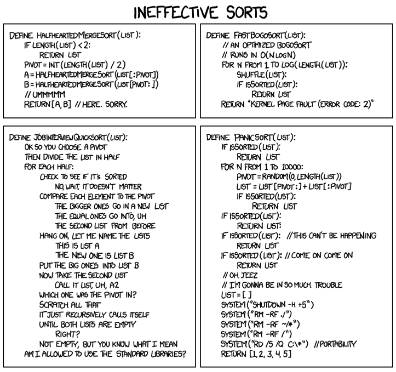
Ordenação



Fonte: http://xkcd.com/1185/

PF 8

http://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html

Ordenação

$v[0:n]$ é crescente se $v[0] \leq \cdots \leq v[n-1]$.

Problema: Rearranjar um lista $v[0:n]$ de modo que ele fique crescente.

Entra:

0										n
33	55	33	44	33	22	11	99	22	55	77

Ordenação

$v[0:n]$ é crescente se $v[0] \leq \cdots \leq v[n-1]$.

Problema: Rearranjar um lista $v[0:n]$ de modo que ele fique crescente.

Entra:

0										n
33	55	33	44	33	22	11	99	22	55	77

Sai:

0										n
11	22	22	33	33	33	44	55	55	77	99

Ordenação por inserção (iteração)

$x = 38$

0						i				n
20	25	35	40	44	55	38	99	10	65	50

Ordenação por inserção (iteração)

$x = 38$

0					j	i				n
20	25	35	40	44	55	38	99	10	65	50

Ordenação por inserção (iteração)

$x = 38$

0					j	i				n
20	25	35	40	44	55	38	99	10	65	50

0				j		i				n
20	25	35	40	44		55	99	10	65	50

Ordenação por inserção (iteração)

$x = 38$

0					j	i				n
20	25	35	40	44	55	38	99	10	65	50

0				j		i				n
20	25	35	40	44		55	99	10	65	50

0			j			i				n
20	25	35	40		44	55	99	10	65	50

Ordenação por inserção (iteração)

$x = 38$

0					j	i				n
20	25	35	40	44	55	38	99	10	65	50

0				j		i				n
20	25	35	40	44		55	99	10	65	50

0			j			i				n
20	25	35	40		44	55	99	10	65	50

0		j				i				n
20	25	35		40	44	55	99	10	65	50

Ordenação por inserção (iteração)

