Nos anos 70, J.Martin sugeriu que 40% do tempo de TODOS os Computadores era gasto fazendo ordenações.

Na França, os computadores são conhecidos como *ordinateurs*

O conceito de ordem é fundamental na informática

Excelente ferramenta para estudar algoritmos e complexidades

Pré-requisito: haver a relação de ordem no conjunto (maior, menor,...)

Ordenar um conjunto V é rearranjar seus elementos de maneira a que para quaisquer i,j com i<j, tenhamos V[i]>V[j] (estritamente decrescente), V[i] >= V[j] (decrescente), V[i]<V[j] (estritamente crescente) e V[i]<=V[j] (crescente)

Tipicamente, o elemento V[i] é composto por inúmeras informações. A que determina a ordenação, é chamada KEY e representada por k (onde k pertence a V[i]).

Os algoritmos tratam apenas da chave k, subentendendo-se a reordenação do elemento V[i] por inteiro.

Se V[i] é muito grande, é comum criar um array auxiliar composto de k[i] e de P[i], sendo que este último é um apontador para V[i].

Existem 2 classes de algoritmos: INTERNOS (o array inteiro cabe na memória) e EXTERNOS (não cabe tudo).

Algoritmos podem ser ESTÁVEIS (Se o elemento A está antes de B na entrada, e se k(A) = k(B), então A estará antes de B na saída) e NÃO ESTÁVEIS (não dá Para dizer nada sobre A e B na saída).
Critério importante: fazer trocas "in situ", buscando preservar espaço.

Existem 2 grupos de algoritmos internos:

Tem complexidade $O(n^2)$	**na verdade $K1.O(n^2)$**
Tem complexidade O(n.log n)	**K2.O(n.log n)**
Lembrar que K2 > K1	

Logo para n pequeno (100 ?), é melhor usar o primeiro grupo

Algoritmos Exponenciais:

Bolha: compara vizinhos. Se fora de ordem, troca-os de lugar.

Bolha estrela: se nenhuma troca é feita, encerra sem ir até o fim

Shaker: bolha com duas direções (demora menos)

Inserção: 2 montes: Tira 1 elemento do primeiro monte (desordenado) e

coloca-o no segundo monte (ordenado)

Seleção: acha o menor e troca-o de lugar com o primeiro. ... Até o fim

Shell: compara vizinhos distantes. Se fora de ordem, troca-os

Algoritmos Logaritmicos:

Heap: cria um heap, retira do topo, e joga no final do monte...

Quick-sort: recursivo por subdivisões disjuntas sucessivas

Tabela de Desempenhos Comparativos medidos em segundos
Vetor Aleatório (valores diferentes randômicos)

