos casos
    - Ou seja, o primeiro if será executado (n−1) vezes, e o else $\frac{(n-1)}{2}$ vezes
  - Portanto, o **número de comparações** é $\boldsymbol{T(n) = (n-1) + \frac{n-1}{2} = \frac{3n}{2} - \frac{3}{2}}$, para n>0

- Este não é um algoritmo ótimo, embora seja melhor do que o primeiro. Vejamos uma terceira versão de algoritmo! **Alguém consegue pensar em uma melhora?**

Outro Exemplo: Cálculo do Tempo de Execução – Maior e Menor Elemento (6)

- **maxMin3() - Lógica:**
  - Compare os elementos de A aos pares, separando-os em dois subconjuntos:
    - A− : conjunto dos menores elementos
    - A+ : conjunto dos maiores elementos
  - Obtenha o maior elemento comparando os $\left\lceil \frac{n}{2} \right\rceil - 1$ elementos do conjunto A+
  - Obtenha o menor elemento comparando os $\left\lceil \frac{n}{2} \right\rceil - 1$ elementos do conjunto A−
- Qual é a complexidade de cada perspectiva deste algoritmo?

```
int maxMin3(int A[ ], int n, int max, int min)
{
    if(n%2 != 0){
        A[n+1] = A[n];
        n = n+1;
    }
    max = A[0];
    min = A[1];
    if(A[0] < A[1]){
        max = A[1];
        min = A[0];
    }
    for(int i=2; i<n-1; i+=2){
        if(A[i] > A[i+1]){
            if(A[i] > max)
                max = A[i];
            if(A[i+1] < min)
                min = A[i+1];
        }else{
            if(A[i] < min)
                min = A[i];
            if(A[i+1] > max)
                max = A[i+1];
        }
    }
}
```

Quando o n (tamanho do array) é ímpar,
o último elemento é duplicado, por simplicidade

Identifica o maior (max) e o menor (min) elemento
do array entre os dois primeiros

Os elementos são comparados dois a dois:
- Os elementos maiores são comparados com max
- Os elementos menores são comparados com min

```
int maxMin3(int A[ ], int n, int max, int min)
{
    if(n%2 != 0){
        A[n+1] = A[n];
        n = n+1;
    }
    max = A[0];
    min = A[1];
    if(A[0] < A[1]){
        max = A[1];
        min = A[0];
    }
    for(int i=2; i<n-1; i+=2){
        if(A[i] > A[i+1]){
            if(A[i] > max)
                max = A[i];
            if(A[i+1] < min)
                min = A[i+1];
        }else{
            if(A[i] < min)
                min = A[i];
            if(A[i+1] > max)
                max = A[i+1];
        }
    }
}
```

Qual o número de comparações?

- Quais Comparações que importam?

- Análise da Complexidade:
  - $T(n) = 1 + \frac{(n-2)}{2} + \frac{(n-2)}{2} + \frac{(n-2)}{2}$
  - $\mathbf{T(n) = \frac{3n}{2} - 2}$

Vamos analisar o Melhor, o Pior e o Caso Médio?????

É tudo igual!

Outro Exemplo: Cálculo do Tempo de Execução – Maior e Menor Elemento (9)

- Comparemos as três versões de algoritmos apresentadas:
  - De uma maneira geral, **maxMin2** e **maxMin3** são superiores a maxMin1
  - **maxMin2** é superior a **maxMin3** no melhor caso
  - **maxMin3** é superior a **maxMin2** com relação ao pior caso
  - **maxMin2** e **maxMin3** são bastante próximos quando ao caso médio

- Qual **algoritmo** você **escolheria**?

