Estrutura de Dados I

Capítulo 7: Ordenação e Análise de Algoritmos

Prof. Abrantes Araújo Silva Filho

Introdução

- Quando começamos a estudar recursão, vimos 2 algoritmos recursivos para Fibonacci, com grande diferença de performance:

```c
int fib_rec (int n)
{
    if (n < 0)
        return -1;
    else if (n < 2)
        return n;
    else
        return fib_rec(n - 1) + fib_rec(n - 2);
}
```

```c
int fib (int n)
{
    if (n < 0) return -1;
    return seq_adit(n, 0, 1);
}
```

```c
int seq_adit (int n, int t0, int t1)
{
    if (n < 0)  return -1;
    if (n == 0) return t0;
    if (n == 1) return t1;
    return seq_adit(n - 1, t1, t0 + t1);
}
```

Introdução

- De fato, a solução recursiva simples tem um “custo” absurdo de tempo, o que por vezes causa uma má reputação da recursividade:

```
Comparação dos algoritmos de Fibonacci:
 N     ITER            REC-S
----------------------------------------
 0     0.000000410       0.000000309
 5     0.000000526       0.000000860
10     0.000000575       0.000003189
15     0.000000569       0.000023415
20     0.000000454       0.000247772
25     0.000000510       0.002760314
30     0.000000572       0.005826769
35     0.000000215       0.062607979
40     0.000000170       0.681828316
45     0.000000198       7.507966543
50     0.000000195      83.073958209
55     0.000000218     941.356740064
```

Introdução

- Mas se usarmos uma implementação eficiente veremos que o problema não é realmente a recursividade:

```
Comparação dos algoritmos de Fibonacci:
 N     ITER            REC-S             REC-A
-------------------------------------------------
 0     0.000000410       0.000000309     0.000000354
 5     0.000000526       0.000000860     0.000000452
10     0.000000575       0.000003189     0.000000401
15     0.000000569       0.000023415     0.000000441
20     0.000000454       0.000247772     0.000000678
25     0.000000510       0.002760314     0.000000989
30     0.000000572       0.005826769     0.000000312
35     0.000000215       0.062607979     0.000000303
40     0.000000170       0.681828316     0.000000352
45     0.000000198       7.507966543     0.000000406
50     0.000000195      83.073958209     0.000000457
55     0.000000218     941.356740064     0.000000500
```

Introdução

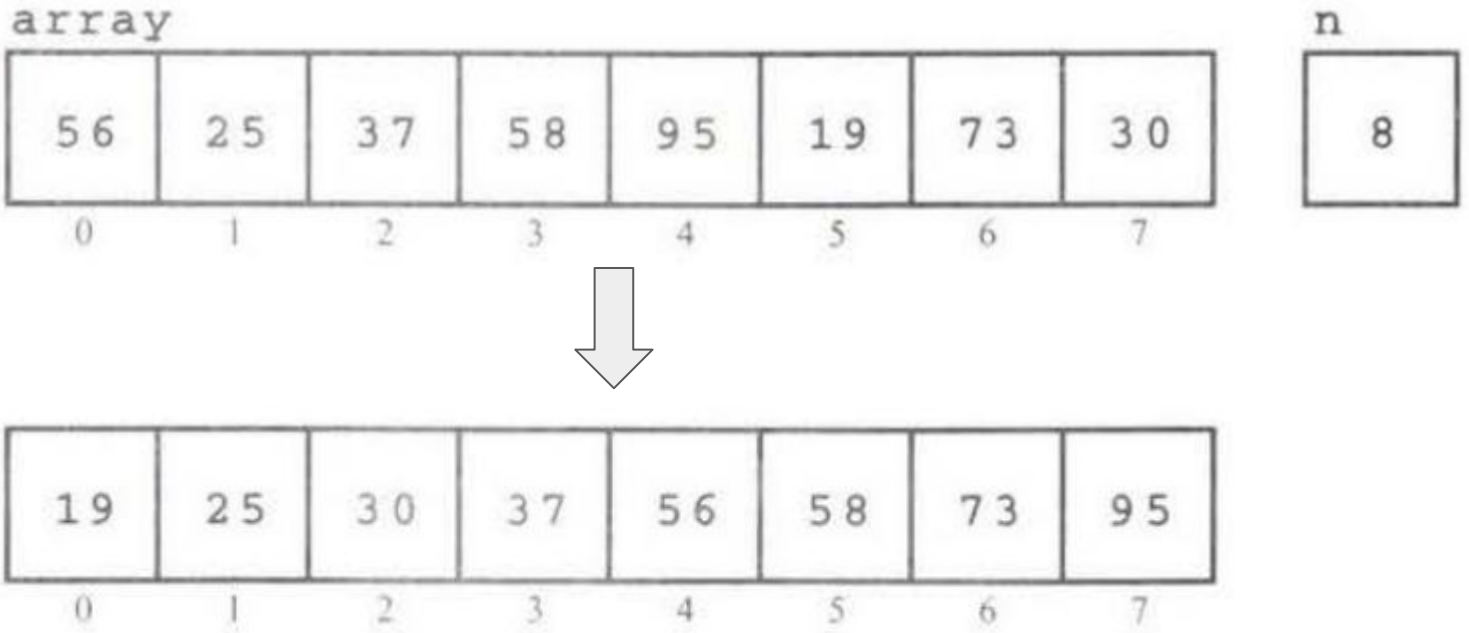
- Na realidade, a habilidade de pensar recursivamente sobre um problema, em muitos casos, nos leva a estratégias muito mais eficientes do que a abordagem iterativa. Veremos claramente nos **algoritmos de ordenação**!

- Mas, antes, é necessário definir melhor o que é essa **eficiência** dos algoritmos:
  - O que significa essa eficiência?
  - Como medir a eficiência?

- Campo específico: **análise de algoritmos**.
  - Vamos entender a análise de algoritmos estudando o **problema da ordenação**.

O problema da ordenação

- De modo informal, o problema da ordenação consiste em reordenar os elementos de um array para que eles fiquem em uma ordem definida:



O problema da ordenação

- De modo mais formal, o problema da ordenação é o seguinte:
  - **Input**: uma seqüência de **n** chaves: **$[a_1, a_2, \ldots, a_n]$**
  - **Output**: uma permutação (reordenação) da seqüência de entrada, **$[a'_1, a'_2, \ldots, a'_n]$**, de tal forma que **$a'_1 \leq a'_2 \leq \ldots \leq a'_n$**.

- Cada seqüência de entrada é uma **instância** do problema de ordenação, que satisfaça qualquer restrição imposta pelo problema.

- O elemento que está sendo usado para a ordenação é chamado de **chave** (**key**) de ordenação. Frequentemente há outros dados associados, chamados de dados **satélites** (**satellite data**).

- Chave + satélite = **registro**.

O problema da ordenação

- Esse problema é tão importante que existem diversos algoritmos para ordenação (mais de 40):
  - Selection sort
  - Insertion sort
  - Bubble sort
  - Cocktail sort
  - Bucket sort
  - Counting sort
  - Merge sort
  - Quick sort
  - Shell sort
  - Heap sort
  - Radix sort
  - …

O problema da ordenação

- Em geral dizemos que estamos ordenando “arquivos” de registros que contém chaves.

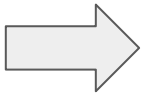
- Se o “arquivo” cabe na memória, então podemos fazer toda a ordenação na memória (**ordenação interna**)

- Se o “arquivo” não cabe na memória, então teremos que fazer parte da ordenação em memória e usar o disco (**ordenação externa**)

O problema da ordenação

- Um método de ordenação pode ser **estável** ou instável (**não estável**):
  - **Estável**: **preserva a ordem relativa de chaves iguais** no arquivo, por exemplo: se uma lista alfabética de estudantes é ordenada por notas, um método estável produz uma lista onde os estudantes com a mesma nota ainda estão ordenados alfabeticamente:

Nome	Nota
Abrantes	9
Amora	10
Carlos	8
Elis	10
Flávia	9
Helena	10
Maria Clara	7
Maria Luiza	7
Mariana	8
Tomás	8
Tomázia	7
Vicente	9

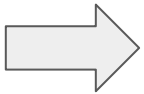


Nome	Nota
Maria Clara	7
Maria Luiza	7
Tomázia	7
Carlos	8
Mariana	8
Tomás	8
Abrantes	9
Flávia	9
Vicente	9
Amora	10
Elis	10
Helena	10

O problema da ordenação

- Um método de ordenação pode ser **estável** ou instável (**não estável**):
  - **Não Estável**: **não preserva a ordem relativa de chaves iguais** no arquivo, por exemplo: se uma lista alfabética de estudantes é ordenada por notas, um método não estável produz uma lista onde os estudantes com a mesma nota não estão ordenados alfabeticamente:

Nome	Nota
Abrantes	9
Amora	10
Carlos	8
Elis	10
Flávia	9
Helena	10
Maria Clara	7
Maria Luiza	7
Mariana	8
Tomás	8
Tomázia	7
Vicente	9



Nome	Nota
Tomázia	7
Maria Clara	7
Maria Luiza	7
Mariana	8
Carlos	8
Tomás	8
Vicente	9
Abrantes	9
Flávia	9
Helena	10
Amora	10
Elis	10

O problema da ordenação

- Um método de ordenação pode ser **estável** ou instável (**não estável**):
  - A maioria dos métodos simples de ordenação são estáveis
  - A maioria dos métodos sofisticados de ordenação não são estáveis
  - Se a estabilidade for um requisito, pode ser forçada por algum artifício:
    - anexar um índice para cada chave
    - trocar a chave
    - etc.
  - Nunca confie cegamente que um método é estável se você não souber

- Atenção: nos exemplos anteriores a lista de estudantes estava ordenada alfabeticamente, mas isso não é sempre necessário! **O que é necessário é que o método preserve, ou não, a ordem relativa de chaves iguais no arquivo**.

O problema da ordenação: como vamos estudar?

- Para o foco ser apenas o algoritmo, trabalharemos com a ordenação de arrays de inteiros.

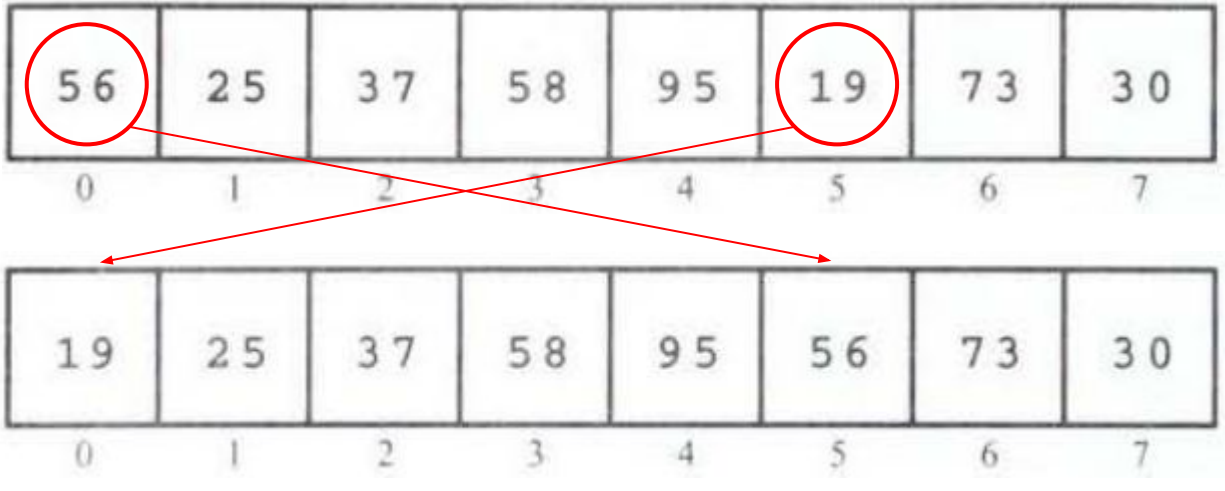
- Se o registro a ser ordenado é grande, deve-se evitar ficar movendo os registros de lugar e fazer uma ordenação “indireta”:
  - Os registros por si mesmos não são ordenados mas, sim, um array de ponteiros (ou de índices) para esses registros é que é ordenado
  - Depois da ordenação dos ponteiros/índices, se necessário, os registros podem ser ordenados

- Não incluiremos muito código de checagem de erro, para simplificar o algoritmo.

- a[0] e a[n+1] podem ser utilizados para manter chaves especiais em alguns algoritmos

Algoritmos simples de ordenação: Selection Sort

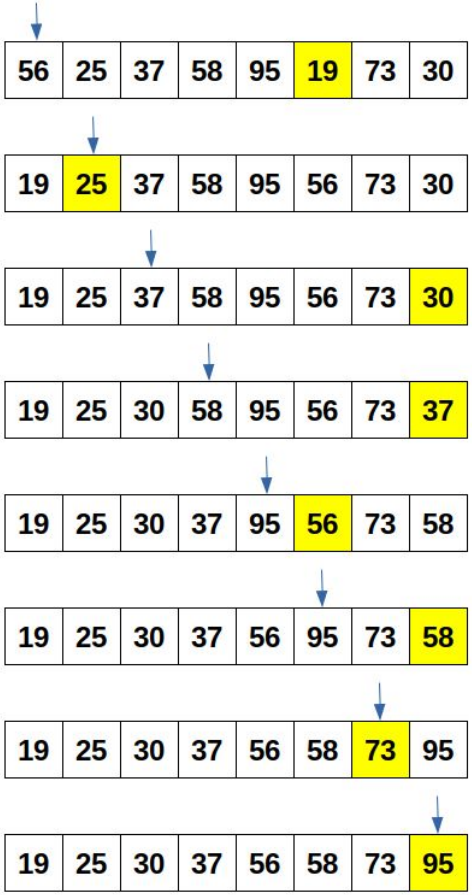
- O selection sort é um dos algoritmos de ordenação mais simples. Dado um array de N elementos ele procura pelo menor e troca esse valor com o valor que está na 1ª posição do array. Depois ele busca o segundo menor valor e troca com o elemento que está na 2ª posição do array. E assim por diante.



Algoritmos simples de ordenação: Selection Sort

```
void selection_sort (int array[], int tam)
{
    int menor;

    for (int i = 0; i < tam; i++)
    {
        menor = i;
        for (int j = i + 1; j < tam; j++)
            if (array[j] < array[menor])
                menor = j;
        trocar(&array[i], &array[menor]);
    }
}
```

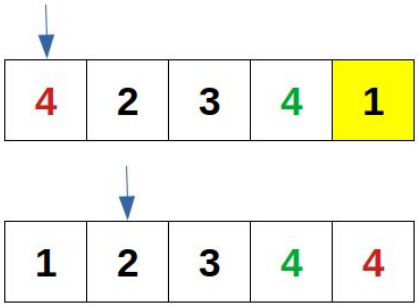
Algoritmos simples de ordenação: Selection Sort

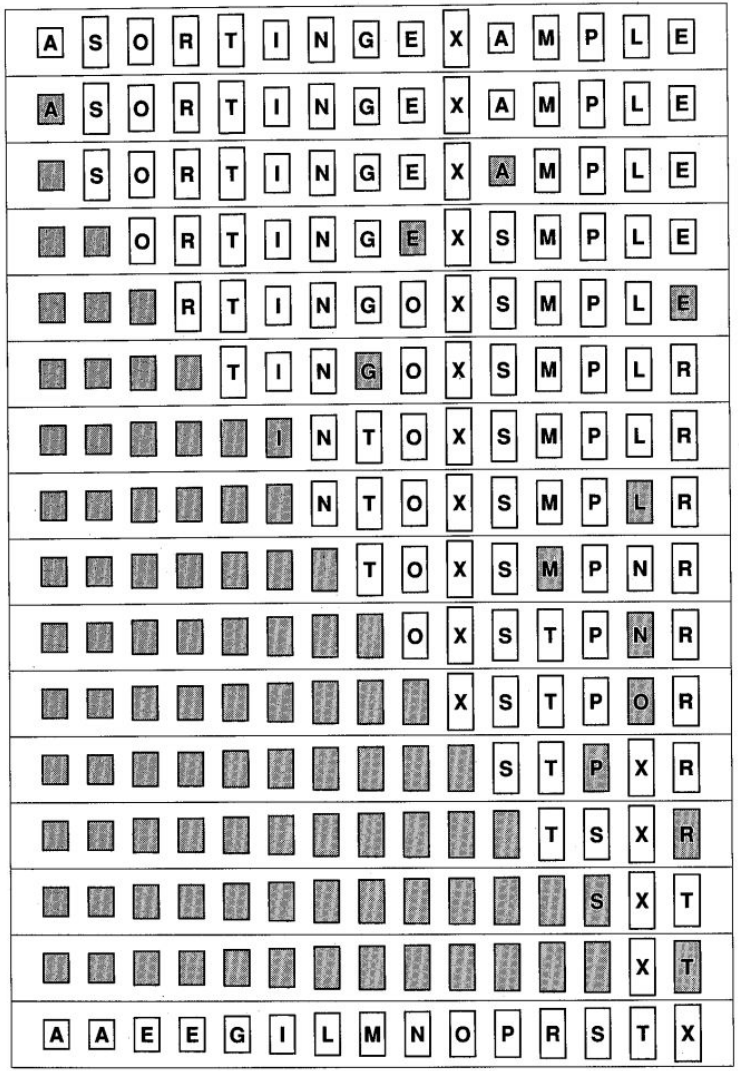


Algoritmos simples de ordenação: Selection Sort

- Como cada elemento é movido apenas 1 única vez, ele pode ser o método de escolha para ordenar arquivos com registros muito grandes e chaves pequenas.