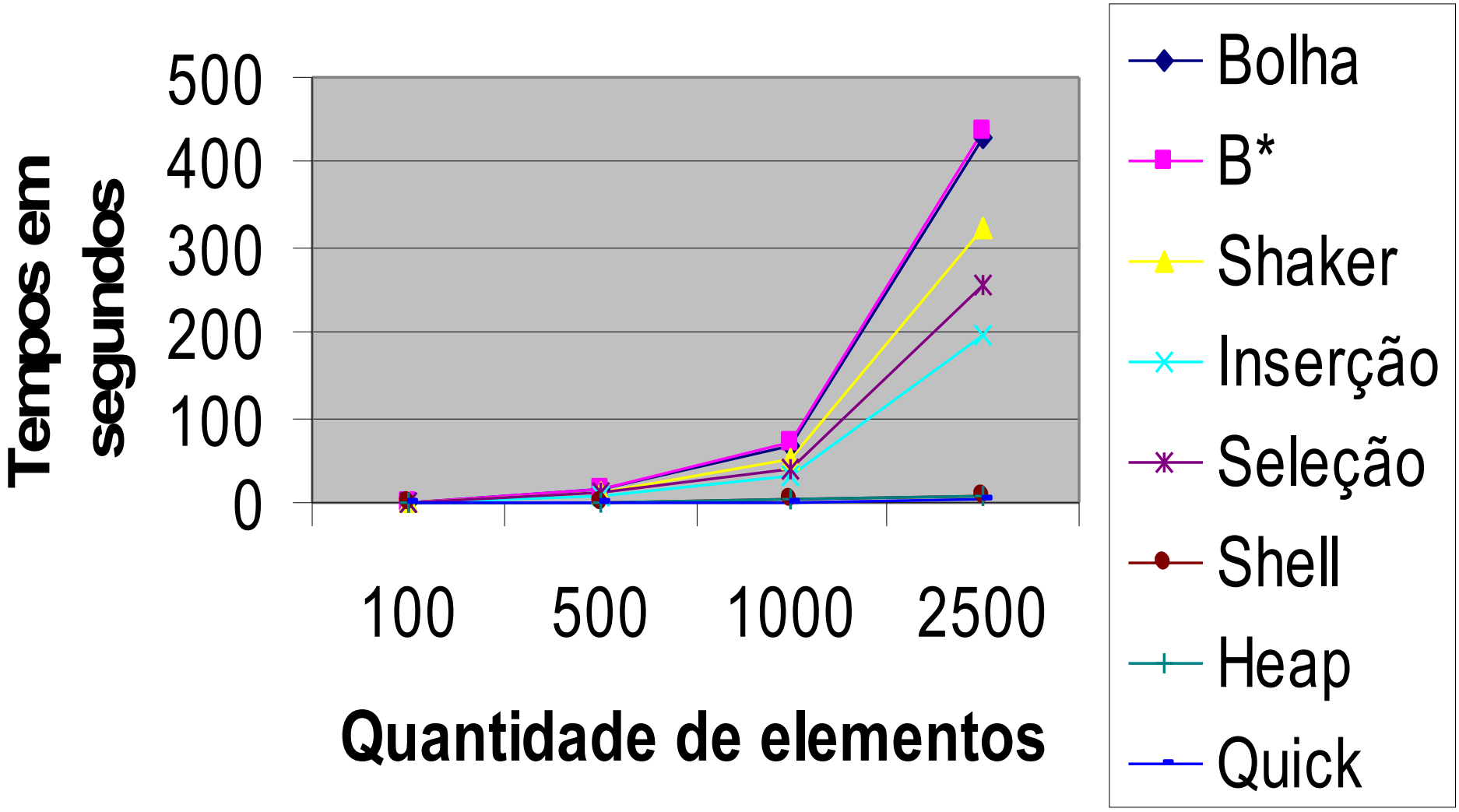
Tam	Bol	B*	Sha	Ins	Sel	Shel	Heap	Quick
100	0.6	0.7	0.4	0.3	0.4	0.1	0.1	0.06
500	16	17	12	7	10	1	1	0.5
1000	68	70	51	31	41	2	2	1
2500	429	436	324	198	255	8	6	3

Vetor Quasi-Ordenado (vetor já ordenado menos o do meio)