$x = 38$

0					j	i				n
20	25	35	40	44	55	38	99	10	65	50

0				j		i				n
20	25	35	40	44		55	99	10	65	50

0			j			i				n
20	25	35	40		44	55	99	10	65	50

0		j				i				n
20	25	35		40	44	55	99	10	65	50

0		j				i				n
20	25	35	38	40	44	55	99	10	65	50

Ordenação por inserção

x	0							i			n
99	20	25	35	38	40	44	55	99	10	65	50

Ordenação por inserção

x	0							i			n
99	20	25	35	38	40	44	55	99	10	65	50

x	0								i		n
10	20	25	35	38	40	44	55	99	10	65	50

Ordenação por inserção

x	0							i			n
99	20	25	35	38	40	44	55	99	10	65	50

x	0								i		n
10	10	20	25	35	38	40	44	55	99	65	50

Ordenação por inserção

x	0							i				n
99	20	25	35	38	40	44	55	99	10	65	50	

x	0								i			n
10	10	20	25	35	38	40	44	55	99	65	50	

x	0									i		n
65	10	20	25	35	38	40	44	55	99	65	50	

Ordenação por inserção

x	0							i				n
99	20	25	35	38	40	44	55	99	10	65	50	

x	0								i			n
10	10	20	25	35	38	40	44	55	99	65	50	

x	0									i		n
65	10	20	25	35	38	40	44	55	65	99	50	

Ordenação por inserção

x	0							i				n
99	20	25	35	38	40	44	55	99	10	65	50	

x	0								i			n
10	10	20	25	35	38	40	44	55	99	65	50	

x	0									i		n
65	10	20	25	35	38	40	44	55	65	99	50	

x	0										i	n
50	10	20	25	35	38	40	44	55	65	99	50	

Ordenação por inserção

x	0							i				n
99	20	25	35	38	40	44	55	99	10	65	50	

x	0								i			n
10	10	20	25	35	38	40	44	55	99	65	50	

x	0									i		n
65	10	20	25	35	38	40	44	55	65	99	50	

x	0										i	n
50	10	20	25	35	38	40	44	50	55	65	99	

insercao

Função rearranja v[0 : n] em ordem crescente.

```
def insercao(v):
    n = len(v)
    for i in range(1,n):  # /*A*/
        x = v[i]
        j = j - 1
        while j >= 0 and v[j] > x:
            v[j+1] = v[j]
            j -= 1
        v[j+1] = x
```

O algoritmo faz o que promete?

Relação **invariante** chave:

♡ (i0) Em /*A*/ vale que: v[0 : i] é crescente.

0						i					n
20	25	35	40	44	55	38	99	10	65	50	

O algoritmo faz o que promete?

Relação **invariante** chave:

♡ (i0) Em `/*A*/` vale que: $v[0:i]$ é crescente.

0						i					n
20	25	35	40	44	55	38	99	10	65	50	

Supondo que a invariante vale.
Correção do algoritmo é evidente.

No início da última iteração tem-se que $i = n$.
Da invariante conclui-se que $v[0:n]$ é crescente.

Mais invariantes

Na linha 4, antes de "`j >= 0...`", vale que:

(i1) $v[0:j{+}1]$ e $v[j{+}2:i{+}1]$ são crescentes

(i2) $v[0:j{+}1] \leq v[j{+}2:i{+}1]$

(i3) $v[j{+}2:i{+}1] > x$

x	0		j				i					n
38	20	25	35		40	44	55	99	10	65	50	

Mais invariantes

Na linha 4, antes de "`j >= 0...`", vale que:

(i1) $v[0:j{+}1]$ e $v[j{+}2:i{+}1]$ são crescentes

(i2) $v[0:j{+}1] \leq v[j{+}2:i{+}1]$

(i3) $v[j{+}2:i{+}1] > x$

x	0		j				i					n
38	20	25	35		40	44	55	99	10	65	50	

invariantes (i1),(i2) e (i3)
+ condição de parada do `while` da linha 4
+ atribuição da linha 6 ⇒ validade (i0)

Verifique!

Correção de algoritmos iterativos

Estrutura "típica" de demonstrações da correção de algoritmos iterativos através de suas relações invariantes consiste em:

1. verificar que a relação vale no início da primeira iteração;
2. demonstrar que

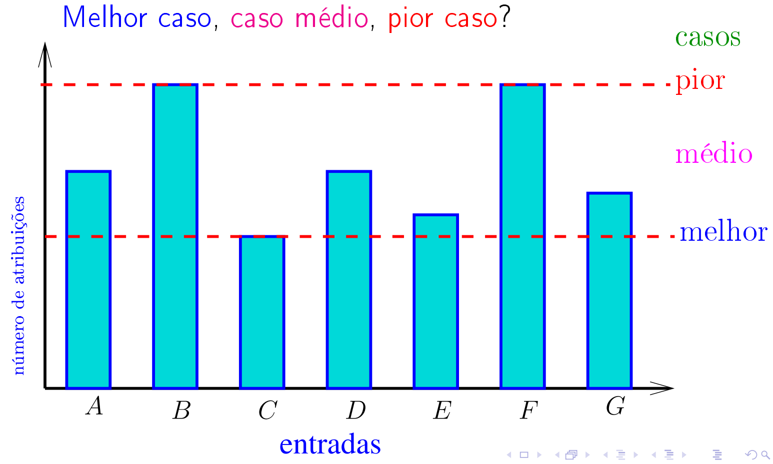
   se a relação vale no início da iteração, então ela vale no final da iteração (com os papéis de alguns atores possivelmente trocados);

3. concluir que, se relação vale no início da última iteração, então a a relação junto com a condição de parada implicam na correção do algoritmo.