- Selection sort NÃO É ESTÁVEL:





Algoritmos simples de ordenação: Insertion Sort

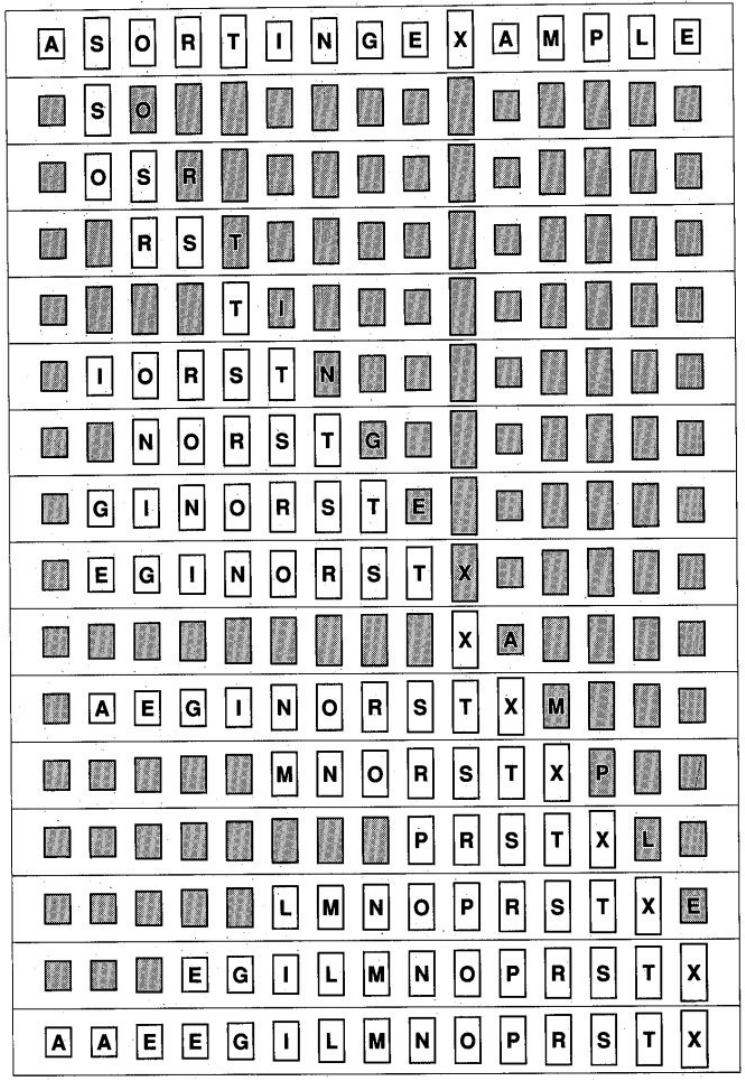
- O insertion sort também é um dos algoritmos de ordenação mais simples. Funciona de modo semelhante a como você ordenaria cartas de um jogo de baralho em sua mão:
  - Pegue a primeira carta e segure com a mão esquerda
  - Pegue a próxima carta com a mão direita e insira na posição correta na mão da esquerda. Você acha a posição correta comparando a carta da mão direita com as cartas da mão esquerda, começando na direita e indo para a esquerda. No momento em que você vê uma carta na mão da esquerda que é menor do que ou igual à carta da mão direita, insira a carta de sua mão direita imediatamente à direita da carta da mão esquerda. Se todas as carta da mão esquerda são maiores do que carta da mão direita, coloque a carta da mão direita na posição mais à esquerda na mão esquerda.
  - As cartas da mão esquerda estão sempre ordenadas

Algoritmos simples de ordenação: Insertion Sort

```
void insertion_sort (int array[], int tam)
{
    for (int i = 1; i < tam; i++)
    {
        int chave = array[i];
        int j = i - 1;
        while (j >= 0 && array[j] > chave)
        {
            array[j + 1] = array[j];
            j = j - 1;
        }
        array[j + 1] = chave;
    }
}
```

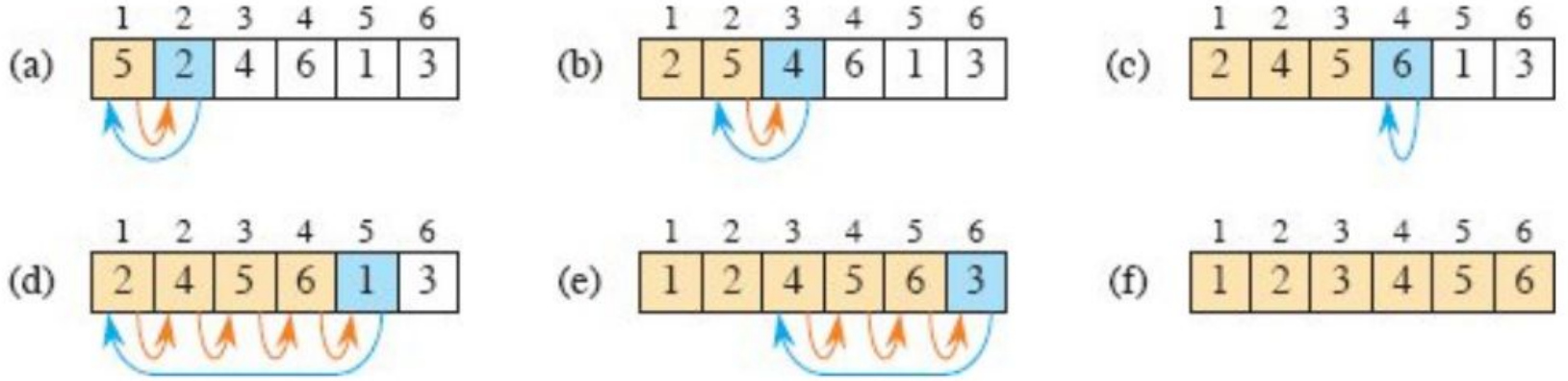
Algoritmos simples de ordenação: Insertion Sort

- Move os elementos várias vezes.
- Insertion sort É ESTÁVEL. Preserva a ordem relativa de elementos iguais porque os elementos são inseridos no lugar correto sem que a posição dos iguais anteriores seja alterada.



Algoritmos simples de ordenação: Insertion Sort

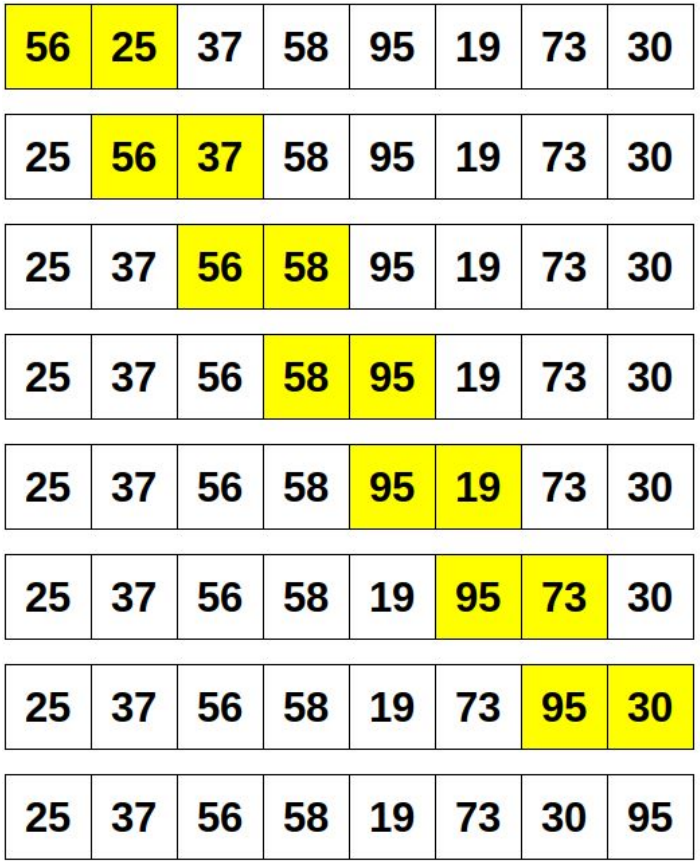
- Subarray esquerdo A[0 : i-1] está sempre ordenado (mesmo que trivialmente)
- Subarray direito A[i+1 : n-1] está não ordenado
- A chave A[i] é o elemento atual (a “carta” a ser ordenada)



Algoritmos simples de ordenação: Bubble Sort

- O bubble sort também é um dos algoritmos de ordenação mais simples. Dado um array de N elementos ele procura pelo maior elemento e “flutua” esse elemento até a maior posição. Depois ele busca o segundo maior elemento e “flutua” esse elemento até a segunda maior posição, e assim por diante.

- É um método ESTÁVEL pois a troca é feita entre elementos adjacentes, garantindo que os outros elementos mantenham sua ordem original.



Algoritmos simples de ordenação: Bubble Sort

```
void bubble_sort (int array[], int tam)
{
    for (int i = tam - 1; i >= 0; i--)
        for (int j = 1; j <= i; j++)
            if (array[j - 1] > array[j])
                trocar(&array[j - 1], &array[j]);
}
```

Algoritmos simples de ordenação: comparação

- Em: n/4, n/2 e 3n/4

- Selection sort:
  - Vai da esquerda para a direita, colocando os elementos na posição, sem “olhar para trás”.
  - Passa a maior parte do tempo tentando encontrar o menor elemento na parte não ordenada do array
  - O subarray à esquerda está ordenado e não muda após cada ordenação

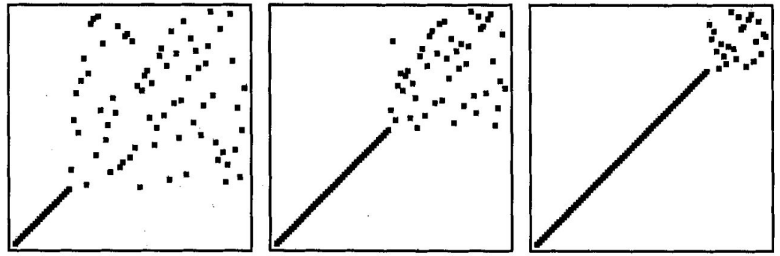


Figure 8.3 Selection sorting a random permutation.

Figure 8.4 Insertion sorting a random permutation.

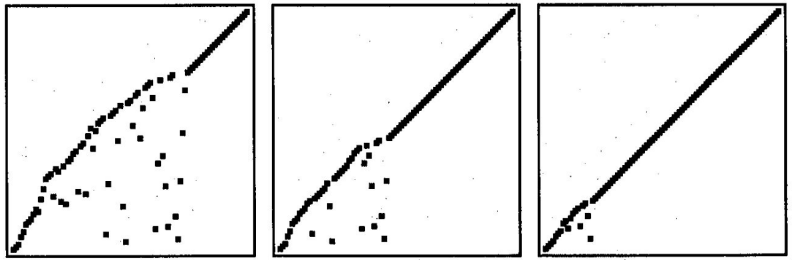


Figure 8.5 Bubble sorting a random permutation.

Algoritmos simples de ordenação: comparação

- Em: n/4, n/2 e 3n/4

- Insertion sort:
  - Vai da esquerda para a direita, colocando os elementos na posição, sem “olhar para frente”.
  - Passa a maior parte do tempo tentando ordenando os elementos
  - O subarray à esquerda está continuamente sendo alterado

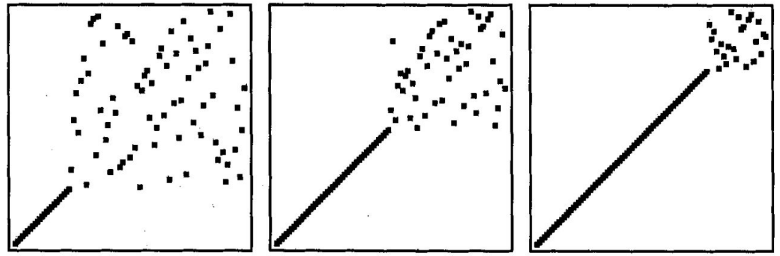


Figure 8.3 Selection sorting a random permutation.

Figure 8.4 Insertion sorting a random permutation.

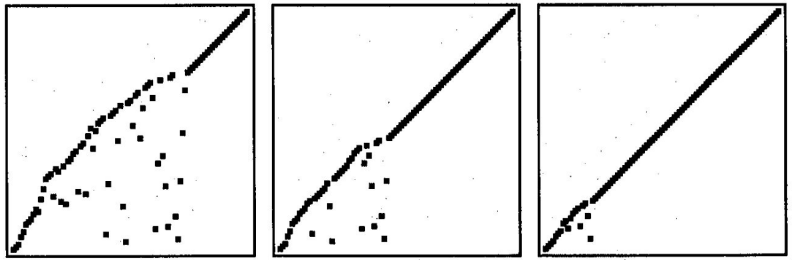


Figure 8.5 Bubble sorting a random permutation.

Algoritmos simples de ordenação: comparação

- Em: n/4, n/2 e 3n/4

- Bubble sort:
  - É, de certo modo, semelhante ao selection sort, selecionando o maior elemento e colocando na posição.
  - Vai da direita para a esquerda, colocando os elementos na posição.
  - Gasta tempo fazendo “ordenações parciais” dentro da parte não ordenada do array.

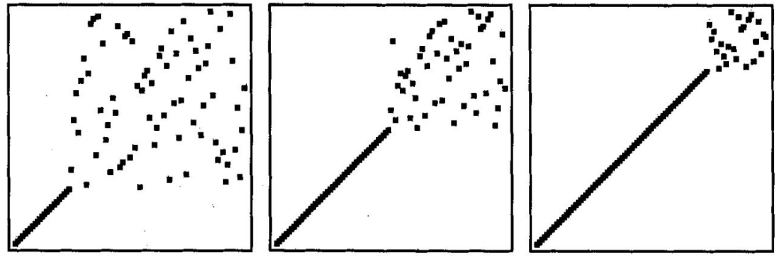


Figure 8.3 Selection sorting a random permutation.

Figure 8.4 Insertion sorting a random permutation.

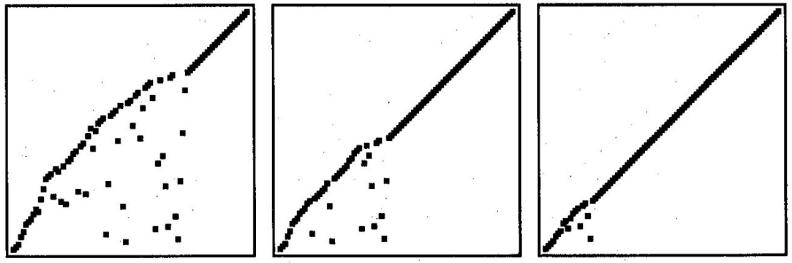


Figure 8.5 Bubble sorting a random permutation.

Algoritmos simples de ordenação: otimizações

- Os métodos de ordenação podem ser implementados de diversas maneiras. Os algoritmos mostrados aqui estão entre os mais simples possíveis, mas podem ser otimizados um pouco com implementações melhores. Ex.: bubble sort:

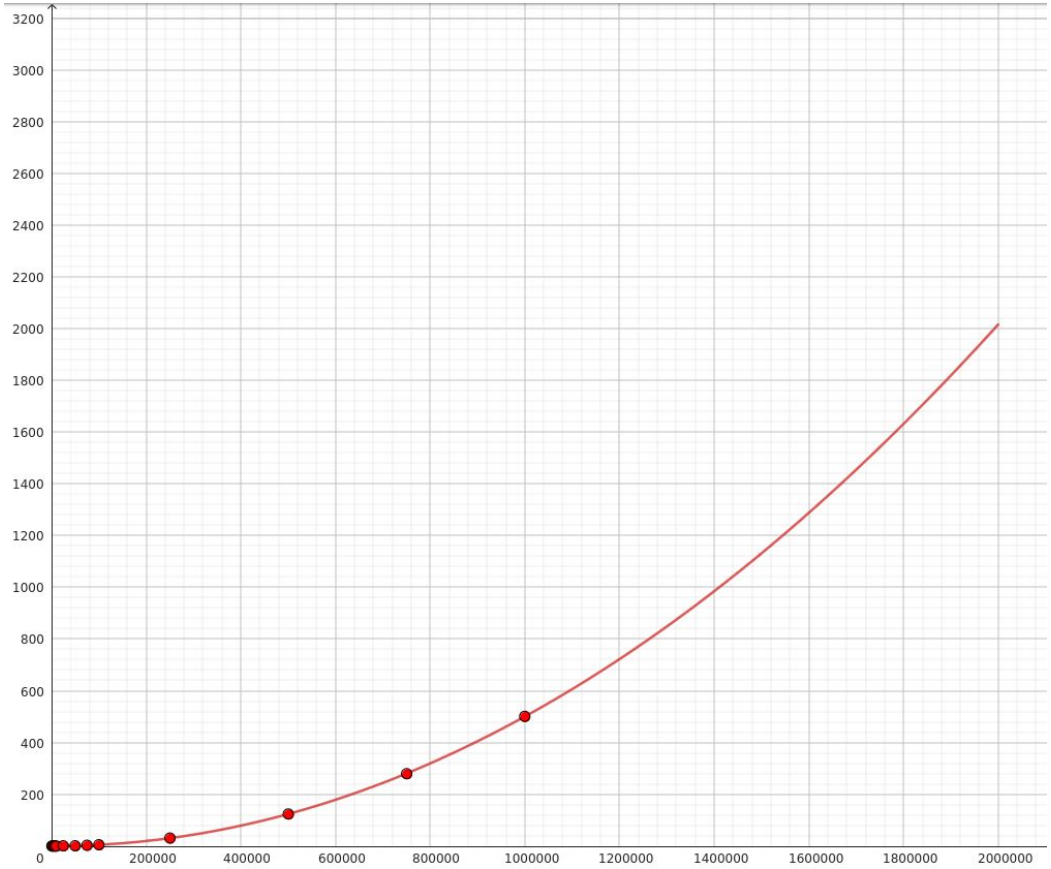
```
void bubble_sort2 (int array[], int tam)
{
    bool troca;
    for (int i = tam - 1; i >= 0; i--)
    {
        troca = FALSE;
        for (int j = 1; j <= i; j++)
        {
            if (array[j - 1] > array[j])
            {
                trocar(&array[j - 1], &array[j]);
                troca = TRUE;
            }
        }
        if (troca == FALSE)
            break;
    }
}
```

Eficiência dos algoritmos simples de ordenação

- Os métodos simples de ordenação que vimos possuem implementação fácil e direta. Todos resolvem o problema da ordenação. Dentre eles, **qual é o melhor? Como medir esse “melhor”?**

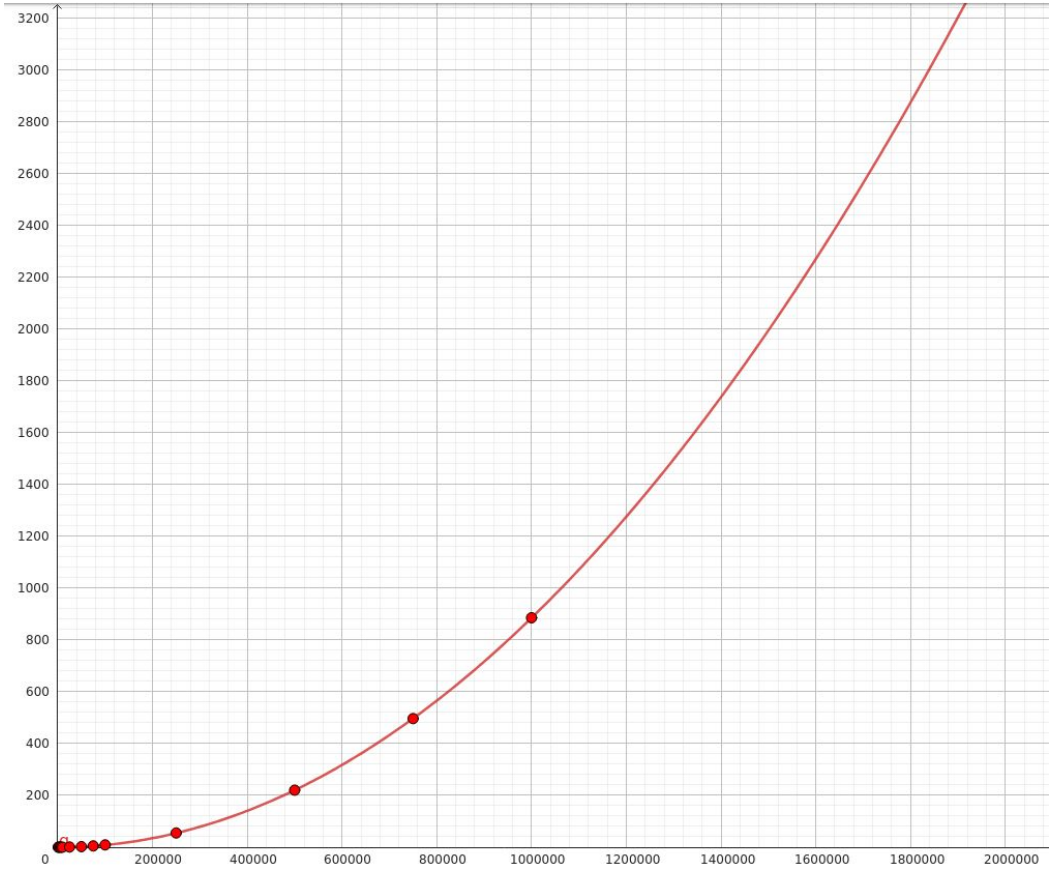
- Uma primeira abordagem seria **medir o tempo (em segundos) que cada algoritmo demora para executar** em diferentes tamanhos de arrays a serem ordenados. Vamos ver a seguir.