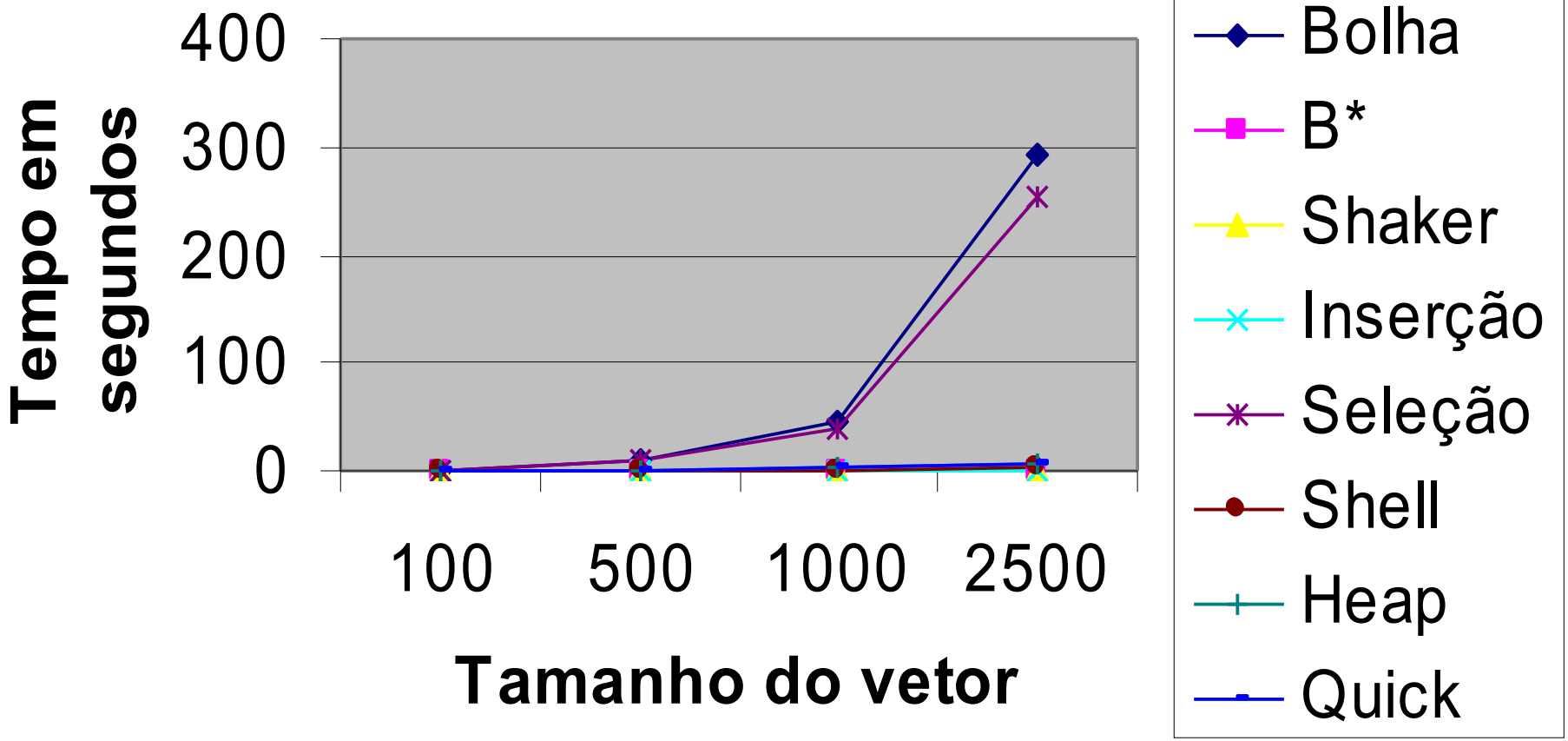
Tam	Bol	B*	Sha	Ins	Sel	Shel	Heap	Quick
100	0.4	0.06	0.05	0.01	0.4	0.1	0.1	0.06
500	11	0.1	0.1	0.1	10	0.5	1	0.3
1000	47	0.3	0.2	0.2	40	1	2	0.8
2500	294	0.6	0.6	0.5	255	3	6	2

Medidas feitas em um Pentium 500 com WME usando APL+ 6

Tempo para ordenar um vetor



Ordenação de Vetor quase ordenado



Heurísticas de USO:

1. Se você só conhece um algoritmo, é esse o melhor !

2. Para n menor que 100, escolha qualquer um

3. Para n até 2000, com uso one-way: escolha qualquer um

4. Para n grande, com múltiplos usos:

Se o vetor está desordenado ou é desconhecido:
Usar: QUICK ou HEAP. Em último caso, usar o
SHELL.

Se o vetor está quase-ordenado (poucos elementos estão fora de ordem):
Uma boa pedida é o INSERÇÃO

Obs: Em certos casos (worst case), o QUICK tem desempenho $O(n^2)$. Se Este comportamento for detectado, uma solução é "embaralhar" os dados Antes de começar.

Se tiver que levar 1 algoritmo de ordenação para uma ilha deserta, eu Sugiro levar o SHELL SORT.

```
Algoritmo BOLHA
 inteiro I, J
 I  <-  TAM(V)
 enquanto (I >= 2)
    J <- 2
    enquanto (J <= I)
       se (V[J-1] > V[J])
          TROCA (J, J-1)
       fim {se}
       J++
    fim {enquanto}
    I--
 fim {enquanto}
fim {algoritmo}
```

```
Algoritmo BOLHAEST
   inteiro I, J
   lógico MUDOU  <-  .V.
   I  <-  TAM(V)
   enquanto (I>=2) E MUDOU
      J  <-  2
      MUDOU  <-  .F.
      enquanto (J <= I)
         se (V[J-1] > V[J])
            TROCA (J, J-1)
            MUDOU <- .V.
         fim {se}
         J++
      fim {enquanto}
      I--
   fim {enquanto}
fim {algoritmo}
```

