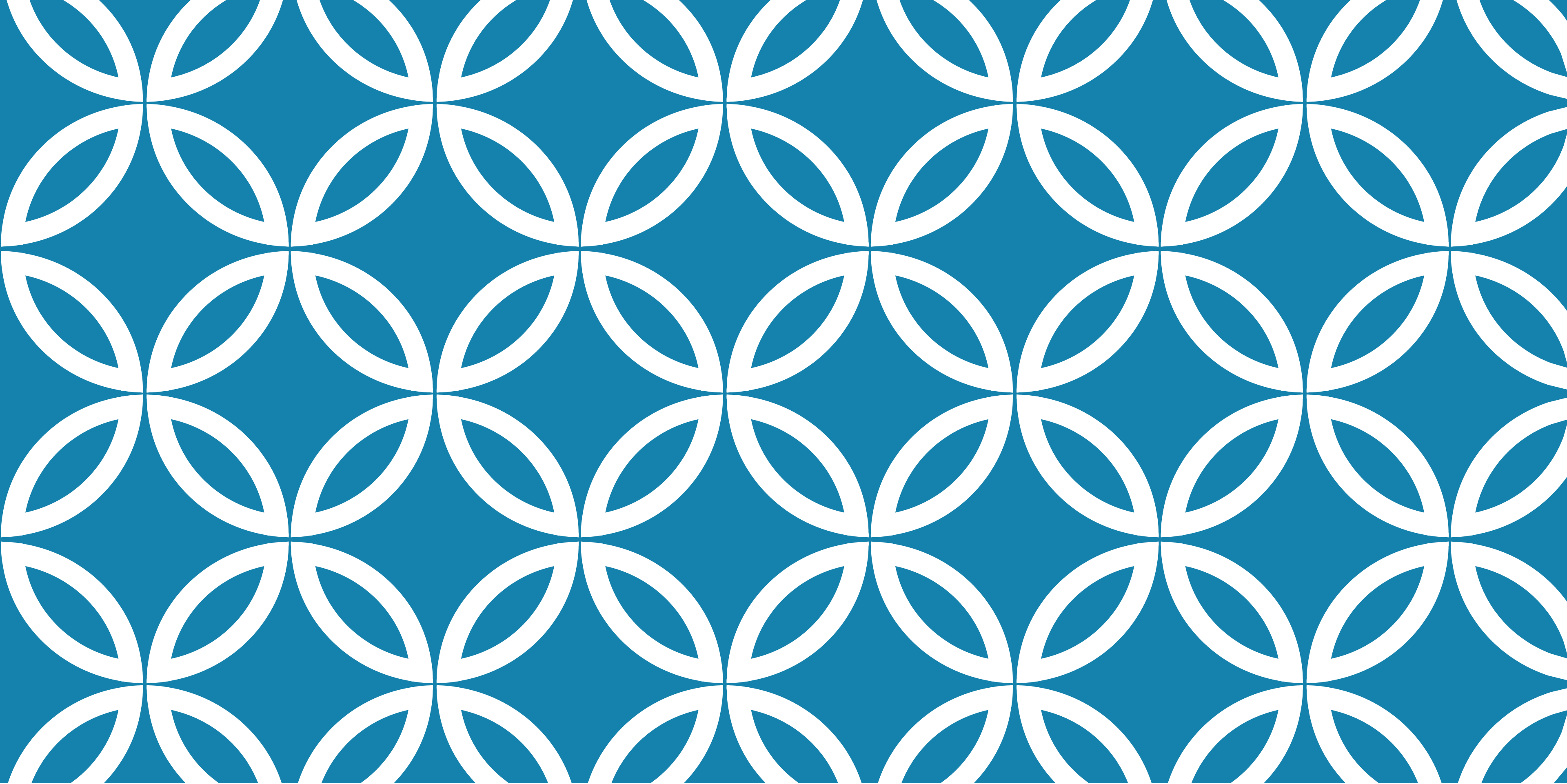


# INTRODUÇÃO

Prof. André Vignatti

Análise de Algoritmos

# COMO SABER SE O ALGORITMO É BOM?

**intuitivamente:**

- testar se o algoritmo funciona
- ver se é rápido

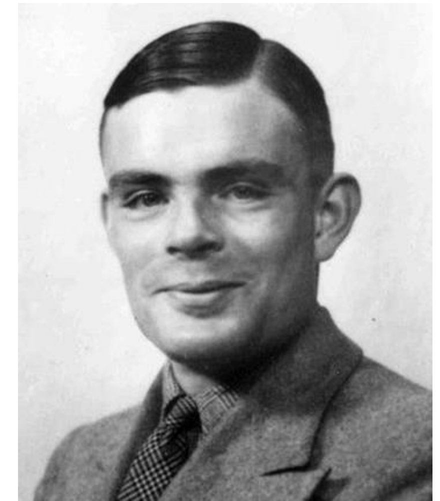
como saber se o algoritmo funciona?