Eficiência dos algoritmos: Insertion Sort



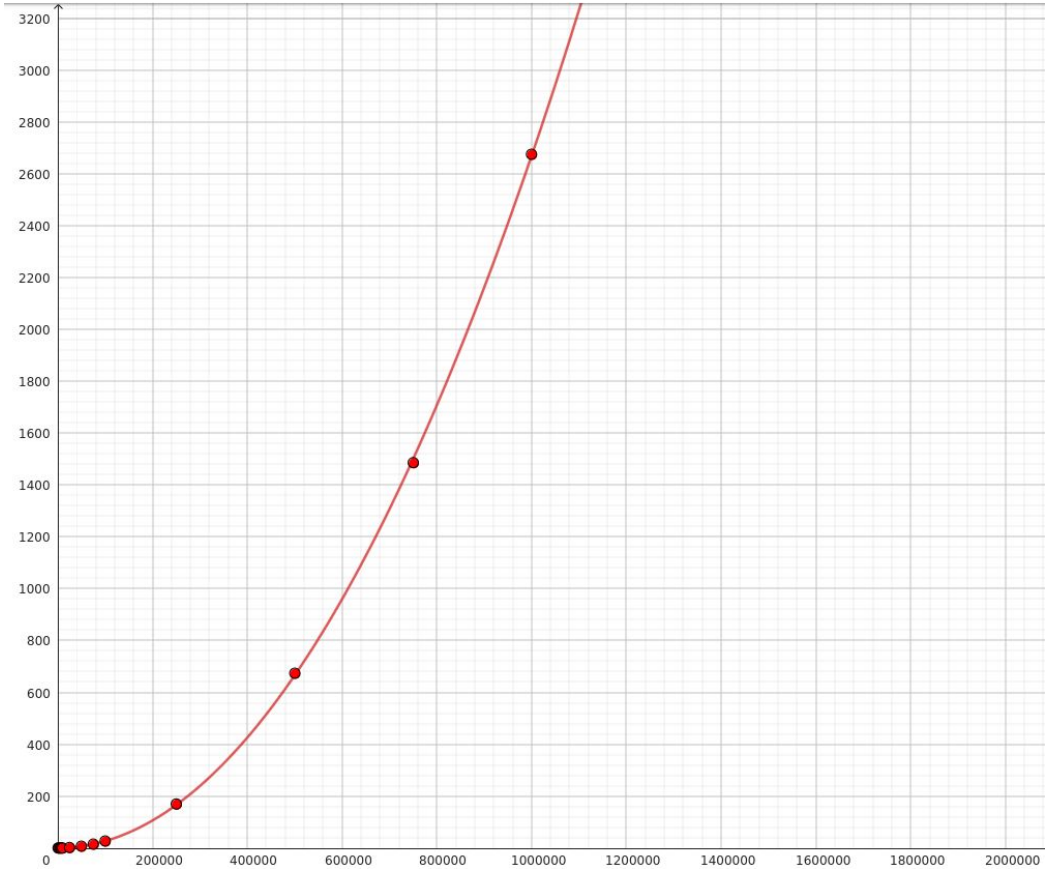
Insertion Sort	
N	T (s)
1.000	0.002665964
2.500	0.007699986
5.000	0.017275266
7.500	0.032358893
10.000	0.055539730
25.000	0.314092648
50.000	1.218458801
75.000	2.762859599
100.000	4.925750682
250.000	30.694431502
500.000	123.409269984
750.000	279.138751696
1.000.000	500.896221395

Eficiência dos algoritmos: Selection Sort



Selection Sort	
N	T (s)
1.000	0,004929366
2.500	0,010312282
5.000	0,027220584
7.500	0,054995412
10.000	0,094631551
25.000	0,558033754
50.000	2,207042104
75.000	4,959149477
100.000	8,791371990
250.000	55,038402343
500.000	220,460744582
750.000	495,756866120
1.000.000	884,514292304

Eficiência dos algoritmos: Bubble Sort



Bubble Sort	
N	T (s)
1.000	0.003878389
2.500	0.015195672
5.000	0.071982125
7.500	0.129035398
10.000	0.232493986
25.000	1.553097368
50.000	6.514312710
75.000	14.882730837
100.000	26.301346291
250.000	169.332235394
500.000	673.885925938
750.000	1486.167077780
1.000.000	2676.016201590

Eficiência dos algoritmos simples de ordenação

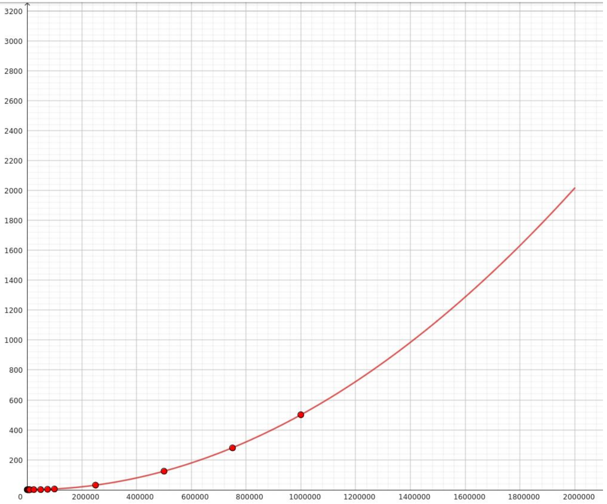
Insertion Sort		Selection Sort		Bubble Sort	
N	T (s)	N	T (s)	N	T (s)
1.000	0.002665964	1.000	0,004929366	1.000	0.003878389
2.500	0.007699986	2.500	0,010312282	2.500	0.015195672
5.000	0.017275266	5.000	0,027220584	5.000	0.071982125
7.500	0.032358893	7.500	0,054995412	7.500	0.129035398
10.000	0.055539730	10.000	0,094631551	10.000	0.232493986
25.000	0.314092648	25.000	0,558033754	25.000	1.553097368
50.000	1.218458801	50.000	2,207042104	50.000	6.514312710
75.000	2.762859599	75.000	4,959149477	75.000	14.882730837
100.000	4.925750682	100.000	8,791371990	100.000	26.301346291
250.000	30.694431502	250.000	55,038402343	250.000	169.332235394
500.000	123.409269984	500.000	220,460744582	500.000	673.885925938
750.000	279.138751696	750.000	495,756866120	750.000	1486.167077780
1.000.000	500.896221395	1.000.000	884,514292304	1.000.000	2676.016201590

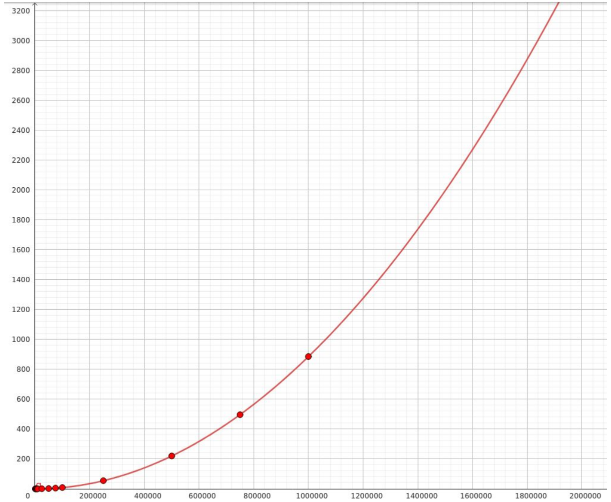
Eficiência dos algoritmos simples de ordenação

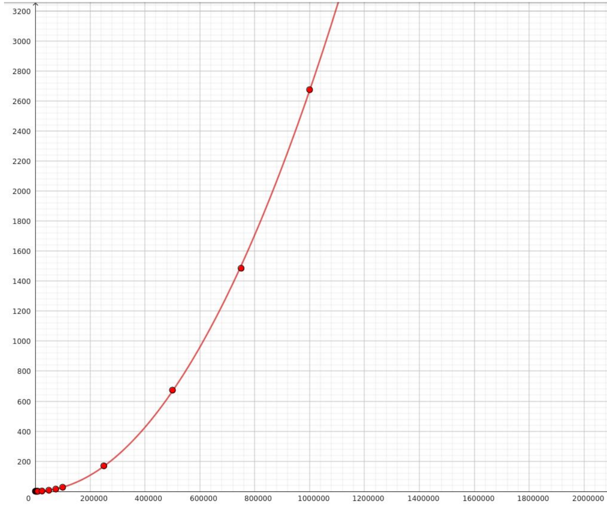
- Pelas medidas empíricas de qual algoritmo é melhor, PARECE que o Insertion Sort foi o mais rápido entre os três (existe um motivo; explicação na disciplina de Análise e Complexidade de Algoritmos).

- MAS: note que **o tempo de execução está aumentando aproximadamente de forma proporcional ao quadrado do número de elementos a serem ordenados**, nos três algoritmos:

Eficiência dos algoritmos simples de ordenação







- Apesar das diferenças nos tempos concretos (segundos), todos os algoritmos variam de forma aproximadamente **proporcional ao quadrado do número de elementos a serem ordenados**. A diferença está nos coeficientes a, b, c:

$$ax^2 + bx + c$$

Eficiência dos algoritmos simples de ordenação

- De certa forma, apesar das diferenças, esses três algoritmos são “parecidos”, da mesma “classe”: são algoritmos cujo **tempo de execução varia de acordo com o quadrado do número de elementos a serem ordenados**.

- Pode-se dizer que são da mesma “família”, que são “primos”.

Erro ao utilizar o tempo em segundos

- Um grande problema que ocorre ao utilizarmos o tempo em segundos para comparar algoritmos, é que esse tempo depende de vários fatores:
  - **Rapidez do processador**
  - **Linguagem de programação utilizada, do compilador ou interpretador, e das bibliotecas**
  - **Carga de trabalho do sistema no momento da avaliação**
  - **Diferenças no input (nas instâncias dos problemas a serem ordenados)**
  - **De uma implementação concreta em um sistema concreto com casos de teste concretos**

- Além disso podemos querer levar em conta, além do **tempo**, a quantidade de **memória**, a utilização da *banda de comunicação*, *consumo de energia* e outros fatores. Na maioria das vezes estamos interessados em:
  - **Tempo** (que não seja em segundos)
  - **Espaço** (consumo de memória)

Como analisar um algoritmo?

- Temos que ignorar os fatores que podem interferir no tempo em segundos. Assim, usamos um **MODELO DE COMPUTAÇÃO** chamado de **MÁQUINA DE ACESSO ALEATÓRIO** (**RAM: random-access machine**):
  - **Instruções executam de forma sequencial**, sem concorrência
  - **Cada instrução gasta a mesma quantidade de tempo** de qualquer outra instrução
  - **Cada acesso a um dado gasta a mesma quantidade de tempo** de qualquer outro acesso
  - O tempo em segundos de uma instrução ou um acesso a dado não é importante, pois ele é constante para as instruções e para os acessos (**cada instrução ou cada acesso a dado gasta uma quantidade constante de tempo**)
  - **Não leva em conta a hierarquia de memória** (cache, memória virtual)

- O modelo de máquina de acesso aleatório fornece uma primeira simplificação para a análise de algoritmos: a análise será feita nesta “máquina fictícia”.

Como analisar um algoritmo?

- E, para completar a tarefa de ignorar os fatores que influenciam no tempo em segundos que um algoritmo leva para executar, nós realizamos uma análise matemática teórica chamada de **análise assintótica**:
  - Mede a **ORDEM DE CRESCIMENTO DO TEMPO DE EXECUÇÃO** como uma **FUNÇÃO DO TAMANHO DA ENTRADA**, mas o tempo de execução não é medido em segundos, **O TEMPO DE EXECUÇÃO É MEDIDO COMO O NÚMERO DE INSTRUÇÕES E ACESSOS AOS DADOS EXECUTADOS** no modelo RAM.

- A análise assintótica é chamada de **análise da complexidade** de um algoritmo. Depende de matemática, combinatória, probabilidade, álgebra…

- **A relação entre a ordem de crescimento do tempo de execução e o tamanho da entrada** é chamada de **complexidade computacional** de um algoritmo.

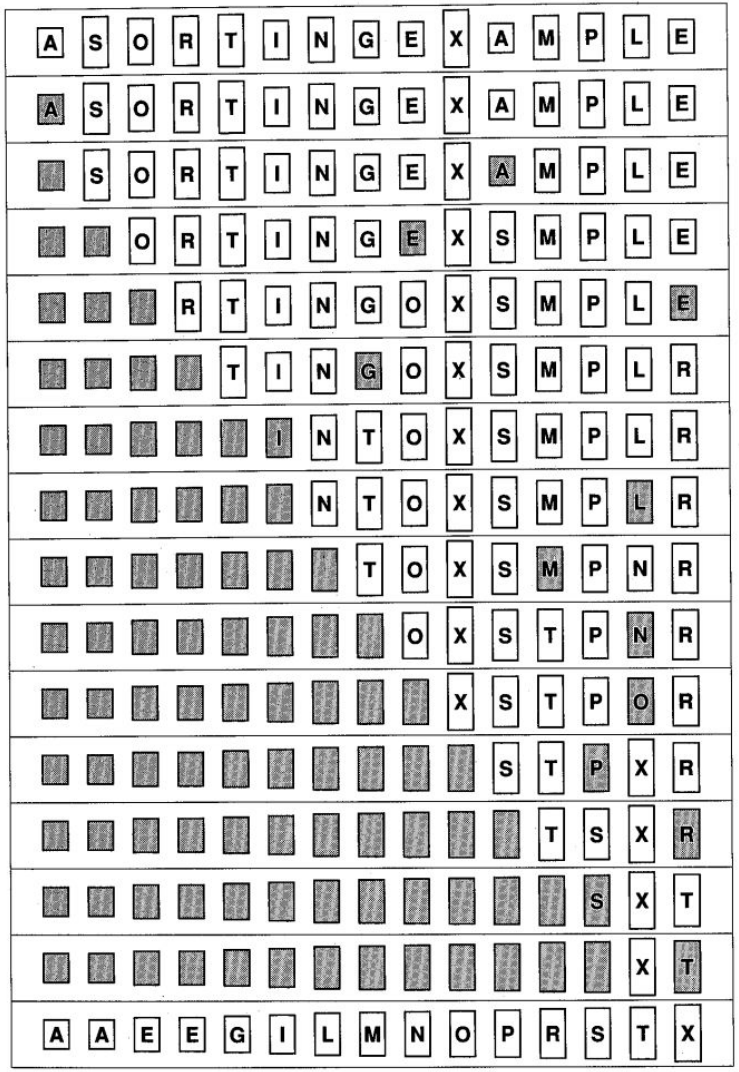
Análise informal de complexidade: Selection Sort

- Em cada iteração do loop, **de 1 até N-1**, há:
  - **N - i** comparações
  - **1** troca

- Comparações:
  (N-1) + (N-2) + … + 2 + 1 = N(N-1)/2

- Trocas:
  1 + 1 + 1 + … + 1 = N

- Resultado: **$N^2/2 - N/2 + N$** ⇐



Análise formal de complexidade: Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1 : i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

Análise formal de complexidade: Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1 : i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

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

Análise formal de complexidade: Insertion Sort

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1 = \frac{n(n+1)}{2} - 1$$

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

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

O tempo de execução é uma função quadrática do input! $an^2 + bn + c$

Análise formal de complexidade: Insertion Sort

- Um detalhe precisa ser considerado agora: mesmo com tudo que fizemos AINDA PODEMOS UMA INFLUÊNCIA EXTERNA NA ANÁLISE: **a instância do problema (características específicas dos dados) pode influenciar o tempo de execução**.

- A análise anterior considerava o **PIOR CASO** para o Insertion Sort, quando o array estava ordenado de modo reverso (ordem decrescente). Nesse caso o algoritmo tem que comparar cada A[i] com todos os elementos no subarray esquerdo, e o loop interno executa todas as vezes.

- Mas e se o array já estivesse ordenado em ordem crescente? Esse é o **MELHOR CASO** para o Insertion Sort, pois o loop interno não executaria!

Análise formal de complexidade: Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1 : i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

No melhor caso, nunca entra no loop (linhas 6 e 7 não executam).

O teste (linha 5) executa N-1 vezes.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) .$$

No MELHOR CASO, o tempo de execução é uma função linear do input! **an + b**

Análise formal de complexidade: Insertion Sort

- Existe também um **CASO MÉDIO**, no qual o array não está totalmente aleatório mas também não está totalmente ordenado de forma crescente. O problema é que o caso médio, grosso modo, é geralmente tão ruim quanto o pior caso.