Os: TROCA(i,j) é troca V[i] com V[j]

```
Algoritmo INSERÇÃO
 inteiro I, J, X
 I  <-  2
 enquanto (I <= TAM)
    X  <-  V[I]
    J  <-  I - 1
    enquanto ((J >= 1)
       .E. (V[J] > X))
       V[J+1]  <-  V[J]
       J--
    fim {enquanto}
    V[J+1]  <-  X
    I++
 fim {enquanto}
fim {algoritmo}
```

```
Algoritmo SELEÇÃO
 inteiro I,J,CO,IN
 I  <-  1
 enquanto (I <= TAM)
    CO  <-  V[I]
    IN  <-  I
    J  <-  I + 1
    enquanto (J <= TAM)
       se V[J] < CO
          CO   <-  V[J]
          IN    <-  J
       fim {se}
       J++
    fim {enquanto}
    TROCA (I, IN)
    I++
 fim {enquanto}
fim {algoritmo}
```

```
Algoritmo SHELL
 inteiro K, H, I, X, J
 DECR  <-  seq {4 1}
 para K de 1 ate tam(DECR)
    H  <-  DECR[K]
    I  <-  H
    enquanto (I < TAM)
       X  <-  V[I+1]
       J  <-  I
       enquanto ((J >= H  .E.
          V[1+J-H] > X))

          V[1+J]  <-  V[1+J-H]

          J  <-  J-H

       fim {enquanto}
       V [1+J]  <-  X
       I++
    fim {enquanto}
 fim {para}
fim {algoritmo}
```

Exemplo de uso (chinês)

```
i  1  2  3  4  5  6  7  8  9 10
-  -- -- -- -- -- -- -- -- -- --
4 (45)15 79 72 (61) 9 58 29 44 94
5 45  (9)79 72 61 (15)58 29 44 94
6 45  9 (58)72 61 15 (79)29 44 94
7 45  9 58 (29)61 15 79 (72)44 94
8 44  9 58 29 (45)15 79 72 (61)94
9 44  9 58 29 45 [15]79 72 61 [94
1 (9)44 58 29 45 15 79 72 61 94
2  9 44 58 29 45 15 79 72 61 94
3  9 (29)44 58 45 15 79 72 61 94
```

Seqüências Conhecidas:
Original do Dr Shell: h_t=chão(N/2), h_k=chão(h_k+1/2)... h_1=1 (ruim)
Exemplo: para N=30 => 15, 7, 3, 1, mas para N=32 => 16, 8, 4, 2, 1

Hibbard = $h_t=(2^k)-1$, ..., 7, 3, 1
Exemplo: para N=30 =>1, 3, 7, 15, 31, para qualquer N

Knuth = $h_i = 1/2\ [(3^i)+1]$
Exemplo: 1, 5, 14, 41...

Gonnet's h_t=chão(N/2.2), h_k=chão(h_k+1/2.2), com h1=1, h2=2
Exemplo: N=30 => 30/2.2=13.6, chao=13, 13/2.2=5.9, chão=5, 5/2.2=2.27 chão=2, h_1=1

Sedgewick: 1, 5, 19, 41, 109, fórmula completa em [Data Structures And Algorithm Analysis in C. Mark Allen Weiss]

Todas estas seqüências geram complexidades $O(n^{5/4})$ ou $O(n^{7/6})$ ou $O(n^{3/2})$ ou $O(n^{4/3})$...

```
Supondo n=100.000, elevado a 7/6 =        681.292      (7/6=1,16)
                   elevado a 5/4 =      1.778.279      (5/4=1,25)
                   elevado a 4/3 =      4.641.588      (4/3=1,33)
                   elevado a 3/2 =     31.622.776      (3/2=1,50)
                   elevado a 2   = 10.000.000.000
```

As regras para a seqüência são:

1. Qualquer número de elementos
2. O último número tem que ser 1
3. cada número tem que ser maior do que o anterior.
Preferencialmente, que estes números sejam primos entre si (melhora o desempenho)
4. Não tem sentido usar número maior que o tamanho do conjunto

Note que a seqüência simples {1} é aceitável (atende às 3 regras acima), mas olhando o algoritmo percebe-se que o desempenho é pífio, pois o algoritmo virá um quase INSERÇÃO DIRETA