- implementação cuidadosa
- testar várias entradas

como saber se ele é rápido?

- testar entradas grandes e variadas
- ver se executa bem

*"É conveniente ter uma medida da* ***quantidade de trabalho*** *envolvida em um processo de computação, mesmo que seja muito bruta ... Podemos, por exemplo, contar o número de adições, subtrações, multiplicações, divisões, gravações de números e extrações de figuras de tabelas."* - **Alan Turing, 1947**



## ROUNDING-OFF ERRORS IN MATRIX PROCESSES

*By* A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

# PERGUNTA

qual é o tempo de execução deste algoritmo?

```
boolean  algoritmo  (char[] v) {
    int n = v.length();
    for (int i = 0; i < n; i++)
        if (v[i] != v[n-i-1])
            return false;
    return true;
}
```

$O(1)$

$O(\log n)$

$O(n^{1/2})$

$O(n)$

$O(n \log n)$

$O(n^2)$

$O(n^2 \log n)$

$O(2^n)$

# ANÁLISE DE ALGORITMOS

Nesta altura do campeonato, você já deve ter visto ao menos uma análise de algoritmo (mesmo que superficialmente)

Como *provavelmente* você viu análise de algoritmo?

- **CASO 1**: há dois laços aninhados, indo até $n$. Portanto, o algoritmo é $O(n^2)$

- **CASO 2**: contar uma operação específica do algoritmo (comparações, trocas, ...)

# SELECTION SORT

ideia: seleciona o 1° menor elemento, troca com 1ª posição, seleciona o 2° menor elemento, troca com a 2ª posição, ...

vetor original:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 15 | 12 | 27 | 23 | 7 | 2 | 0 | 18 | 19 | 21 |

execução:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 12 | 27 | 23 | 7 | 2 | **15** | 18 | 19 | 21 |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | **2** | 27 | 23 | 7 | **12** | 15 | 18 | 19 | 21 |
| 0 | 2 | **7** | 23 | **27** | 12 | 15 | 18 | 19 | 21 |
| 0 | 2 | 7 | **12** | 27 | **23** | 15 | 18 | 19 | 21 |
| 0 | 2 | 7 | 12 | **15** | 23 | **27** | 18 | 19 | 21 |
| 0 | 2 | 7 | 12 | 15 | **18** | 27 | **23** | 19 | 21 |
| 0 | 2 | 7 | 12 | 15 | 18 | **19** | 23 | **27** | 21 |
| 0 | 2 | 7 | 12 | 15 | 18 | 19 | **21** | 27 | **23** |
| 0 | 2 | 7 | 12 | 15 | 18 | 19 | 21 | **23** | **27** |

# SELECTION SORT

```
SelectionSort(A[1...n])
    para i ← 1 até n - 1
        menor ← i
        para j ← i + 1 até n
            se A[menor] > A[j]
                menor ← j
        Troca(A, i, menor)
```

```
Troca(A, i, j)
    temp ← A[i]
    A[i] ← A[j]
    A[j] ← temp
```

# ANÁLISE – SELECTION SORT

exemplo de análise contando as *comparações*

```
SelectionSort(A[1...n])
    para i ← 1 até n - 1
        menor ← i
        para j ← i + 1 até n
            se A[menor] > A[j]
                menor ← j
        Troca(A, i, menor)
```

quando $i = 1$:

- $j$ vai de $2$ até $n$
- `if` é executado $n - 1$ vezes

quando $i = 2$:

- $j$ vai de $3$ até $n$
- `if` é executado $n - 2$ vezes

quando $i = 3$:

- $j$ vai de $4$ até $n$
- `if` é executado $n - 3$ vezes

$\vdots$

quando $i = n - 2$:

- $j$ vai de $n - 1$ até $n$
- `if` é executado $2$ vezes

quando $i = n - 1$:

- $j$ vai de $n$ até $n$
- `if` é executado $1$ vez

# ANÁLISE – SELECTION SORT

contando as comparações do selection sort:

quando $i = 1$:

- `if` é executado $n - 1$ vezes

quando $i = 2$:

- `if` é executado $n - 2$ vezes

quando $i = 3$:

- `if` é executado $n - 3$ vezes

$\vdots$

quando $i = n - 2$:

- `if` é executado $2$ vezes

quando $i = n - 1$:

- `if` é executado $1$ vez

$$S = (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$$

# ANÁLISE – SELECTION SORT

contando as comparações do selection sort:

$$S = (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$$

$$= \sum_{i=1}^{n-1} i$$

Eu tenho uma fórmula pra isso!!!