	Melhor Caso	Caso Médio	Pior Caso
maxMin1	$2(n\text{-}1)$	$2(n\text{-}1)$	$2(n\text{-}1)$
maxMin2	$n\text{-}1$	$\frac{3n}{2} - \frac{3}{2}$	$2(n\text{-}1)$
maxMin3	$\frac{3n}{2} - 2$	$\frac{3n}{2} - 2$	$\frac{3n}{2} - 2$

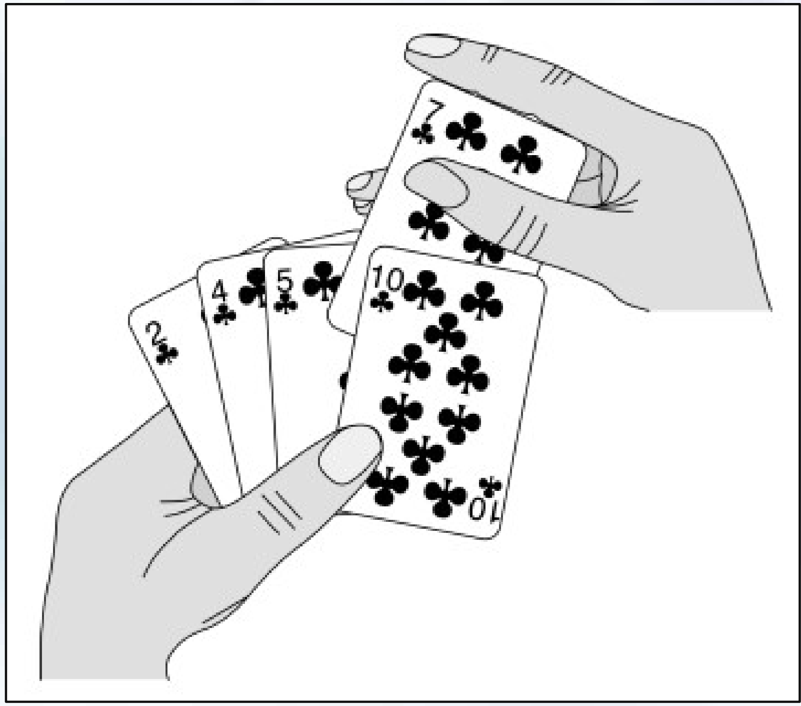
Continuando os Cálculos de Tempo de Execução.....

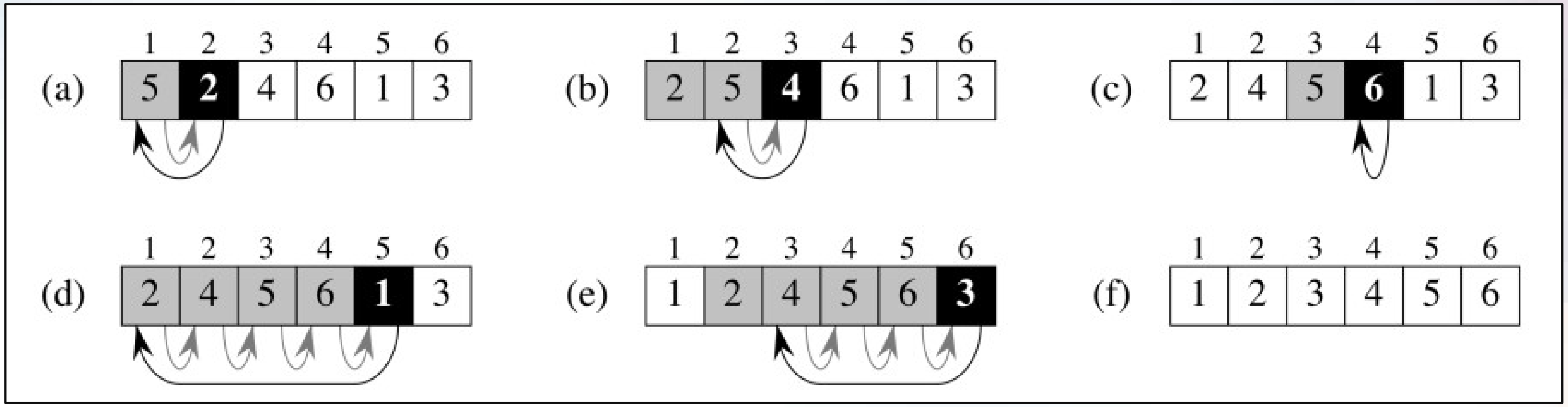
ORDENAÇÃO POR INSERÇÃO (*INSERTION SORT*)