Quantas atribuições faz a função?

Quantas atribuições faz a função?

Número mínimo, médio ou máximo?
Melhor caso, caso médio, pior caso?



Quantas atribuições faz a função?

```
LINHAS 2-6 (v, i, x)
2       x = v[i]
3       j = i - 1
4       while j >= 0 and v[j] > x:
5          v[j+1] = v[j]
6          j -= 1
```

Quantas atribuições faz a função?

```
LINHAS 2-6 (v, i, x)
2       x = v[i]
3       j = i - 1
4       while j >= 0 and v[j] > x:
5          v[j+1] = v[j]
6          j -= 1
```

linha	atribuições (número máximo)
2-3	?
4	?
5-6	?
total	?

Quantas atribuições faz a função?

```
LINHAS 2-6 (v, i, x)
2       x = v[i]
3       j = i - 1
4       while j >= 0 and v[j] > x:
5          v[j+1] = v[j]
6          j -= 1
```

linha	atribuições (número máximo)
2-3	$= 1 + 1$
4	$= 0$
5-6	$\leq 2 \times (1 + i)$
total	?

Quantas atribuições faz a função?

```
LINHAS 2-6 (v, i, x)
2       x = v[i]
3       j = i - 1
4       while j >= 0 and v[j] > x:
5          v[j+1] = v[j]
6          j -= 1
```

linha	atribuições (número máximo)
2-3	$= 1 + 1$
4	$= 0$
5-6	$\leq 2 \times (1 + i)$
total	$\leq 2i + 3 \leq 3n$

Quantas atribuições faz a função?

```
 def insercao (n, v):
0    n = len(v)
1    for i in range(1,n):   # /*A*/
2       LINHAS 2-6 (v, i, x)
7       v[j+1] = x
```

linha	atribuições (número máximo)
0	?
1	?
2–6	?
7	?
total	?

Quantas atribuições faz a função?

```
 def insercao (n, v):
0    n = len(v)
1    for i in range(1,n):   # /*A*/
2       LINHAS 2-6 (v, i, x)
7       v[j+1] = x
```

linha	atribuições (número máximo)
0	$= 1$
1	$= n$
2–6	$\leq (n - 1)3n$
7	$= n - 1$
total	$\leq 3n^2 - n + 1 \leq 3n^2 + 1$

Análise mais fina

linha	atribuições (número máximo)
0	?
1	?
2	?
3	?
4	?
5	?
6	?
7	?
total	?

Análise mais fina

linha	atribuições (número máximo)
0	$= 1$
1	$= n$
2	$= n - 1$
3	$= n - 1$
4	$= 0$
5	$\leq 1 + 2 + \cdots + n = n(n+1)/2$
6	$\leq 1 + 2 + \cdots + (n-1) = (n-1)n/2$
7	$= n - 1$
total	$\leq n^2 + 3n - 2$

$n^2 + 3n - 2$ versus n^2

n	$n^2 + 3n - 2$	n^2
1	2	1
2	8	4

$n^2 + 3n - 2$ versus n^2

n	$n^2 + 3n - 2$	n^2
1	2	1
2	8	4
3	16	9
10	128	100

$n^2 + 3n - 2$ versus n^2

n	$n^2 + 3n - 2$	n^2
1	2	1
2	8	4
3	16	9
10	128	100
100	10298	10000
1000	1002998	1000000