- Em geral estaremos sempre preocupados com a complexidade (o tempo de execução) no PIOR CASO, porque:
  - Fornece um **limite superior para o tempo de execução para qualquer entrada** (a eficiência do algoritmo será, no mínimo, tão boa quanto a análise indica; podemos ter "sorte" e o algoritmo rodar mais rápido em situações específicas, mas garantimos que a eficiência não será pior do que a calculada)
  - **Ocorre frequentemente** na prática

Análise formal de complexidade: notação assintótica

- Até aqui achamos uma equação exata para a complexidade do Insertion Sort:

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

- Mas essa exatidão toda é desnecessária para o nosso objetivo, que é comparar algoritmos para saber qual é o mais eficiente. Na verdade, o que nos interessa na análise assintótica, é a **ORDEM DE CRESCIMENTO DO TEMPO DE EXECUÇÃO** em função do **TAMANHO DA ENTRADA**. Nós expressamos isso em uma notação especial, a **notação assintótica**.

Análise formal de complexidade: notação assintótica

- Na notação assintótica:
  - **Ignoramos os termos de menor ordem**
  - **Ignoramos os termos constantes (inclusive os coeficientes, pois são constantes)**

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

- Podemos dizer então que a complexidade do insertion sort é $\mathbf{n^2}$ (estamos sempre considerando o pior caso).

Análise formal de complexidade: notação assintótica

- Existem, na realidade, 5 notações assintóticas diferentes:
  - Big-O — Paul Bachmann, 1892 (matemático alemão)
  - Little-o — Edmund Landau, início do século XX (matemático alemão)
  - Big-Ômega Ω — Donald Knuth, 1976
  - Little-ômega ω — Donald Knuth, 1976
  - Theta Θ — Donald Knuth, 1976

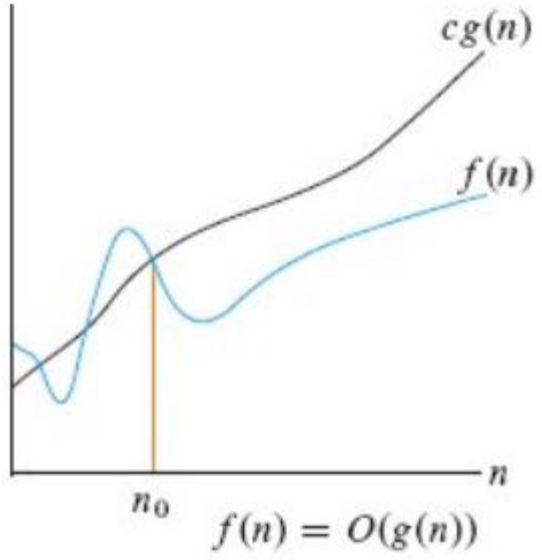
- Estamos interessados agora em apenas 3 notações:
  - Big-O
  - Big-Ômega Ω
  - Theta Θ

Notação assintótica: Big-O (noção informal)

- Caracteriza um **limite superior para o comportamento assintótico de uma função**. Mostra que o tempo de execução da função não cresce mais rápido do que uma certa taxa, baseada no termo de maior ordem.
  - **Cuidado com a interpretação do "não cresce mais rápido": não é mais lenta do que**

- Ex: $7n^3 + 100n^2 - 20n + 6$ é:
  - $O(n^3)$
  - $O(n^4)$
  - $O(n^5)$
  - $O(n^6)$

  - $O(n^c)$ para c >= 3.

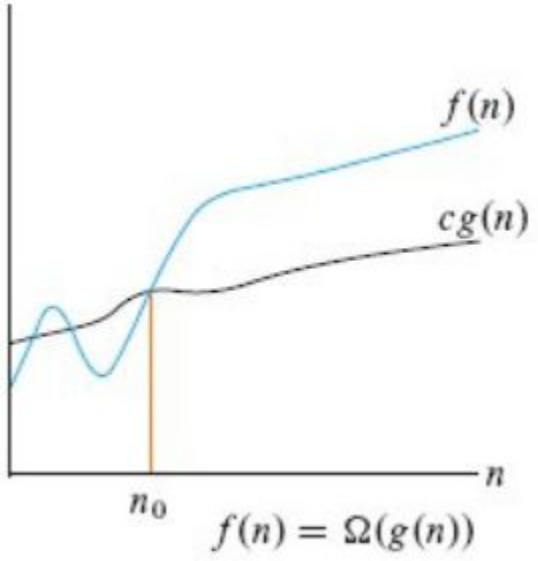


Notação assintótica: Big-Ω (noção informal)

- Caracteriza um **limite inferior para o comportamento assintótico de uma função**. Mostra que o tempo de execução da função cresce pelo menos tão rápido quanto uma certa taxa, baseada no termo de maior ordem.
  - **Cuidado com a interpretação do "cresce tão rápido quanto": não é mais rápida do que**

- Ex: $7n^3 + 100n^2 - 20n + 6$ é:
  - $\Omega(n^3)$
  - $\Omega(n^2)$
  - $\Omega(n^1)$

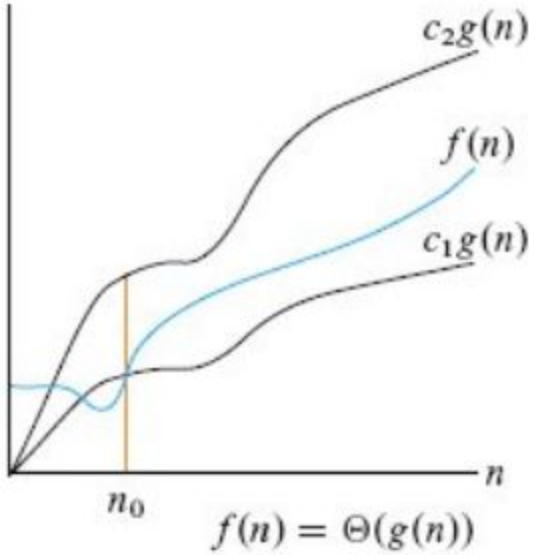
  - $\Omega(n^c)$ para c <= 3.



Notação assintótica: Θ (noção informal)

- Caracteriza um **limite estreito, rígido, para o comportamento assintótico de uma função**. Mostra que o tempo de execução da função cresce precisamente a uma certa taxa, baseada no termo de maior ordem.
  - **Para ser Θ, tem que ser O e Ω.**

- Ex: $7n^3 + 100n^2 - 20n + 6$ é:
  - $\Theta(n^3)$



Notação assintótica: Insertion Sort

- No pior caso:
  - $O(n^2)$
  - $\Omega(n^2)$
  - $\Theta(n^2)$

No melhor caso:

$O(n)$
$\Omega(n)$
$\Theta(n)$

  - Atenção: ao dizer que um algoritmo é $\Omega(n^2)$ no pior caso, estamos querendo dizer que **para todo input de tamanho n acima de um certo valor, existirá pelo menos 1 input de tamanho n para o qual o algoritmo executa em, pelo menos, tempo cn^2 para alguma constante positiva c. Isso não significa necessariamente que o algoritmo executa em pelo menos cn^2 para todos os casos.**

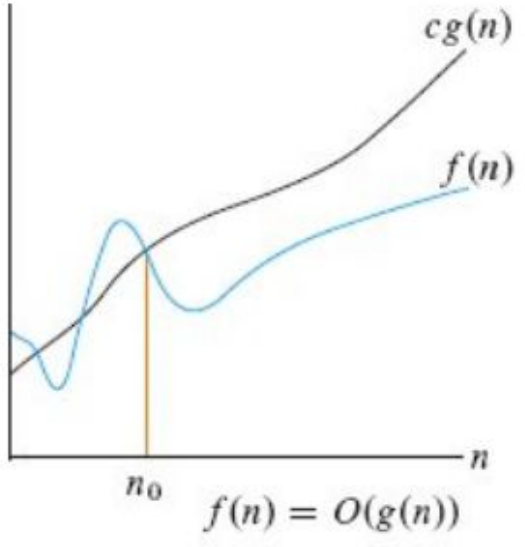
  - Atenção: ao dizer que um algoritmo é $\Theta(n^2)$ no pior caso, não significa dizer que ele executa em $\Theta(n^2)$ em todos os casos! Lembre-se de que, no melhor caso, o insertion sort é $O(n)$ e $\Omega(n)$ e, portanto, é $\Theta(n)$ no melhor caso.

Notação assintótica: Big-O (formalização)

- **f(n) = O(g(n))**
  Formalmente queremos dizer que **g(n)** é uma aproximação de **f(n)** com a seguinte característica: é possível encontrar constantes $\mathbf{n_0}$ e **c** tal que, para todo valor $\mathbf{n \geq n_0}$, a seguinte condição seja verdadeira:

  f(n) ≤ cg(n)

- Em outras palavras, desde que **n** seja grande o suficiente, a função **f(n)** sempre será **limitada superiormente por um múltiplo constante** da função **g(n)**. Dizemos que **g(n) domina assintoticamente** a função **f(n)**.

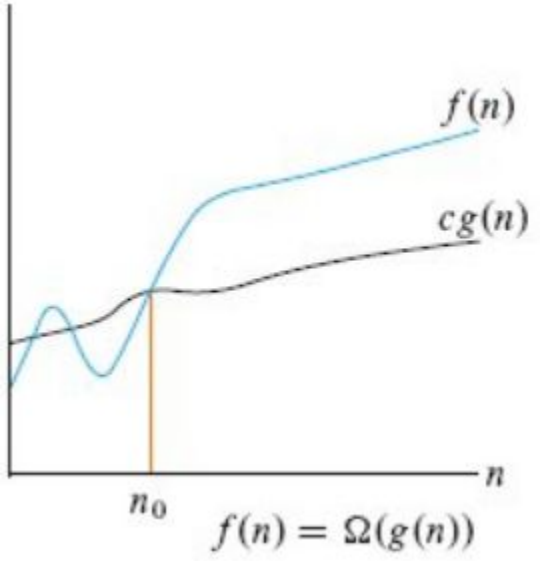


Notação assintótica: Big-Ω (formalização)

- **f(n) = Ω(g(n))**
  Formalmente queremos dizer que **g(n)** é uma aproximação de **f(n)** com a seguinte característica: é possível encontrar constantes $\mathbf{n_0}$ e **c** tal que, para todo valor $\mathbf{n \geq n_0}$, a seguinte condição seja verdadeira:

  cg(n) ≤ f(n)

- Em outras palavras, desde que **n** seja grande o suficiente, a função **f(n)** sempre será **limitada inferiormente por um múltiplo constante** da função **g(n)**.

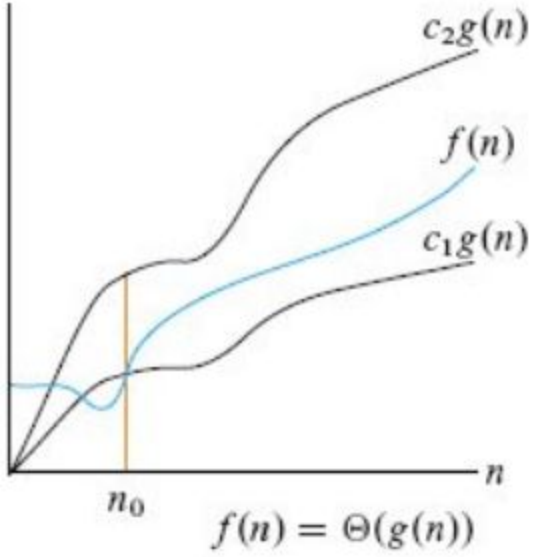


Notação assintótica: Θ (formalização)

- **f(n) = Θ(g(n))**
  Formalmente queremos dizer que **g(n)** é uma aproximação de **f(n)** com a seguinte característica: é possível encontrar constantes $\mathbf{n_0}$, $\mathbf{c_1}$ e $\mathbf{c_2}$ tal que, para todo valor $\mathbf{n \geq n_0}$, a seguinte condição seja verdadeira:

  $$\mathbf{c_1 g(n) \leq f(n) \leq c_2 g(n)}$$

- Em outras palavras, desde que **n** seja grande o suficiente, a função **f(n)** sempre será **limitada inferiormente e superiormente por um múltiplo constante** da função **g(n)**.



Notação assintótica: formalização

$$O(g(n)) = \{f(n) \mid \exists(c, n_0) > 0 \mid \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$$

$$\Omega(g(n)) = \{f(n) \mid \exists(c, n_0) > 0 \mid \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)\}$$

$$\Theta(g(n)) = \{f(n) \mid \exists(c_1, c_2, n_0) > 0 \mid \forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Mais detalhes na disciplina de Análise e Complexidade de Algoritmos!

Notação assintótica: exemplo de formalização

- Prove que **f(n) = 2n + 3** é **O(n)**.

$$O(g(n)) = \{f(n) \mid \exists(c, n_0) > 0 \mid \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$$

- Achar **c** e $\mathbf{n_0}$:
  - Vamos considerar **c = 3**. Então:
    2n + 3 ≤ 3n
    3 ≤ 3n - 2n
    3 ≤ n
  - Encontramos então duas constantes positivas, c = 3 e n_0 = 3, tais que para todos os valores de n ≥ 3, a seguinte relação é verdadeira:
    2n + 3 ≤ 3n
  - Portanto, f(n) = 2n + 3 é O(n).

Notação assintótica: exemplo de formalização

- Prove que **f(n) = 2n + 3** é **Ω(n)**.

$$\Omega(g(n)) = \{f(n) \mid \exists(c, n_0) > 0 \mid \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)\}$$

- Achar **c** e $\mathbf{n_0}$:
  - Vamos considerar **c = 1**. Então:
    n ≤ 2n + 3
    -3 ≤ 2n - n
    -3 ≤ n
  - Encontramos então duas constantes positivas, c = 3 e n_0 = 0, tais que para todos os valores de n ≥ 0, a seguinte relação é verdadeira:
    n ≤ 2n + 3
  - Portanto, f(n) = 2n + 3 é Ω(n).

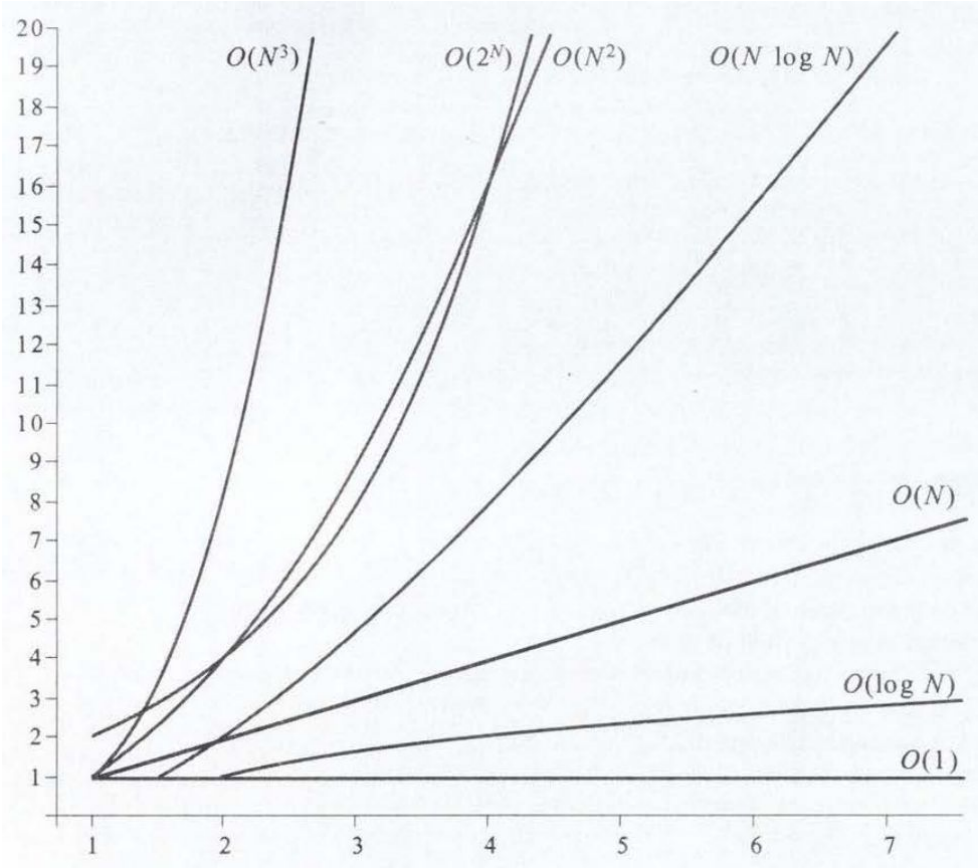
Notação assintótica: classes de complexidade

- Na programação, a maioria dos algoritmos pertence a uma de diversas classes comuns de complexidade, listadas a seguir em ordem crescente:
  - $O(1)$ constante
  - $O(\log_2 n)$ logarítmica
  - $O(n)$ linear
  - $O(n \log_2 n)$ log-linear
  - $O(n^2)$ quadrática
  - $O(n^3)$ cúbica
  - $O(2^n)$ exponencial
  - $O(n!)$ fatorial

- Um algoritmo de menor complexidade sempre será mais eficiente do que um de maior complexidade, para **N grande**. Se N é pequeno, termos de menor ordem e termos constantes podem fazer com que isso não seja verdade.