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

$$\approx \frac{n^2}{2} \approx \boxed{n^2}$$

Carl Friedrich Gauss
(1777 - 1855)

# OBJETIVO: COMPARAR ALGORITMOS

para a ordenação:

algoritmos "ingênuos": $n^2$

algoritmos "espertos": $n \log_2 n$

| $n$ | $\boldsymbol{n^2}$ | $n \log_2 n$ |
|---|---|---|
| 10 | 100 | 320 |
| 100 | 10000 | 650 |
| 1000 | 1 milhão | 10000 |
| 1 milhão | 1 trilhão | 20 milhões |
| 1 bilhão | $10^9$ bilhões | 30 bilhões |

# ABRE PARÊNTESES: VELOCIDADE

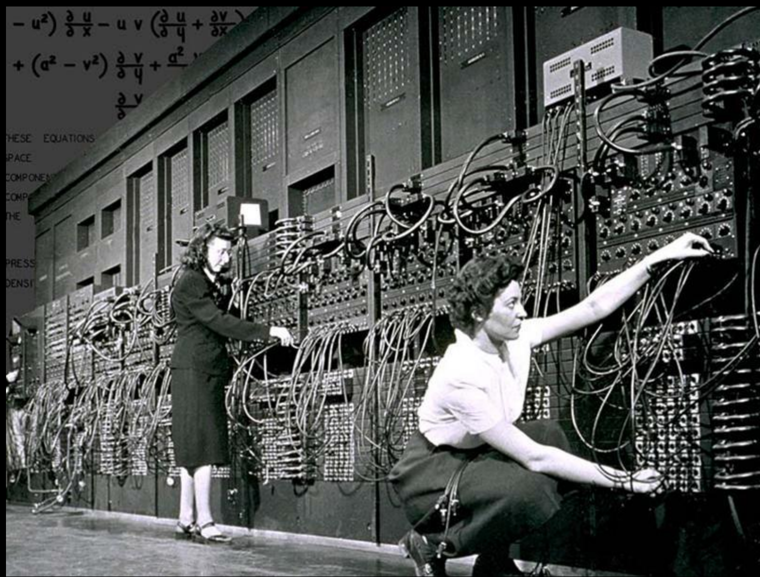
medida popular, porém **imprecisa**:

**núm. de instruções/segundo**

atualmente: alguns bilhões de instruções/segundo

ordenação com 1 bilhão instr/seg

| $n$ | ingênuo | esperto |
|---|---|---|
| 10 | - | - |
| 100 | 0,00001 s | - |
| 1000 | 0,001 s | 0,00001s |
| 1 milhão | 16,6 min | 0,2 s |
| 1 bilhão | 31,7 anos | 30 s |