$n^2 + 3n - 2$ versus n^2

n	$n^2 + 3n - 2$	n^2
1	2	1
2	8	4
3	16	9
10	128	100
100	10298	10000
1000	1002998	1000000
10000	100029998	100000000
100000	10000299998	10000000000

n^2 domina os outros termos

Consumo de tempo

Se a execução de cada linha de código consome 1 unidade de tempo, qual o consumo total?

```
 def insercao (n, v):
0    n = len(v)
1    for i in range(1,n):  # /*A*/
2       x = v[i]
3       j = j - 1
4       while j >= 0 and v[j] > x:
5          v[j+1] = v[j]
6          j -= 1
7       v[j+1] = x
```

Consumo de tempo no pior caso

linha	todas as execuções da linha
0	$= 1$
1	$= n$
2	$= n-1$
3	$= n-1$
4	$\leq 2+3+\cdots+n = (n-1)(n+2)/2$
5	$\leq 1+2+\cdots+(n-1) = n(n-1)/2$
6	$\leq 1+2+\cdots+(n-1) = n(n-1)/2$
7	$= n-1$
total	$\leq (3/2)n^2+(7/2)n-5 = \mathrm{O}(n^2)$

Consumo de tempo no melhor caso

linha	todas as execuções da linha
0	$= 1$
1	$= n$
2	$= n-1$
3	$= n-1$
3	$= n-1$
5	$= 0$
6	$= 0$
7	$= n-1$
total	$= 5n-3 = \mathrm{O}(n)$

Pior e melhor casos

O maior consumo de tempo da função `insercao` ocorre quando a lista `v[0:n]` dada é decrescente. Este é o pior caso para a função `insercao`.

O menor consumo de tempo da função `insercao` ocorre quando a lista `v[0:n]` dada é já é crescente. Este é o melhor caso para a função `insercao`.

Conclusões

O consumo de tempo da função `insercao` no pior caso é proporcional a n^2.

O consumo de tempo da função `insercao` melhor caso é proporcional a n.

O consumo de tempo da função `insercao` é $\mathrm{O}(n^2)$.

Resultados experimentais

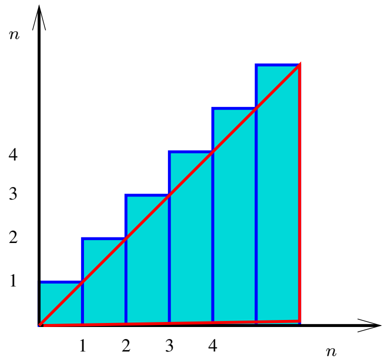
insercao		
n	tempo (s)	comentário
256	0.00	
512	0.02	
1024	0.07	
2048	0.22	
4096	1.02	
8192	3.88	
16384	16.23	
32768	66.23	> 1 min
65536	275.75	> 4 min
131072	1158.20	> 19 min
262144	5119.28	> 85 min

Enquanto isso... em outro computador...

insercao		
n	tempo (s)	comentário
256	0.00	
512	0.01	
1024	0.04	
2048	0.17	
4096	0.66	
8192	2.66	
16384	10.70	
32768	42.98	$>$ 0.5 min
65536	170.25	$\approx$ 2.8 min
131072	687.99	$\approx$ 11.45 min

$1 + 2 + \cdots + (n - 1) + n = ?$

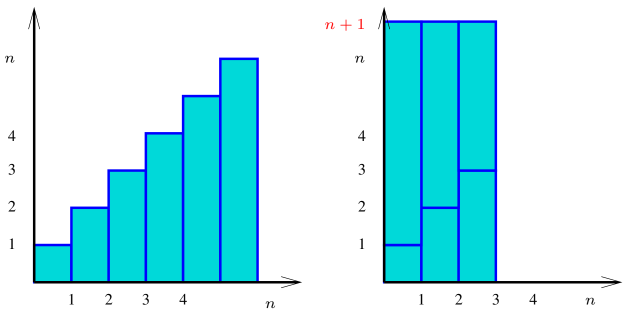
Carl Friedrich Gauss, 1787



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

$1 + 2 + \cdots + (n - 1) + n = ?$

Carl Friedrich Gauss, 1787



$$(n+1) \times \frac{n}{2} = \frac{n(n+1)}{2}$$