Notação assintótica: classes de complexidade

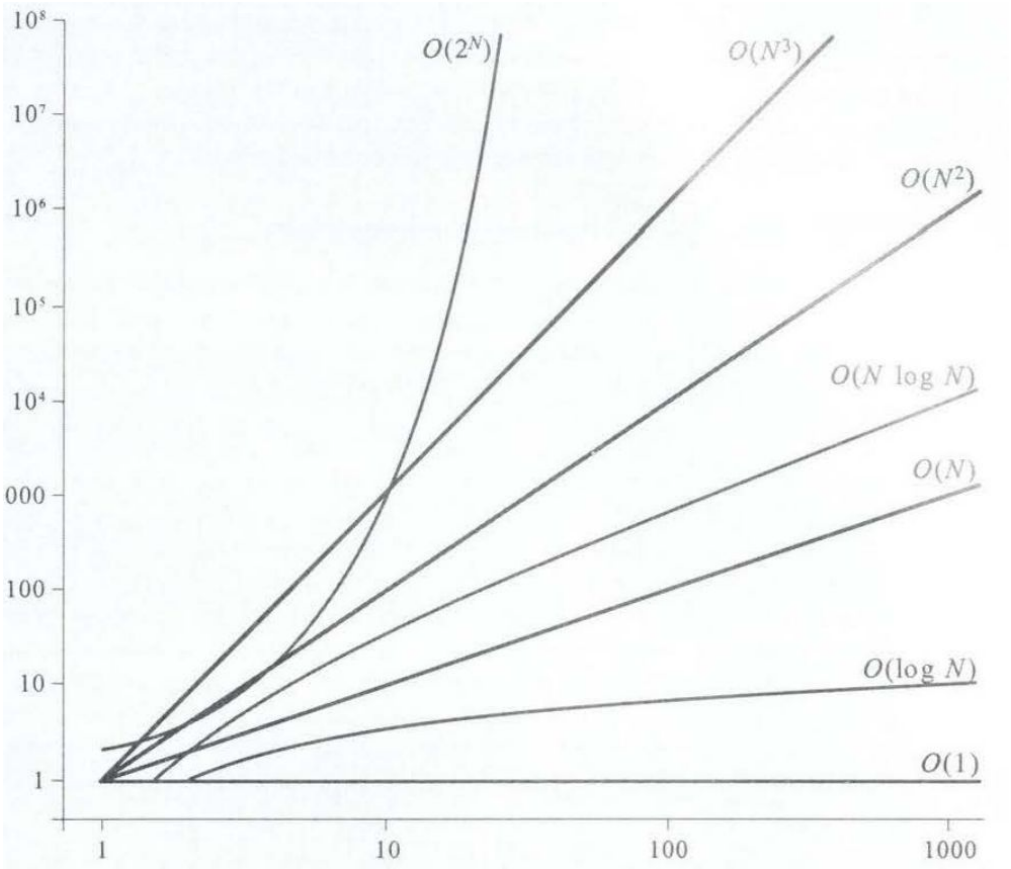
- Esse gráfico comum é um pouco “enganador”, pois está em escala linear, com N pequeno.



Notação assintótica: classes de complexidade

- Se trocarmos a escala para logarítmica, teremos um entendimento bem melhor de como as classes de complexidade se comportam para grandes valores de N.

- Algoritmos **polinomiais**: O(1), O(n), $O(n^2)$, $O(n^3)$, …
  - Em geral: $O(n^k)$
  - Retas com inclinação proporcional à k
  - Menores que $O(2^n)$ e O(n!)



Notação assintótica: características das classes

- O(1):
  - Complexidade constante
  - Independem do tamanho de N
  - Executam um número constante de instruções
  - Ex.: retornar o 1º elemento de um array

- $O(\log_2 n)$:
  - Complexidade logarítmica
  - Típico em algoritmos que transformam um problema em outros menores
  - Ex.: busca binária em array ordenado

- O(n):
  - Complexidade linear
  - Típico em algoritmos cujo trabalho é realizado uma vez sobre todos os elementos
  - Ex.: busca linear em array não ordenado

Notação assintótica: características das classes

- $O(n \log_2 n)$:
  - Complexidade log-linear
  - Típico em algoritmos que quebram um problema em outros problemas menores, resolvem cada um deles de modo independente e depois combinam as soluções
  - Ex.: merge sort

- $O(n^2)$:
  - Complexidade quadrática
  - Típico em algoritmos que processam dados aos pares, em um loop dentro de outro loop
  - Ex.: selection sort

- $O(n^3)$:
  - Complexidade cúbica
  - Típico em algoritmos que processam dados em três loops aninhados
  - Ex.: multiplicação de matrizes

Notação assintótica: características das classes

- $O(2^n)$:
  - Complexidade exponencial
  - Típico em algoritmos que usam força bruta para resolver problemas
  - Geralmente não são úteis na prática
  - Ex.: fibonacci por recursividade simples

- $O(n!)$:
  - Complexidade fatorial
  - Típico em algoritmos que usam força bruta para resolver problemas
  - Geralmente não são úteis na prática
  - Ex.: caixeiro viajante

Notação assintótica: tratabilidade de problemas

- Nem todos os problemas podem ser solucionados via computação, existem problemas **tratáveis** e problemas **intratáveis**

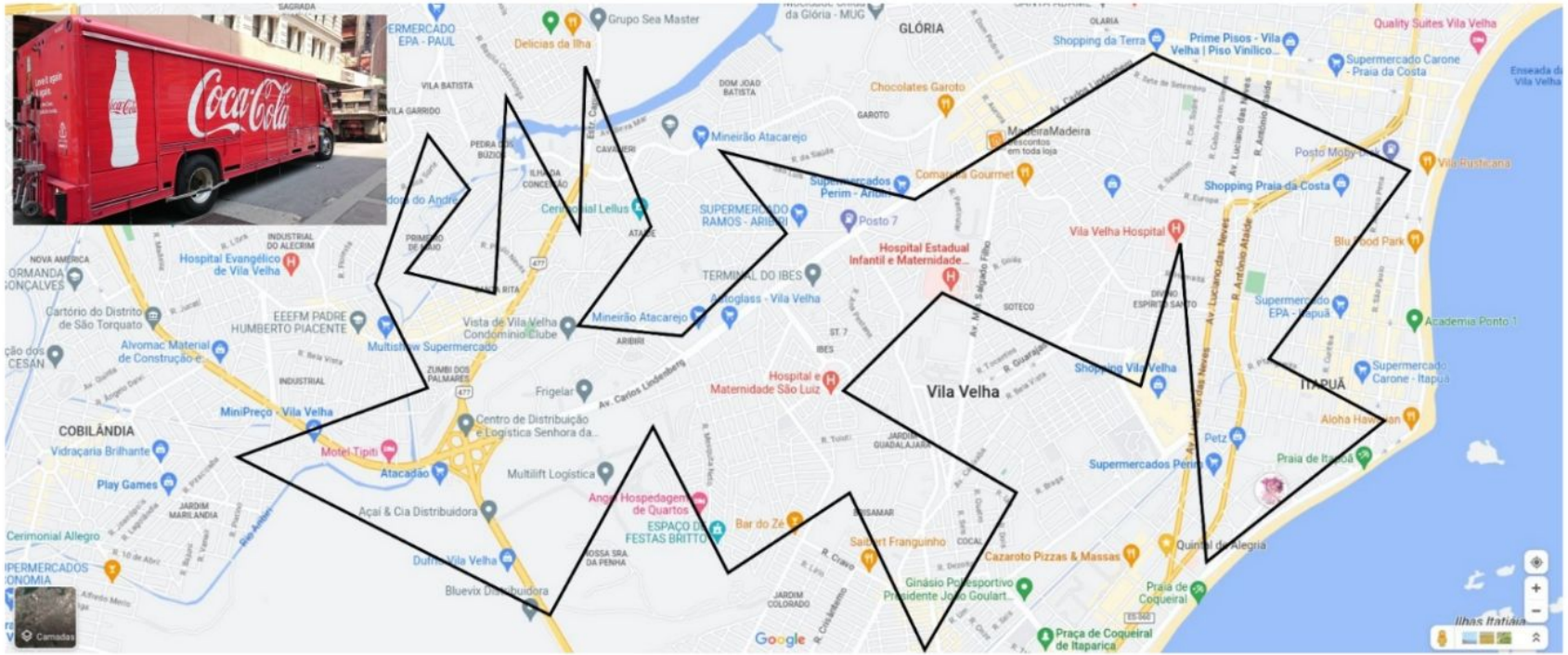
- Regra geral:
  - Problemas **tratáveis**: são problemas para os quais um **algoritmo polinomial** (ou mais eficiente) existe e pode ser implementado

  - Problemas **intratáveis**: são problemas para os quais somente algoritmos exponenciais ou fatoriais existem. Só podem resolver problemas com N muito pequeno.

- Observações: exceções que “confirmam” a regra
  - $O(2^n)$ é mais rápido que $O(n^5)$ valores de $N < 20$
  - Simplex é exponencial no pior caso, mas executa rápido na prática

Notação assintótica: tratabilidade de problemas

Qual a melhor rota possível para o caminhão, a rota ótima, com 31 bares?



Notação assintótica: tratabilidade de problemas

O Caixeiro Viajante

Bares	Intel Core i9-13900KS (6 GHz)	Supercomputador Frontier (1,1E18 FLOPS)
5	insignific.	insignific.
10	0,0005 seg	insignific.
15	3,4 min	insignific.
20	12 anos	2 seg
25	79 milhões de anos	5 meses
30	1,4 quatrilhões de anos	7,4 milhões de anos
35	53 sextilhões de anos	289 trilhões de anos

Notação assintótica: tratabilidade de problemas

Notação assintótica: tratabilidade de problemas

- Até hoje não se sabe se problemas para os quais só existem algoritmos exponenciais (chamados de **não polinomiais**) poderão ser resolvidos no futuro com algoritmos polinomiais.
    - Até hoje não se encontrou nenhum algoritmo polinomial para resolver o problema do caixeiro viajante, mas também não se provou que não exista um algoritmo polinomial para isso. Essa é uma das questões em aberto na computação:

Notação assintótica: tratabilidade de problemas

- Existe uma propriedade importante na questão P = NP? que é a seguinte:
  - **Se alguém achar um algoritmo polinomial para algum problema NP, então TODOS OS PROBLEMAS NP terão solução polinomial.**

- Quer ganhar 1 milhão de dólares? Ache a resposta!

https://www.claymath.org/millennium/p-vs-np/

Notação assintótica: utilidade

- Para comparar algoritmos, usamos a ordem de crescimento do tempo de execução: **o melhor algoritmo (o mais eficiente) será o que tiver a menor ordem de crescimento do tempo de execução**.

- Todos os algoritmos que vimos (insertion, bubble e selection sort) têm complexidade quadrática no pior caso, ou seja, $O(n^2)$: **a ordem de crescimento do tempo de execução é proporcional à N^2 para grandes valores de N**.

- E agora? Qual é o melhor dos três?

Algoritmos simples de ordenação: quando usar?

- São extremamente lentos! Nunca para grandes quantidades de elementos a serem ordenados e/ou quando os elementos estão totalmente aleatórios

- Situações nas quais vale a pena considerar:
  - A ordenação será feita uma única vez (ou poucas vezes)
  - O número de elementos não é grande
  - Os elementos estão quase ordenados (ou totalmente ordenados)
  - Quando há um grande número de chaves repetidas

- Como melhorar? Recursão!
  - Merge Sort
  - Quick Sort

Algoritmos sofisticados de ordenação: Merge Sort

- Por mais esquisito que pareça, a chave para encontrar uma alternativa melhor para ordenação está em reconhecer que algoritmos quadráticos como o Selection Sort tem uma virtude escondida:
  - Se o tamanho do problema dobrar, o tempo de execução aumenta por um fator de 4:

$$N \rightarrow (N)^2 \rightarrow N^2$$
$$2N \rightarrow (2N)^2 \rightarrow 4N^2$$

  - Se o tamanho do problema cair pela metade, o tempo de execução também diminui por um fator de 4:

$$N \rightarrow (N)^2 \rightarrow N^2$$
$$\frac{N}{2} \rightarrow \left(\frac{N}{2}\right)^2 \rightarrow \frac{N^2}{4}$$

Isso sugere que dividir um array no meio e usar uma estratégia de **dividir para conquistar** pode melhorar o tempo da ordenação!

Algoritmos sofisticados de ordenação: Merge Sort

- Numa “conta de padeiro”:

$$N = 500\,000 \quad \therefore \quad N^2 = 250\,000\,000\,000$$

$$N = 250\,000 \quad \therefore \quad N^2 = 62\,500\,000\,000$$

$$N = 250\,000 \quad \therefore \quad N^2 = 62\,500\,000\,000$$

$$125\,000\,000\,000 \quad (-50\%)$$

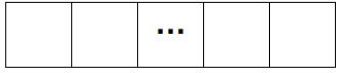
Algoritmos sofisticados de ordenação: Merge Sort

- Numa “conta de padeiro”:

$$N = 500\,000 \quad \therefore \quad N^2 = 250\,000\,000\,000$$

$N = 125\,000 \quad \therefore \quad N^2 = 15\,625\,000\,000$

$N = 125\,000 \quad \therefore \quad N^2 = 15\,625\,000\,000$



$N = 125\,000 \quad \therefore \quad N^2 = 15\,625\,000\,000$

$N = 125\,000 \quad \therefore \quad N^2 = 15\,625\,000\,000$

$$62\,500\,000\,000 \quad (-75\%)$$

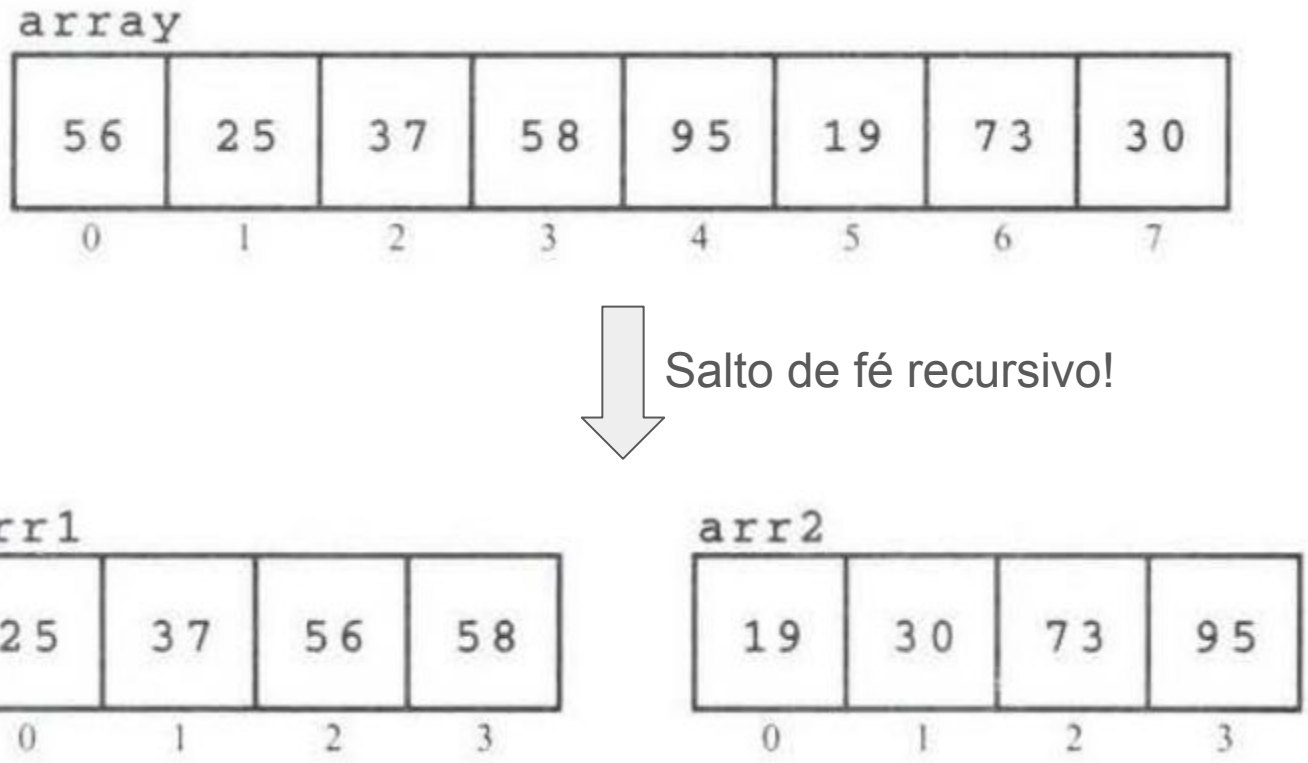
Algoritmos sofisticados de ordenação: Merge Sort

- Aparentemente, usar uma estratégia de dividir para conquistar simplifica o problema de ordenação. Mais ainda: agora que já sabemos que é possível dividir o array para aumentar a performance, **será que é possível aplicar um algoritmo para ordenar recursivamente cada subarray**?
    - Existe alguma decomposição recursiva possível? Eu consigo quebrar o problema em subproblemas menores da mesma forma?
    - Existe algum caso simples? Eu consigo achar um caso que é resolvido facilmente, sem nenhuma outra chamada recursiva?