Eis algumas comparações em um vetor de 300 aleatórios:

seq={1}	**:23.104 comparações**
seq={1,5}	**:6.269**
seq={1,5,19}	**:3.548**
seq={1,5,19,41}	**:3.272**
seq={1,5,19,41,109}	**:3.131**

```
Algoritmo HEAPSORT
   inteiro I, J
   I  <-  chão [TAM(V)/2]
   para J de I até 1 p -1
      DESCE J,TAM
   fim {para}
   para J de TAM ate 1 p -1
      V[J]  <-  EXCLUI(J)
   fim {para}
fim {algoritmo}
```

```
função DESCE (int K, LEN)
   inteiro J
   J  <-  K * 2
   se (J <= LEN)
      se (J < LEN)
         se (V[J+1] > V[J])
            J++
         fim {se}
      fim {se}
      se V[K] < V[J]
         TROCA (K, J)
         DESCE (J, LEN)
      fim {se}
   fim {se}
fim {função}
```

```
int função EXCLUI (int J)
   devolva V[1]
   V[1]  <-  V[J]
   DESCE (1, j)
fim {função}
```

```
Algoritmo QUICKSORT
  função RAP(inteiro I, F)
     inteiro ANT, DEP, MEI, AUX
     ANT <- I
     DEP <- F
     MEI  <-  V[(I+F) div 2]
     repita
        enquanto (V[ANT] < MEI)
           ANT++
        fim {enquanto}
        enquanto (V[DEP] > MEI)
           DEP- -
        fim {enquanto}
        se (ANT <= DEP)
             TROCA (ANT,DEP)
             ANT++
             DEP--
          fim {se}
        ate (ANT > DEP)
        se (I < DEP)
           RAP (I, DEP)
        fim {se}
        se (ANT < F)
           RAP (ANT, F)
        fim {se}
     fim {função}
     RAP [1,tamanho(V)]
     devolva (V)
fim {algoritmo}
```

Ordenação Externa
Premissa básica: não é possível ter todos dados na memória ao mesmo Tempo. Em um instante: só um pequeno conjunto de dados
Visão alternativa: construir um índice e percorrer seqüencialmente

Exemplos: 3.000.000 veículos DETRAN ou 9.500.000 RGs ou 110.000.000 boletos cartão crédito (ao mês !)

Algoritmo Básico:
Supondo k arquivos em paralelo na entrada
Que o processador consegue ordenar B registros
Que o conjunto total tem N registros

Os dados a ordenar estão em S1. Que é lido de B em B registros. Cada bloco B É ordenado e os blocos ordenados são gravados em S3 e S4 alternadamente.

Ao final S3 e S4 são intercalados, gravando-se em S1 e S2 Alternadamente. Agora o tamanho do bloco é 2B.

Assim sucessivamente até ter-se o conjunto todo ordenado em um único S.

Neste caso k=2 (two-way-merge)

Exemplo: N=13, k=2, B=3
S1=44, 27, 78, 12, 81, 17, 23, 90, 46, 73, 29, 15, 20
S2=vazia
S3=27, 44, 78 | 23, 46, 90 | 20
S4=12, 17, 81 | 15, 29, 73
Intercalando S3 e S4 e guardando em S1 e S2
S1=12, 17, 27, 44, 78, 81 | 20
S2=15, 23, 29, 46, 73, 90

Intercalando S1 e S2 e guardando em S3, S4
S3=12, 15, 17, 23, 27, 29, 44, 46, 73, 78, 81, 90
S4=20

Juntando tudo
S1= 12, 15, 17, 20, 23, 27, 29, 44, 46, 73, 78, 81, 90

Se mais do que 2 entradas podem ser usadas, o processo fica mais curto.
Na hora da intercalação usa-se um heap para a escolha do menor.
A complexidade associada é O(teto($\log_k$ (N/B))).

No exemplo: teto($\log_2$(13/3)) = teto($\log_2$ 4,3) = teto(2.1) = 3 passadas

Outro: Quantas passadas são necessárias para classificar 10.500.000 recs
Com 4 entradas e bloco de 2000 recs ?

R= teto($\log_4$ (10.500.000/2000)) = teto($\log_4$ 5250) = teto 6.17 = 7