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 $a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_n$
- Vamos começar estudando o **algoritmo** de **Ordenação por Inserção** (***Insertion Sort***)

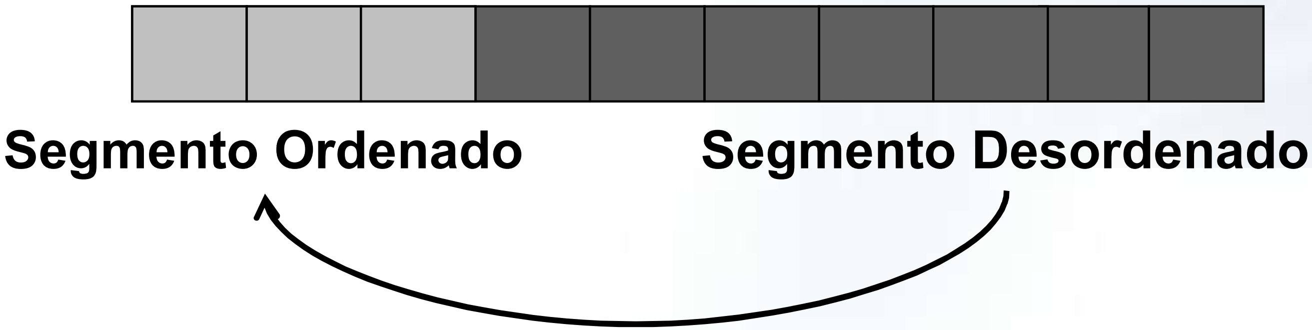
Ordenação por Inserção: Ilustração (1)





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	<u>30</u>	40	50	90	80

S1 S2

i	0	1	2	3	4	5
Vet[i]	30	60	<u>40</u>	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	<u>55</u>	12	42	94	18	06	67	-
Vet[i]	44	55	<u>12</u>	42	94	18	06	67	1
Vet[i]	12	44	55	<u>42</u>	94	18	06	67	2
Vet[i]	12	42	44	55	<u>94</u>	18	06	67	3
Vet[i]	12	42	44	55	94	<u>18</u>	06	67	4
Vet[i]	12	18	42	44	55	94	<u>06</u>	67	5
Vet[i]	06	12	18	42	44	55	94	<u>67</u>	6
Vet[i]	06	12	18	42	44	55	67	94	7

Ordenação por Inserção: Pseudocódigo (7)

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

Ordenação por Inserção: Análise da Complexidade (8)

- **O que é importante analisar?**
  - **Finitude**: o algoritmo para?
  - **Corretude**: o algoritmo faz o que promete?
  - **Complexidade de Tempo**: quantas instruções são necessárias no pior caso para ordenar os **n** elementos?

Ordenação por Inserção: Finitude (9)

- No **laço enquanto (linha 5)**, o valor de **i diminui a cada iteração** e o valor inicial é **i = j−1 ≥ 1** → sua execução para em algum momento por causa do teste condicional i ≥ 1
- O **laço na linha 1** evidentemente **para** (o contador j atingirá o valor n + 1 após n − 1 iterações)
- Portanto, o **algoritmo para**!

```
ORDENA
1    para j ← 2 até n faça
     ...
4        i ← j − 1
5        enquanto i ≥ 1 e A[i] > chave faça
6            ...
7            i ← i − 1
8        ...
```

Ordenação por Inserção: Corretude (10)

- **Invariante de Laços e Provas de Corretude:**
  - **Definição**: é uma **propriedade** que **relaciona** as **variáveis** do algoritmo **a cada execução completa do laço**
  - Ele deve ser escolhido de modo que, ao término do laço, tenha-se uma propriedade útil para mostrar a corretude do algoritmo
  - A **prova de corretude** de um algoritmo **requer** que **sejam encontrados e provados invariantes** dos vários laços que o compõem
  - Em geral, é mais **difícil descobrir um invariante apropriado** do que **mostrar sua validade** se ele for dado de bandeja. . .