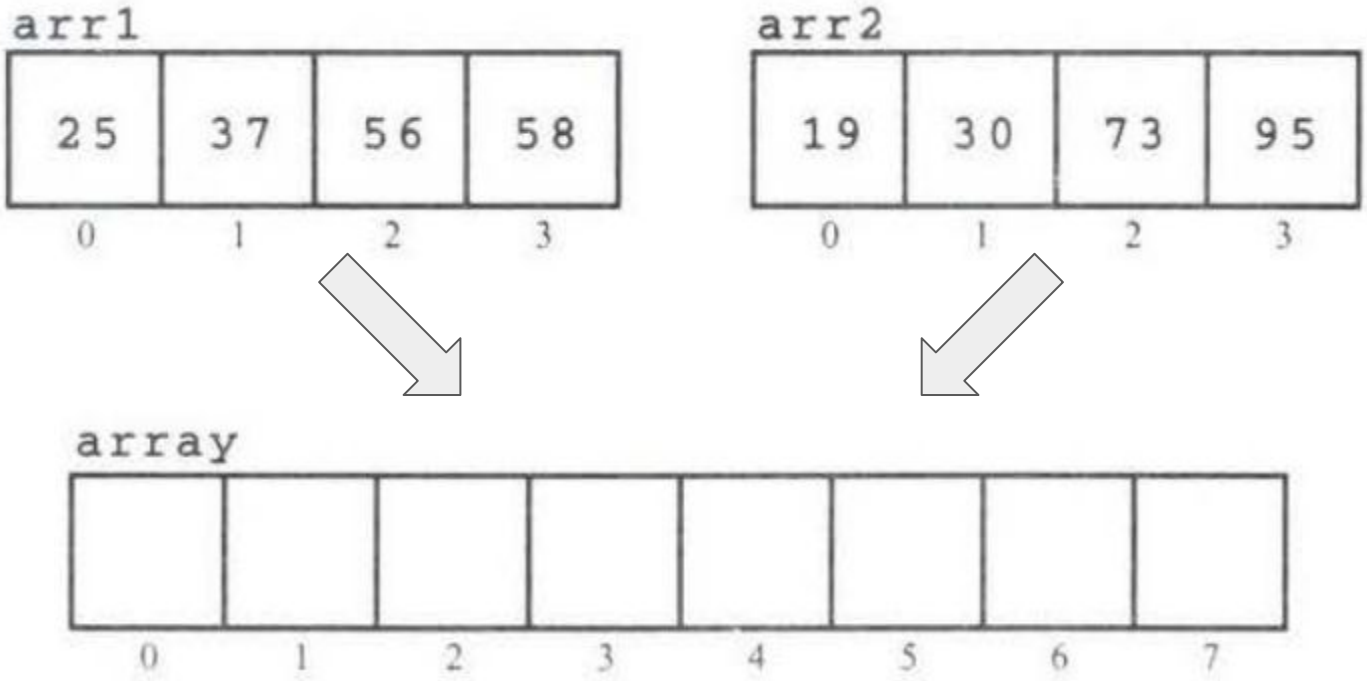
Algoritmos sofisticados de ordenação: Merge Sort

- Aparentemente, usar uma estratégia de dividir para conquistar simplifica o problema de ordenação. Mais ainda: agora que já sabemos que é possível dividir o array para aumentar a performance, **será que é possível aplicar um algoritmo para ordenar recursivamente cada subarray**?
    - Existe alguma decomposição recursiva possível? Eu consigo quebrar o problema em subproblemas menores da mesma forma?
        - **Sim!**
    - Existe algum caso simples? Eu consigo achar um caso que é resolvido facilmente, sem nenhuma outra chamada recursiva?
        - **Sim, se o subarray estiver vazio ou só tiver 1 único elemento!**

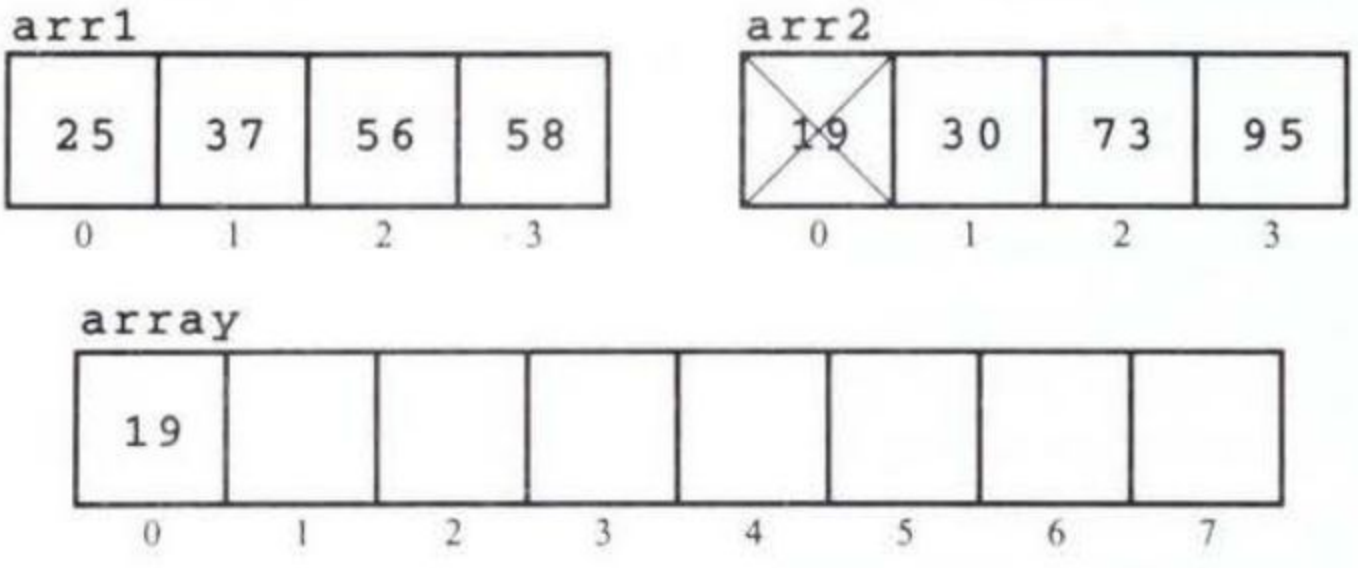
Algoritmos sofisticados de ordenação: Merge Sort



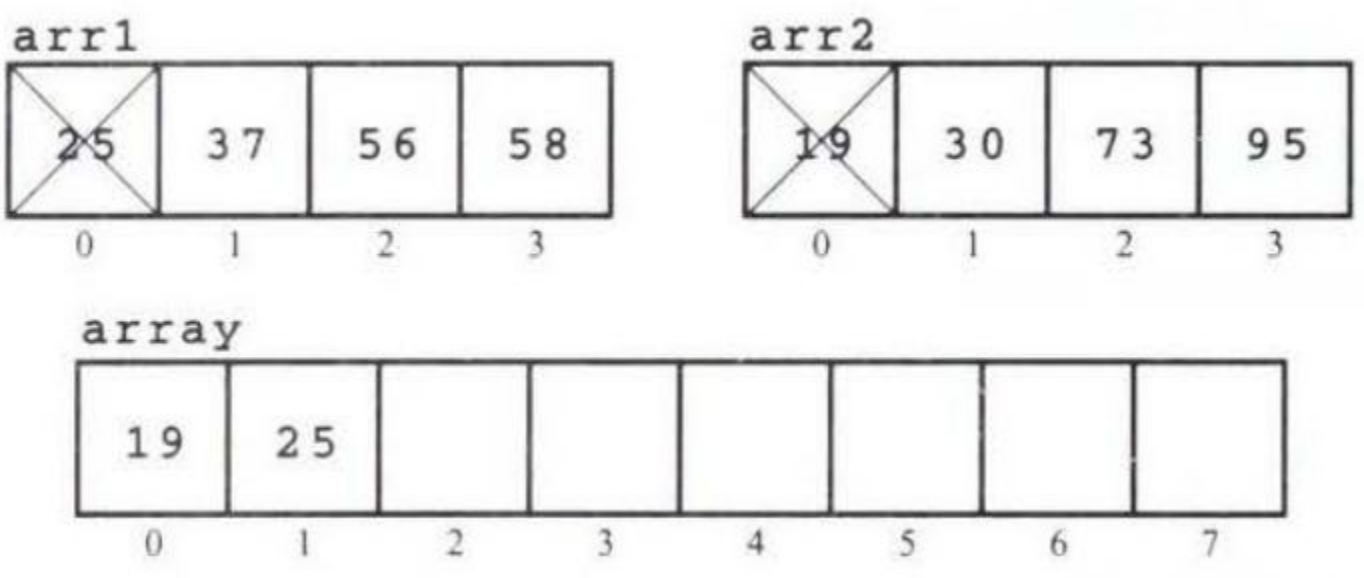
Algoritmos sofisticados de ordenação: Merge Sort



Algoritmos sofisticados de ordenação: Merge Sort



Algoritmos sofisticados de ordenação: Merge Sort



Algoritmos sofisticados de ordenação: Merge Sort

- É possível fazer o megre de 2 arrays (como demonstrado), de 3 arrays e de N arrays, desde que todos os arrays estejam ordenados.

- A idéia básica é sempre a mesma: escolher o menor elemento dentre todos os arrays que estão sendo combinados, e colocar esse elemento na posição correta do array final.

- Para o merge sort que estamos estudando, só precisaremos fazer o merge de dois arrays.

- Se um array terminar antes do outro, basta colocar todos os elementos do array que não terminou ainda.

Algoritmos sofisticados de ordenação: Merge Sort

- Merge sort em pseudocódigo:
    - Verificar se o array está vazio ou se só tem um elemento (são os casos simples). Se isso for verdade o array já está ordenado (trivialmente) e você só retorna a função sem fazer nada.
    - Se o array tem mais de 1 elemento, divida o array em 2 subarrays, cada um com a metade do tamanho do array original.
    - Ordene recursivamente cada subarray
    - Ao final, combine (merge) os dois subarrays no array original.

Algoritmos sofisticados de ordenação: Merge Sort

```
void merge_sort (int array[], int n)
{
    // Caso simples da recursão: um array com 1 elemente ou vazio
    if (n <= 1)
        return;

    // arrayE = subarray à esquerda; arrayD = subarray à direita
    // nE = tamanho do subarray à esquerda; nD = tamanho do subarray à direita
    int *arrayE, nE, *arrayD, nD;

    // Calcula limites dos subarrays:
    nE = n / 2;
    nD = n - nE;

    // Cria os subarrays à esquerda e à direita:
    arrayE = copiar_sub_array(array, 0, nE);
    arrayD = copiar_sub_array(array, nE, nD);

    // Casos recursivos: ordena os subarrays à esquerda e à direita
    merge_sort(arrayE, nE);
    merge_sort(arrayD, nD);

    // Após a recursão, combina os resultados:
    combinar(array, arrayE, nE, arrayD, nD);

    // Libera os subarrays criados:
    FreeBlock(arrayE);
    FreeBlock(arrayD);
}
```

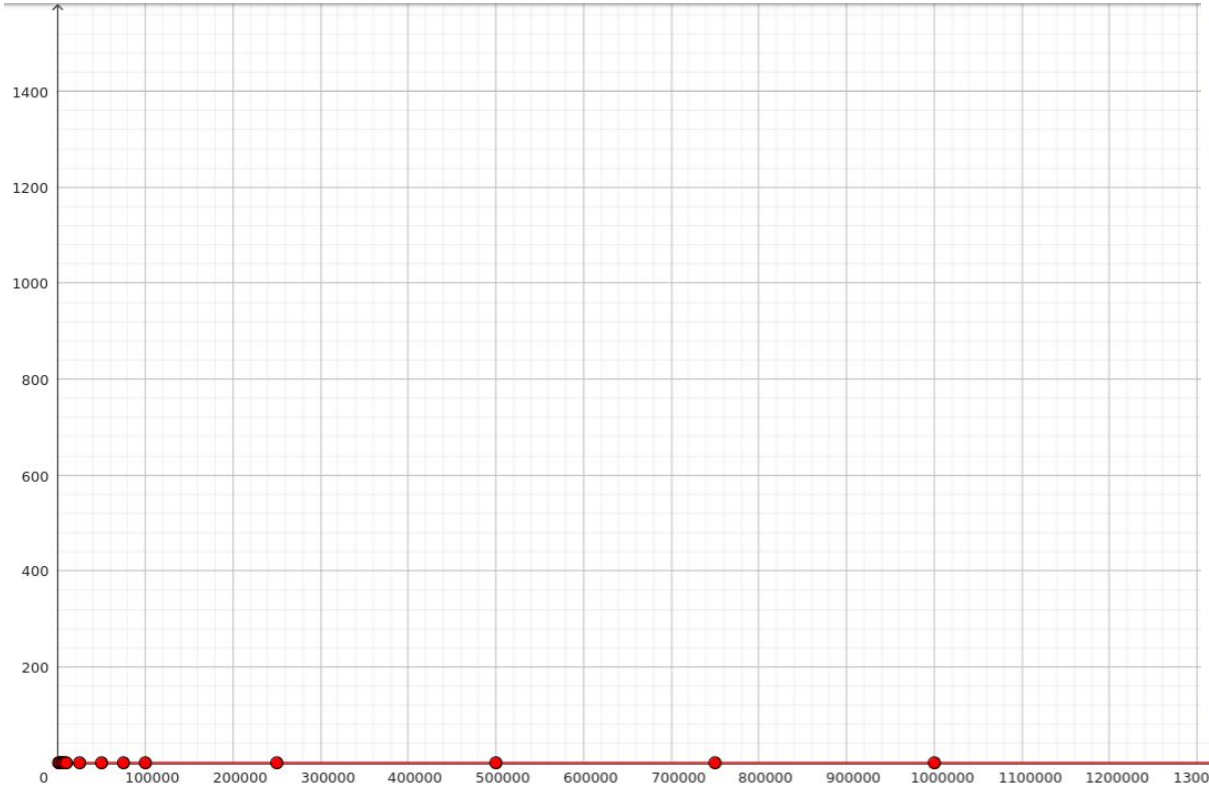
Algoritmos sofisticados de ordenação: Merge Sort

```
/**
 * Função: copiar_sub_array
 * Uso: copiar_sub_array(array, inicio, n);
 * ----------------------------------------
 * Esta função faz uma cópia de um subarray de um array de inteiros e retorna um
 * ponteiro para o novo array dinâmico contendo os novos elementos. O subarray
 * começa na posição de "inicio" do array original, e continua por "n"
 * elementos. Esta função só existe para tornar o código do procedimento
 * merge_sort mais legível. Esta função faz uso da função "NewArray" da CSLIB,
 * que aloca um novo array dinamicamente e retorna um ponteiro para esse array.
 */

static int *copiar_sub_array (int array[], int inicio, int n)
{
    int *resultado;
    resultado = NewArray(n, int);

    for (int i = 0; i < n; i++)
    {
        resultado[i] = array[inicio + i];
    }

    return resultado;
}
```

Algoritmos sofisticados de ordenação: Merge Sort

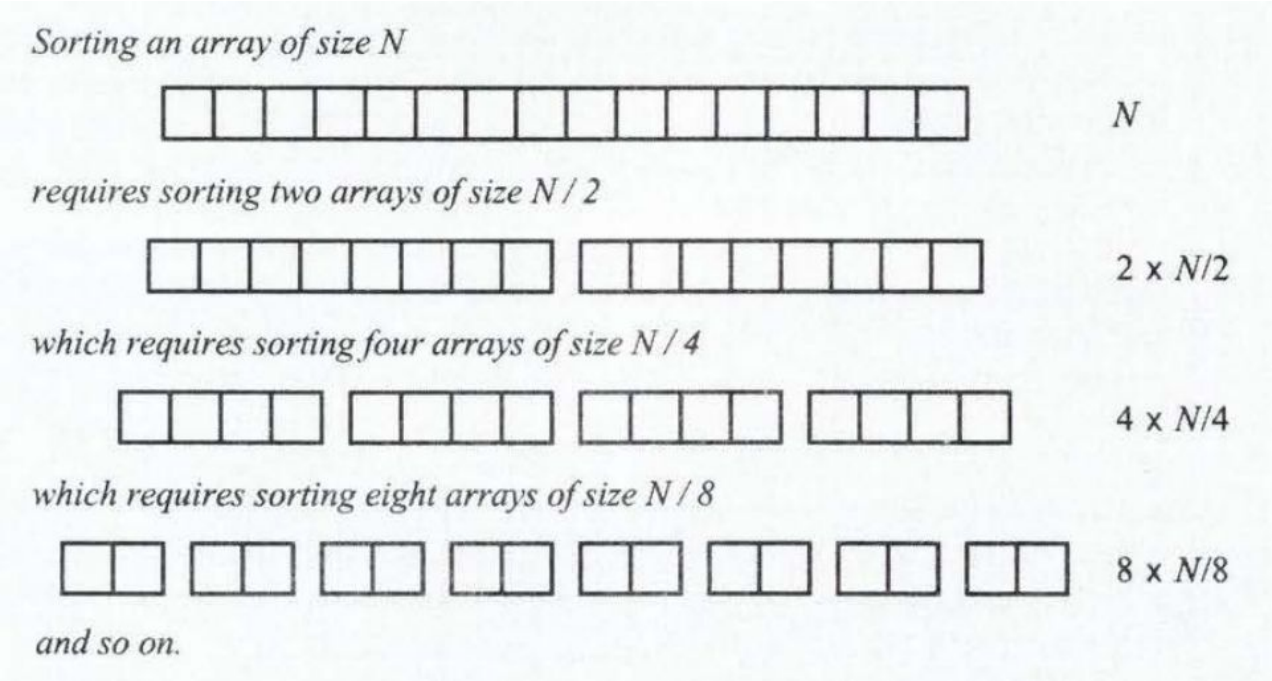
```
/**
 * Procedimento: combinar
 * Uso: combinar(array, arrayE, nE, arrayD, nD);
 * ---------------------------------------------
 * Este procedimento realiza a combinação (merge) de dois arrays já previamente
 * ordenados (arrayE, arrayD) em um único array de saía (array). Como os arrays
 * de entrada já estão ordenados, esta implementação sempre seleciona o primeiro
 * elemento não utilziado em um dos dois arrays de entrada para preencher a
 * próxima posição do array de saída.
 */

static void combinar (int array[], int arrayE[], int nE, int arrayD[], int nD)
{
    // Índices para as posições do array (p), e dos subarrays à esquerda (pE) e
    // à direita (pD):
    int p, pE, pD;
    p = pE = pD = 0;

    // Enquanto há elementos nos dois subarrays:
    while (pE < nE && pD < nD)
        array[p++] = (arrayE[pE] < arrayD[pD]) ? arrayE[pE++] : arrayD[pD++];

    // Se o subarray da direita terminou mas o da esquerda contém elementos:
    while (pE < nE)
        array[p++] = arrayE[pE++];

    // Se o subarray da esquerda terminou mas o da direita contém elementos:
    while (pD < nD)
        array[p++] = arrayD[pD++];
}
```

Complexidade do Merge Sort (informal)



Merge Sort	
N	T (s)
1.000	0.000618854
2.500	0.001710206
5.000	0.003546991
7.500	0.005063314
10.000	0.005606306
25.000	0.006829420
50.000	0.009053267
75.000	0.013589757
100.000	0.017126127
250.000	0.042460661
500.000	0.088629550
750.000	0.136676137
1.000.000	0.182864090

Complexidade do Merge Sort (informal)



- **A complexidade será a soma dos tempos de execução em cada nível. Em cada nível o trabalho será proporcional à N. O problema é achar quantos níveis recursivos teremos.**

Complexidade do Merge Sort (informal)

- Em cada nível da hierarquia, o valor de N é dividido por 2. O número total de níveis é igual ao número de vezes que você consegue dividir N por 2 até chegar em 1. Em termos matemáticos, queremos achar o númeo K tal que:

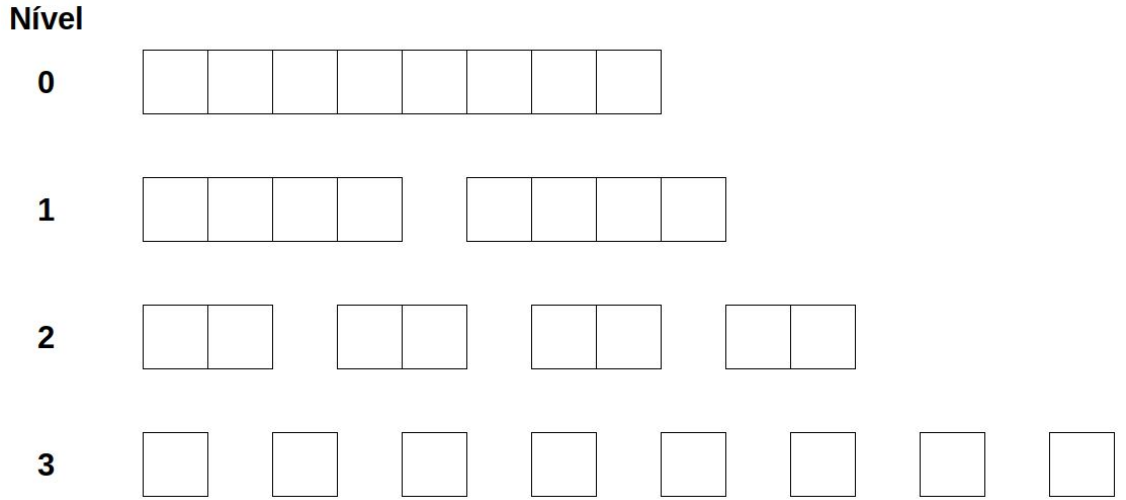
  $N = 2^K$

- Resolvendo temos:

  $K = \log_2 N$

- Como em cada nível a complexidade é proporcional à N, **em todos os casos**:

  $O(N \log_2 N)$



Complexidade do Merge Sort (informal)

- O que está ocorrendo aqui?