Ordenação por Inserção: Corretude (11)

- **Invariante Principal de ORDENA (i1):**
  - No começo de cada iteração do laço **para** das linha 1–8, o sub(array) **A[1... j−1]** está ordenado

```
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: Corretude (12)

- A **estratégia** "típica" para mostrar a corretude de um algoritmo iterativo através de invariantes segue os seguintes passos:
  1. Mostre que o invariante **vale** no **início** da **primeira iteração** (trivial, em geral)
  2. Suponha que o invariante **vale** no **início** de **uma interação qualquer** e prove que ele **vale** no **início** da **próxima iteração**
  3. Conclua que se o algoritmo **para** e o invariante **vale** no **início da última iteração**, então o **algoritmo** é **correto**
- Note que (1) e (2) implicam que o invariante vale no início de qualquer iteração do algoritmo. Isto é similar ao método **de indução matemática**!

Ordenação por Inserção: Corretude (13)

- Vamos verificar a **corretude do algoritmo** de **ordenação por inserção** usando a técnica de **prova por invariantes de laços**
- **Invariante Principal (i1):**
  - No começo de cada iteração do laço **para** das linha 1–8, o sub(array) **A[1... j−1]** está ordenado

1						*j*				*n*
20	25	35	40	44	55	38	99	10	65	50

  - Suponha que o invariante vale
  - Então a corretude do algoritmo é “evidente”. Por que?
    - No início da última iteração temos **j = n + 1**. Assim, do invariante segue que o (sub)array A[1 . . . n] está ordenado!

Ordenação por Inserção: Corretude (14)

- **Um Invariante Mais Preciso (i1’):**
  - No começo de cada iteração do laço **para** das linha 1–8, o sub(array) **A[1... j−1]** é uma permutação ordenada do sub(array) original **A[1... j−1]**

```
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: Corretude (15)

- **Validade na primeira iteração**: temos **j=2** e o invariante simplesmente afirma que **A[1...1]** está ordenado → Evidente

- **Validade de uma iteração para a seguinte**: O algoritmo **empurra** os elementos maiores que a **chave** para seus lugares corretos e ela é colocada no **espaço vazio**

- **Corretude do algoritmo**: na última iteração, temos **j=n+1** e logo **A[1...n]** está ordenado com os elementos originais do array → **O algoritmo é Correto!**

Ordenação por Inserção: Complexidade de Tempo (16)

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 (17)

ORDENA	Custo	Vezes
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad$ *chave* $\leftarrow A[j]$	c_2	$n-1$
3 $\quad \triangleright$ Insere $A[j]$ em $A[1 \ldots j-1]$	0	$n-1$
4 $\quad i \leftarrow j-1$	c_4	$n-1$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] >$ *chave* **faça**	c_5	$\sum_{j=2}^{n} t_j$
6 $\qquad A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $\qquad i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $\quad A[i+1] \leftarrow$ *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 (18)

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

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

  - 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 (19)

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

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	$n-1$

  - 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 (20)

- **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, $\mathbf{t_j = j}$ para j = 2, ... , n
  - Lembrem-se que:

ORDENA	Custo	Vezes
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 *chave* $\leftarrow A[j]$	c_2	$n-1$
3 ▷ 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	$n-1$

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 (21)

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

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

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

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

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

- Este tempo de execução é da forma **a**n^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: Implementação 01 (22)

```
/*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