N	Selection sort	Merge sort
10	0.12 msec	0.54 msec
20	0.39 msec	1.17 msec
40	1.46 msec	2.54 msec
100	8.72 msec	6.90 msec
200	33.33 msec	14.84 msec
400	135.42 msec	31.25 msec
1000	841.67 msec	84.38 msec
2000	3.35 sec	179.17 msec
4000	13.42 sec	383.33 msec
10,000	83.90 sec	997.67 msec

Complexidade do Merge Sort (formal)

- Para obtermos formalmente a complexidade do Merge Sort, temos que resolver a seguinte **relação de recorrência** (equações ou desigualdades que descrevem uma função em termos de seu valor para entradas menores):

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) & \text{se } n > 1 \end{cases}$$

- Não vou entrar nesse nível de detalhe nesta disciplina (você na disciplina de Análise e Complexidade de Algoritmos). Vou deixar um material extra de leitura, para os interessados.

Algoritmos sofisticados de ordenação: Quick Sort

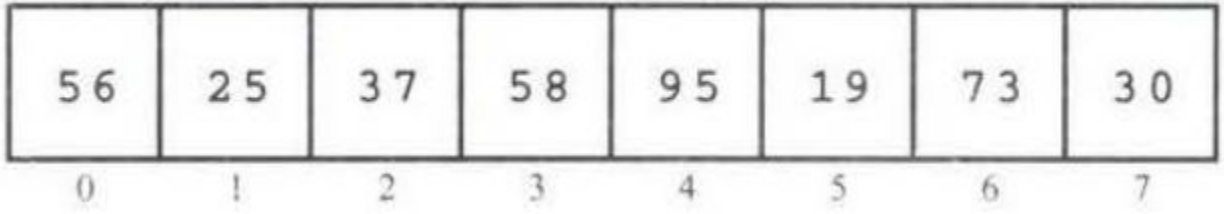
- Criado pelo cientista da computação britânico C. A. R. (Tony) Hoare, em 1961, e é um dos mais utilizados até hoje.

- É semelhante ao Merge Sort, no uso de dividir para conquistar.

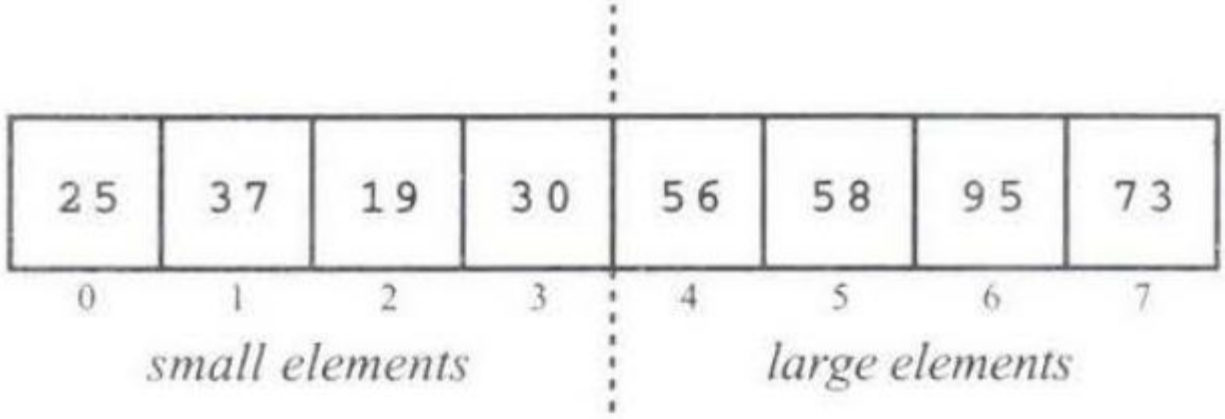
- Difere do Merge Sort na maneira de dividir o array:
  - **E se fizéssemos uma 1ª passagem pelo array, alterando as posições dos elementos de forma que os valores “menores” ficassem no início do array e os valores “maiores” ficassem no final do array, para alguma definição apropriada das palavras “menores” e “maiores”?**

Algoritmos sofisticados de ordenação: Quick Sort

- Considere o seguinte array: metade dos elementos são menores do que 50:

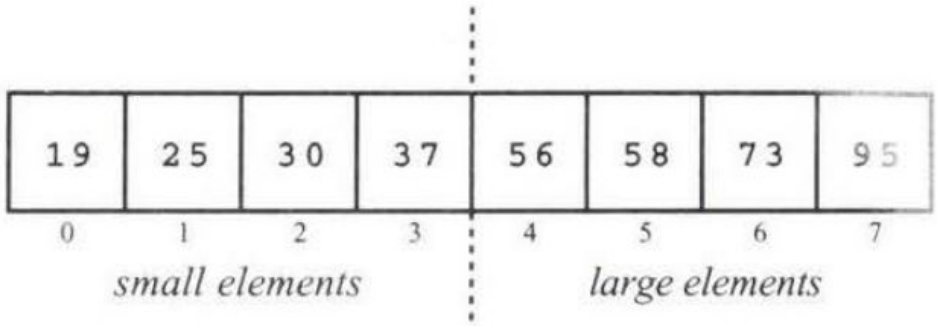


- Poderíamos definir “menores” como sendo menos de 50, e “maiores” como sendo 50 ou mais, e assim teríamos a ordenação ao lado (uma das possíveis ordenações)



Algoritmos sofisticados de ordenação: Quick Sort

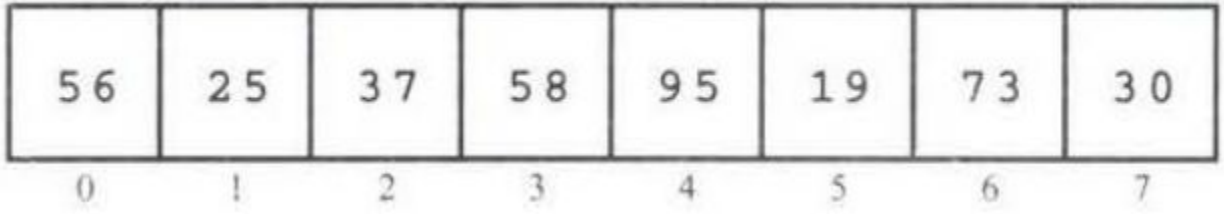
- Tudo que resta a fazer agora é ordenar recursivamente os dois subarrays do mesmo modo, usando recursão. No final teríamos o array completamente ordenado:



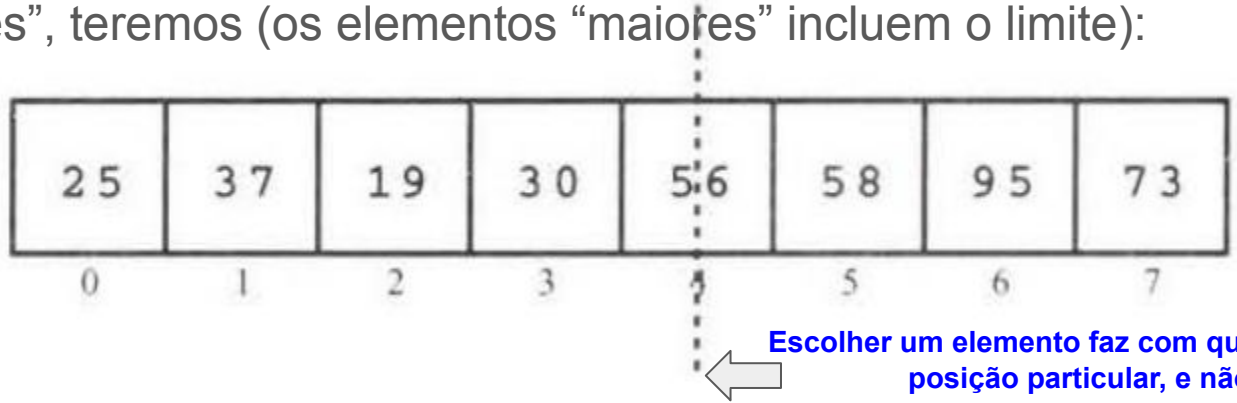
- Se pudéssemos escolher sempre o limite ótimo entre os elementos maiores e menores, dividiríamos o array sempre no meio e ele seria qualitativamente igual ao Merge Sort. Na prática isso exigiria uma passagem pelo array inteiro e não fazemos isso: **escolhemos ao acaso um elemento para ser o limite**. Uma prática comum é usar o primeiro ou último elemento.

Algoritmos sofisticados de ordenação: Quick Sort

- Considerando novamente o array anterior:

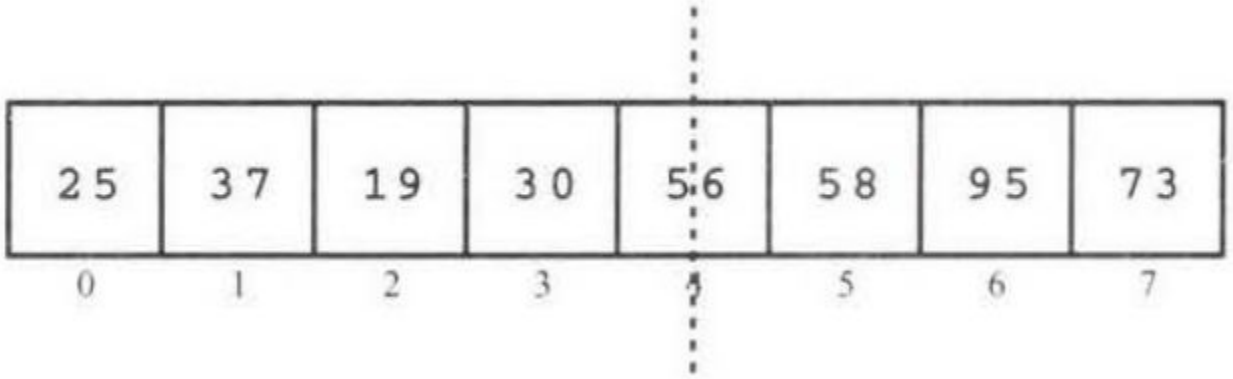


- Se escolhermos o 1º elemento para ser o limite entre elementos “menores” e “maiores”, teremos (os elementos “maiores” incluem o limite):



Algoritmos sofisticados de ordenação: Quick Sort

- As chamadas recursivas agora devem fazer o seguinte:
  - Ordenar os elementos nas posições 0 a 3 (subarray à esquerda do limite)
  - Ordenar os elementos nas posições 5 a 7 (subarray à direita do limite)
  - O limite já está no lugar correto



Algoritmos sofisticados de ordenação: Quick Sort

- Em pseudocódigo, o Quick Sort é assim:
  - O caso simples ocorre quando o array está vazio ou contém 1 elemento, o que significa que o array já está ordenado (trivialmente)

  - Escolher um dos elementos do array para servir como limite entre os elementos menores e os elementos maiores. Esse elemento é chamado de **PIVÔ**.
    - O primeiro, último ou o elemento do meio
    - Um elemento aleatório
    - Outro método de escolha (mediana de 3 ou 5 valores)

  - Reordenar os elementos de forma que os menores fiquem à esquerda do PIVÔ, o os maiores (ou iguais) fiquem à direita do PIVÔ. Esse processo é chamado de **PARTICIONAR** o array.

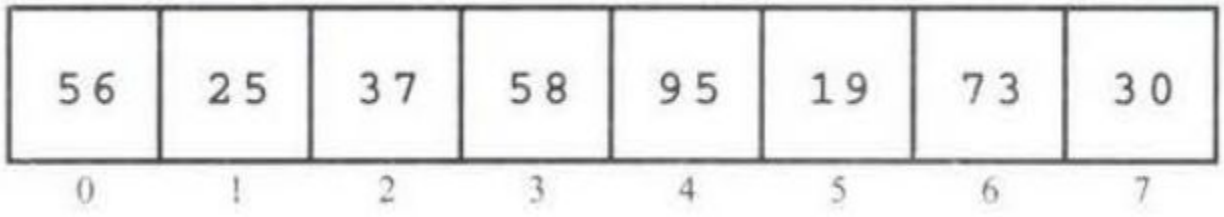
  - Ordenar, recursivamente, os elementos em cada um dos subarrays particionados.

Quick Sort: particionamento do array

- A escolha do pivô é simples (escolheremos sempre o 1º elemento). O processo de particionamento do array é mais complexo.

- O particionamento do array tem 3 objetivos:
  - deixar os elementos menores à esquerda do pivô
  - deixar o pivô na posição correta
  - deixar os elementos maiores (ou iguais ao pivô) à direita do pivô

- Ilustração do processo de particionamento com o array:

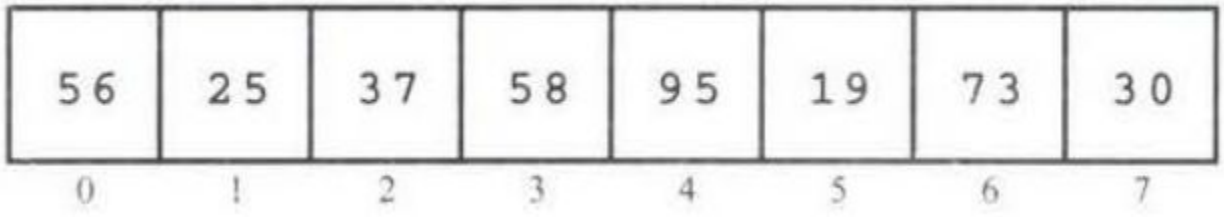


Quick Sort: particionamento do array

- A escolha do pivô é simples (escolheremos sempre o 1º elemento). O processo de particionamento do array é mais complexo.

- O particionamento do array tem 3 objetivos:
  - deixar os elementos menores à esquerda do pivô
  - deixar o pivô na posição correta
  - deixar os elementos maiores (ou iguais ao pivô) à direita do pivô

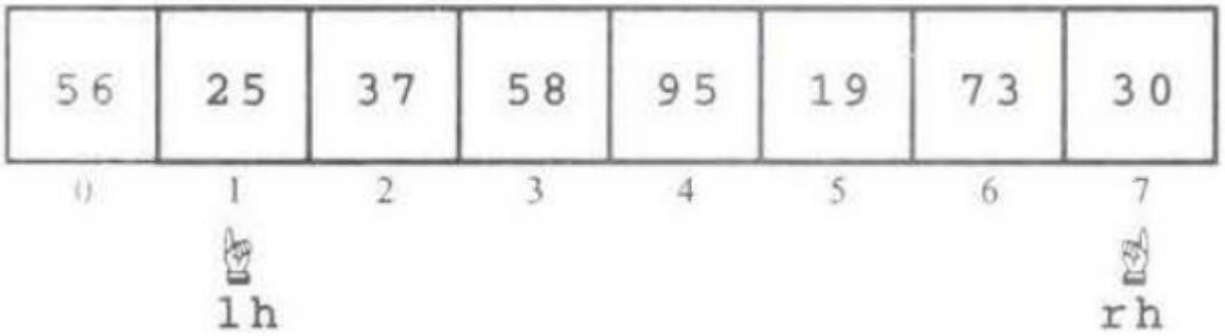
- Ilustração do processo de particionamento com o array:



Quick Sort: particionamento do array

- O elemento escolhido como PIVÔ será o primeiro, o 56 no índice 0.