```
para j ← 2 até n faça
    chave ← A[j]
    ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
    i ← j − 1
    enquanto i ≥ 1 e A[i] > chave faça
        A[i + 1] ← A[i]
        i ← i − 1
    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 02 (23)

```
/*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

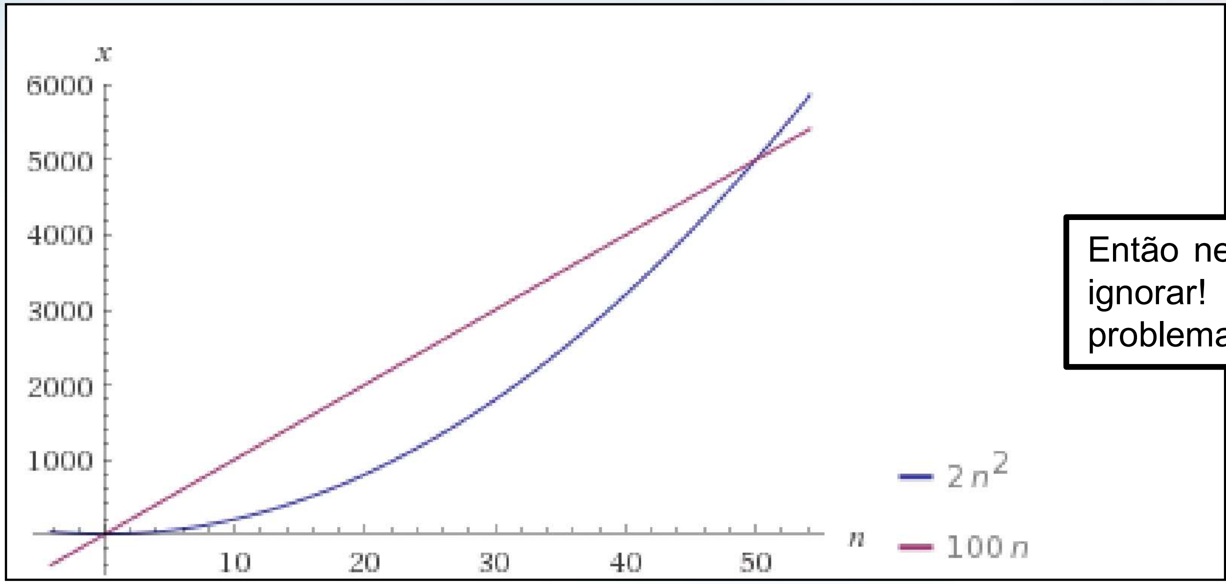
Algoritmo Importante!!

Conclusão: Comparando Algoritmos (1)

- **Comparação Justa?**
  - Podemos **comparar algoritmos** utilizando as **funções de complexidade** de espaço e tempo, **negligenciando as constantes** de proporcionalidade
  - Desta forma, um algoritmo **n^2** é pior que outro **n**, ambos para o mesmo problema
  - **Contudo, as constantes de proporcionalidade podem revelar fatos escondidos**

Conclusão: Comparando Algoritmos (2)

- **Exemplo:**
  - Suponha dois algoritmos: um exige 100.n unidades de tempo e outro exige $2.n^2$ unidades de tempo
  - Dependendo do tamanho do problema, o melhor algoritmo pode variar:
    - Para n < 50, o segundo algoritmo é melhor que o primeiro
    - Se a quantidade de dados for pequena, é preferível optar pelo segundo
    - Entretanto, o tempo de execução do segundo algoritmo cresce mais rapidamente que o tempo de execução do primeiro



Então nem sempre podemos ignorar! Depende do problema. Mas...

Conclusão: Comparando Algoritmos (3)

- O **estudo assintótico** nos permite “jogar para debaixo do tapete” os **valores das constantes envolvidas**, i.e., aquilo que independe do tamanho da entrada
- Considere $3n^2 + 10n + 50$:

n	$3n^2 + 10n + 50$	$3n^2$	Diferença percentual
64	12978	12288	5,32%
128	50482	49152	2,63%
512	791602	786432	0,65%
1024	3156018	3145728	0,33%
2048	12603442	12582912	0,16%
4096	50372658	50331648	0,08%
8192	201408562	201326592	0,04%
16384	805470258	805306368	0,02%
32768	3221553202	3221225472	0,01%

- **$3n^2$** é o **termo dominante** para **n muito grande** → **Concentrar** nos termos **dominantes**

PANC: Projeto e Análise de Algoritmos

Aula 03: Calculando Tempo de Execução – Ordenação por Inserção e outros exemplos

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