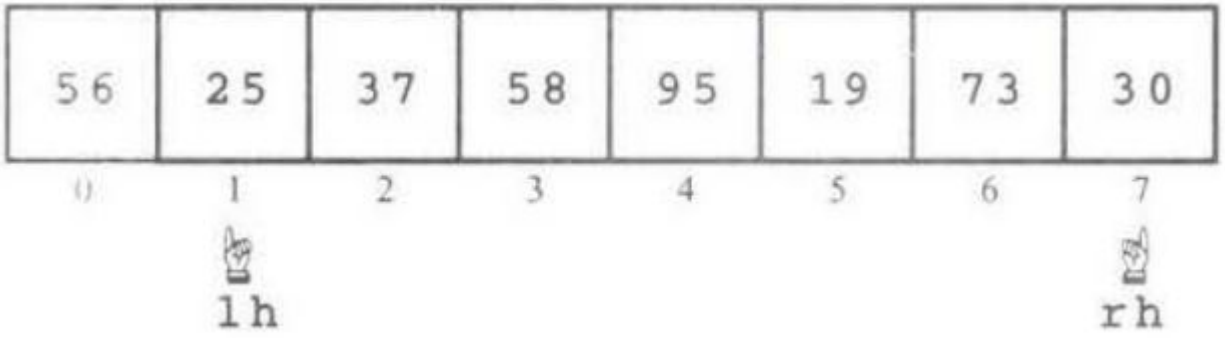
- Vamos criar 2 índices, **lh** e **rh**, para apontar para o primeiro e o último elemento do array (ignorando o pivô)



Quick Sort: particionamento do array

- Ande com o índice rh para a ESQUERDA, até que uma das seguinte situações ocorra:
  - Ele encontre com o índice lh
  - Ele encontre um elemento menor do que o pivô

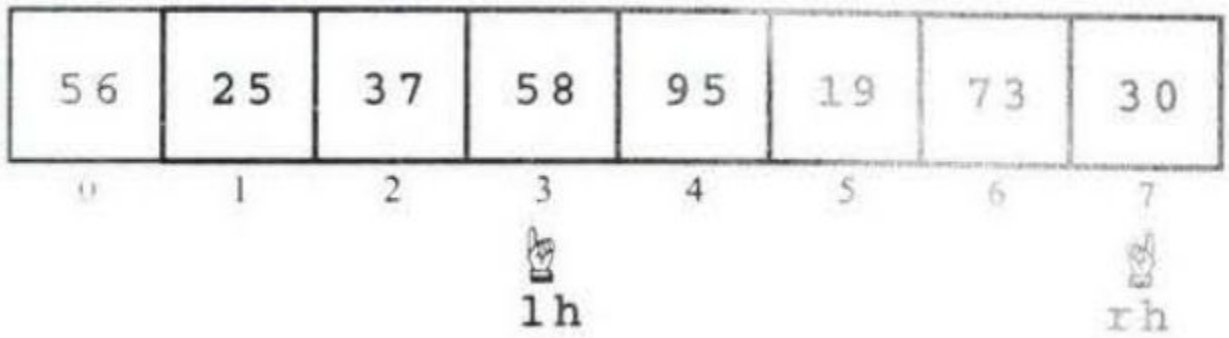
- Em nosso exemplo, o rh já está em um elemento menor do que o pivô, então ele pára na mesma posição:



Quick Sort: particionamento do array

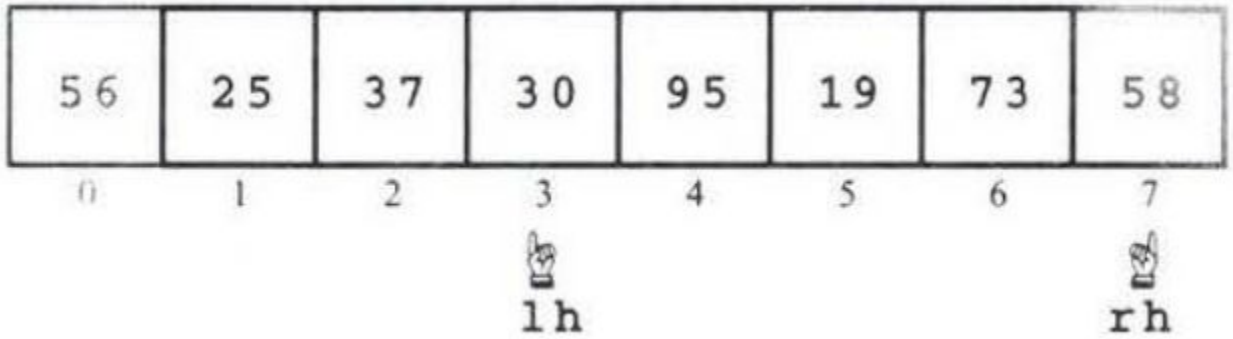
- Ande com o índice lh para a DIREITA, até que uma das seguinte situações ocorra:
  - Ele encontre com o índice rh
  - Ele encontre um elemento maior do que ou igual ao pivô

- Em nosso exemplo, o lh vai andar até a posição 3:



Quick Sort: particionamento do array

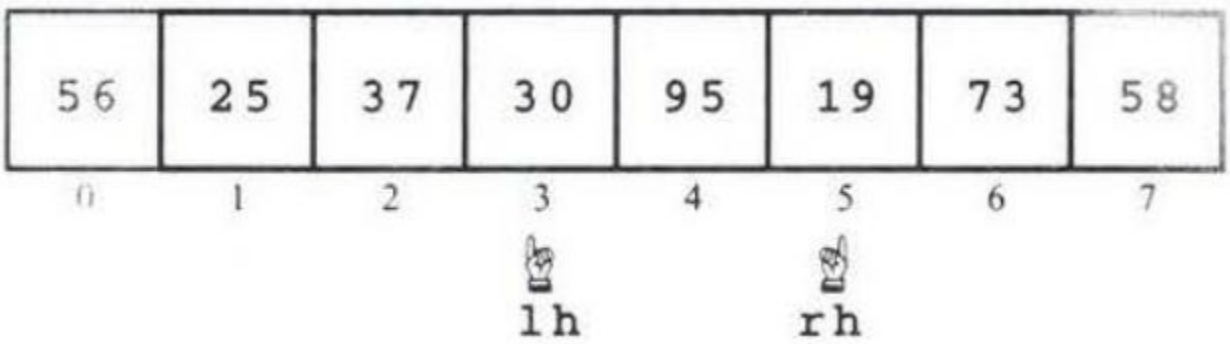
- Se os índices lh e rh NÃO estão na mesma posição, trocamos os elementos que eles apontam:



Quick Sort: particionamento do array

- Como os índices lh e rh não estão na mesma posição, começamos a mover novamente o índice rh para a ESQUERDA, até encontrar o índice lh ou encontrar um valor menor do que o pivô

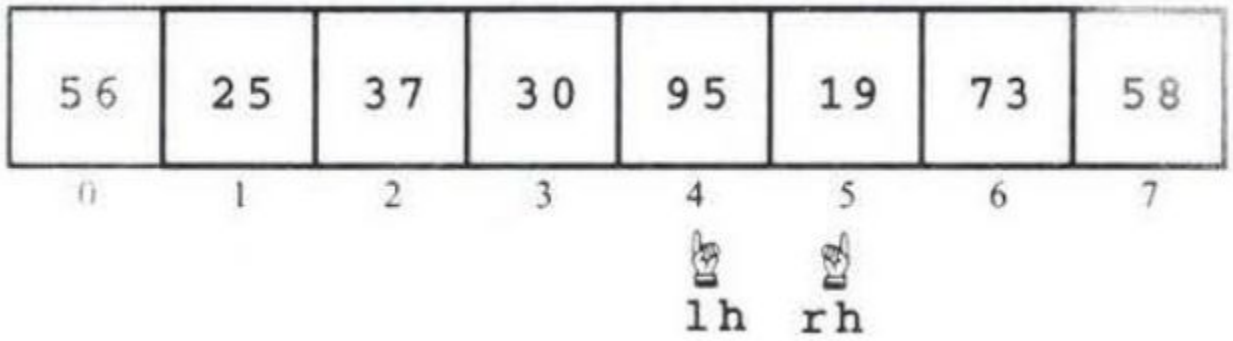
- Em nosso caso o índice rh irá parar na posição 5:



Quick Sort: particionamento do array

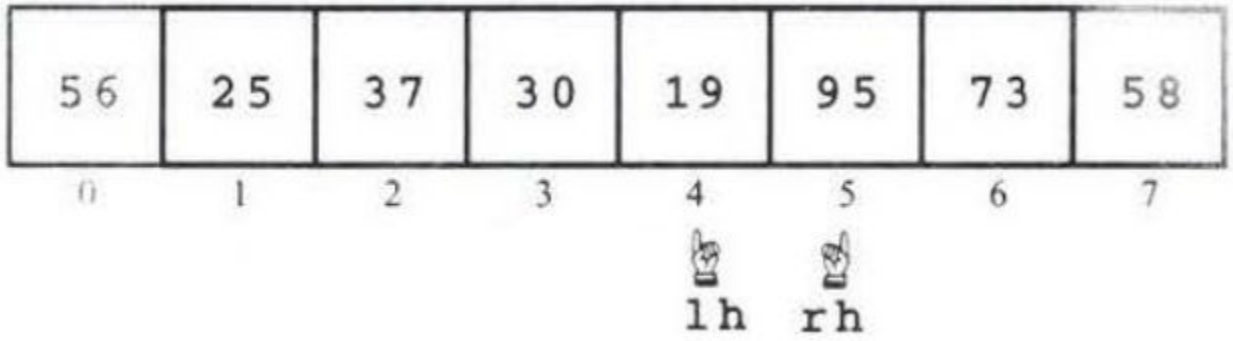
- Agora o índice lh vai se moer para a DIREITA até encontrar o índice rh ou encontrar um valor maior do que ou igual ao pivô

- Em nosso caso o índice lh irá parar na posição 4:



Quick Sort: particionamento do array

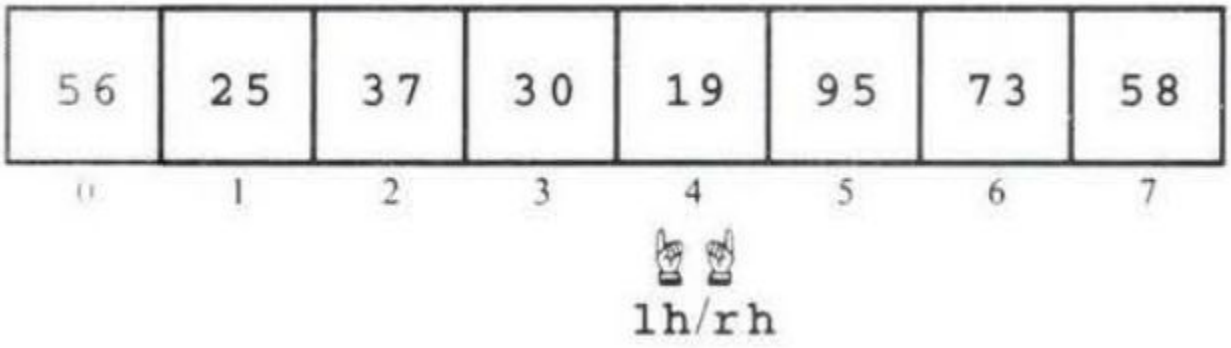
- Se os índices lh e rh NÃO estão na mesma posição, trocamos os elementos que eles apontam:



Quick Sort: particionamento do array

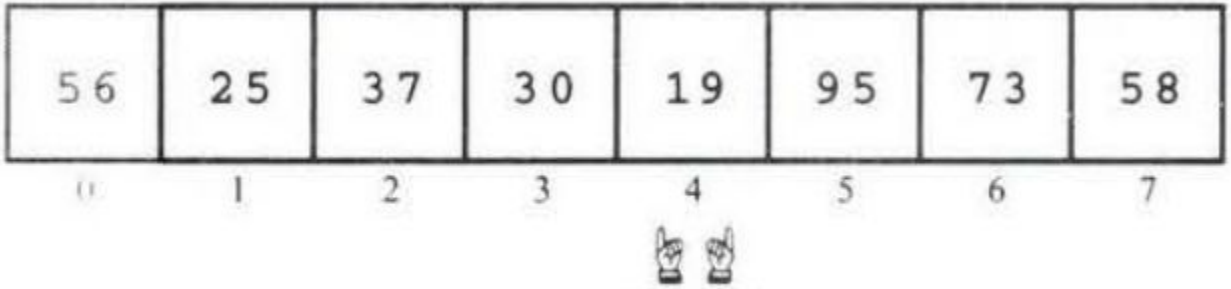
- Como os índices lh e rh não estão na mesma posição, começamos a mover novamente o índice rh para a ESQUERDA, até encontrar o índice lh ou encontrar um valor menor do que o pivô

- Em nosso caso o índice rh irá encontrar o índice lh:

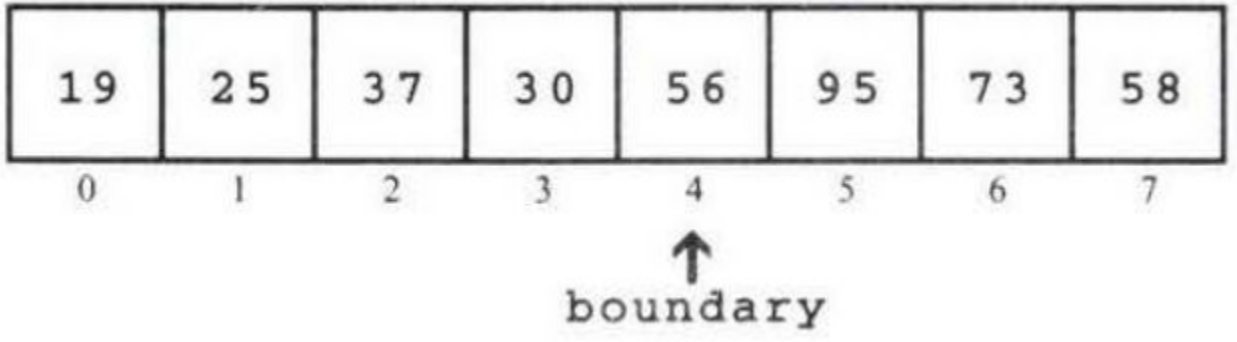


Quick Sort: particionamento do array

- Quando os índices se encontrarem, o elemento apontado por eles GERALMENTE será o menor valor mais à direita no array:



- Basta trocarmos esse valor com o pivô para terminarmos o particionamento, e o limite ficará nessa posição



Quick Sort: particionamento do array

- Se, no momento que os índices se encontrarem, o elemento não é menor do que o pivô, isso quer dizer que o pivô é o menor elemento em todo o array, e o limite ficará na posição 0.
  - Por sorte isso não ocorre com muita freqüência.

- Após participar o array, basta chamar recursivamente o quick sort nos subarrays à esquerda e à direita do limite (pivô).

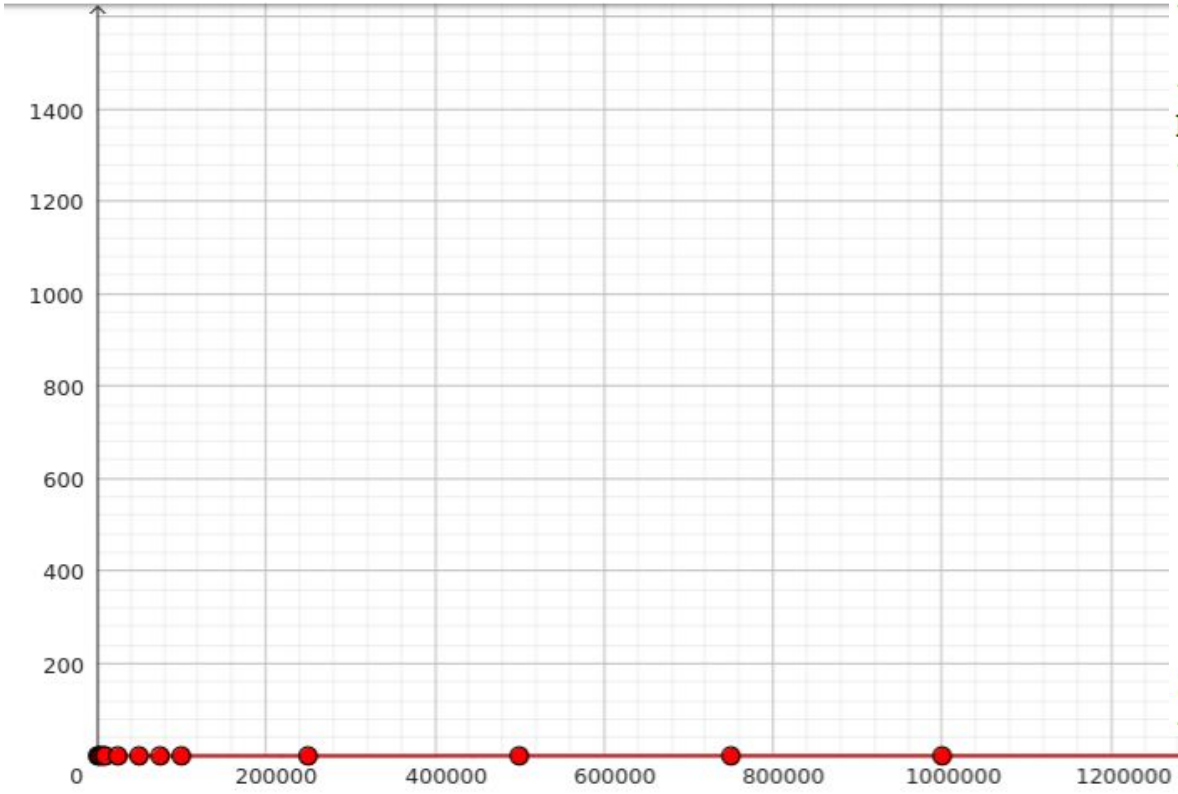
Algoritmos sofisticados de ordenação: Quick Sort

```
void quick_sort (int array[], int n)
{
    // Caso simples da recursão: um array com 1 elemento ou vazio
    if (n <= 1)
        return;

    // Particiona o array e determina o índice do limite:
    int limite = particionar(array, n);

    // Ordenar recursivamente:
    quick_sort(array, limite);
    quick_sort(array + limite + 1, n - limite - 1);
}
```

Algoritmos sofisticados de ordenação: Quick Sort

```
static int particionar (int array[], int n)
{
    // Pivô e ponteiros
    int pivo = array[0];
    int pe, pd;
    pe = 1;
    pd = n - 1;

    // Rearranja elementos menores e maiores do que o pivô:
    while (TRUE)
    {
        while (pe < pd && array[pd] >= pivo) pd--;
        while (pe < pd && array[pe] <  pivo) pe++;
        if (pe == pd) break;
        trocar(&array[pe], &array[pd]);
    }

    // Caso o pivô seja o menor elemento:
    if (array[pe] >= pivo) return 0;

    // Caso o pivô não seja o menor elemento:
    array[0] = array[pe];
    array[pe] = pivo;
    return pe;
}
```

Complexidade do Quick Sort (informal)



Quick Sort	
N	T (s)
1.000	0.000231743
2.500	0.001077546
5.000	0.002264351
7.500	0.003495930
10.000	0.004762044
25.000	0.006162882
50.000	0.007928435
75.000	0.009947799
100.000	0.012134895
250.000	0.029793580
500.000	0.061903133
750.000	0.094444008
1.000.000	0.129125696

Complexidade do Quick Sort (informal)

Empirical comparison of merge sort and Quicksort

N	Merge sort	Quicksort
10	0.54 msec	0.10 msec
20	1.17 msec	0.26 msec
40	2.54 msec	0.52 msec
100	6.90 msec	1.76 msec
200	14.84 msec	4.04 msec
400	31.25 msec	8.85 msec
1000	84.38 msec	26.04 msec
2000	179.17 msec	56.25 msec
4000	383.33 msec	129.17 msec
10,000	997.67 msec	341.67 msec

Complexidade do Quick Sort (informal)

- A complexidade do Quick Sort é **afetada pela escolha do pivô**! Se o pivô estiver próximo do valor mediano do array, o particionamento dividirá o array em 2 partes aproximadamente iguais e, nesse caso, a complexidade será:

  $O(N \log_2 N)$, no melhor caso e no caso médio

- Se a pivô for um valor muito alto ou muito baixo, um dos subarrays será muito maior do que o outro, invalidando a estratégia dividir para conquistar. Um array completamente ordenado é o pior caso para o Quick Sort:

  $O(N^2)$

- Melhorar a probabilidade de escolher um bom pivô:
  - aleatório (é improvável que o algoritmo escolha pivôs ruins repetidamente)
  - mediana de 3 ou 5 valores