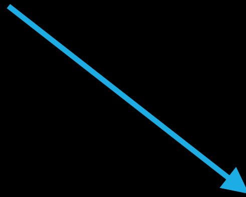



- ENIAC: ≈ 5 mil instr/seg

Intel I9: ≈ 5 bilhões instr/seg

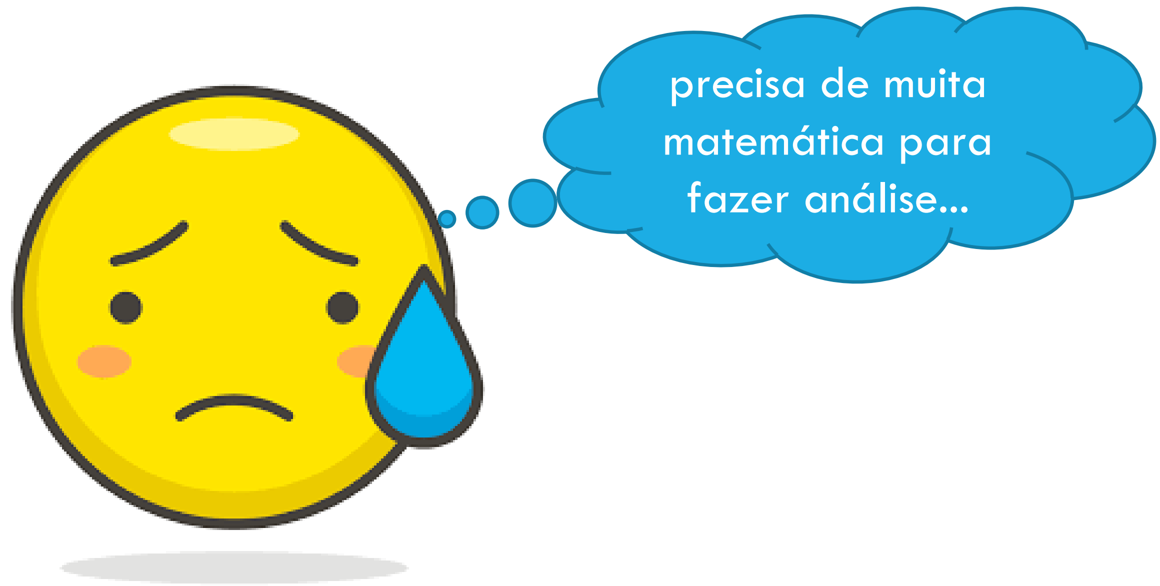
Intel I9 é 1 milhão de vezes mais rápido que ENIAC

$n$ grande: **algoritmo bom no ENIAC é melhor que um algoritmo ruim no I9**



na ordenação: $n$ grande é ≈ $\mathbf{20}$ milhões

fonte en.wikipedia.org/wiki/Clock_rate




vamos tentar uma forma alternativa da análise...

exemplo: algoritmo de ordenação

executou para vetor de tam n=100, levou 1s

só isso não traz muita informação...

exemplo: algoritmo de ordenação

executou para vetor de tam n=100, levou 1s

só isso não traz muita informação...

vou executar para outros tamanhos:

executou para vetor de tam n=200, levou 2s

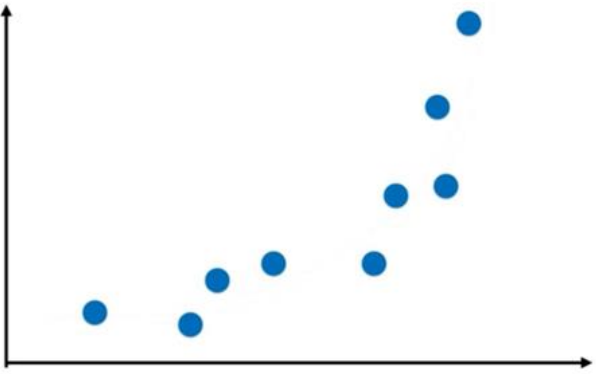
executou para vetor de tam n=300, levou 4s

executou para vetor de tam n=400, levou 7s

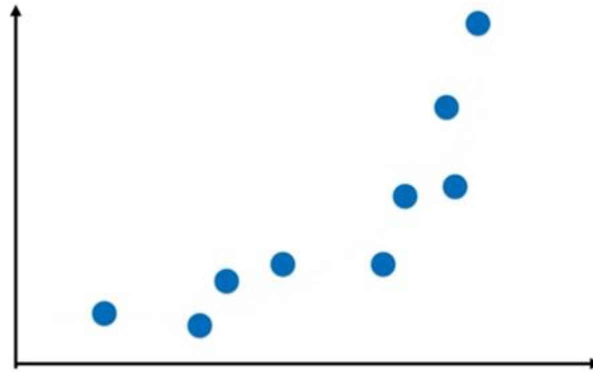
executou para vetor de tam n=500, levou 17s

executou para vetor de tam n=600, levou 30s

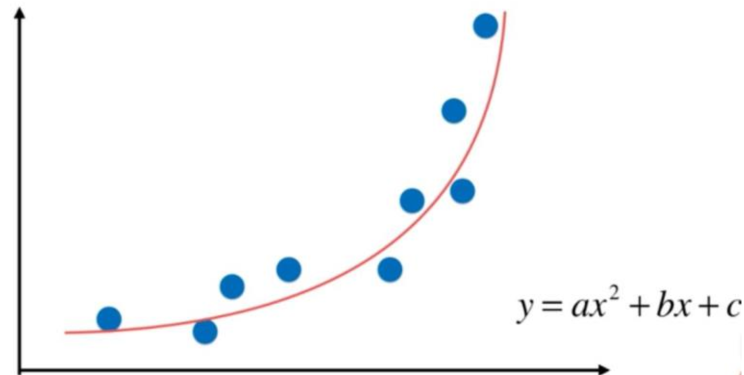
plota o gráfico:



plota o gráfico:

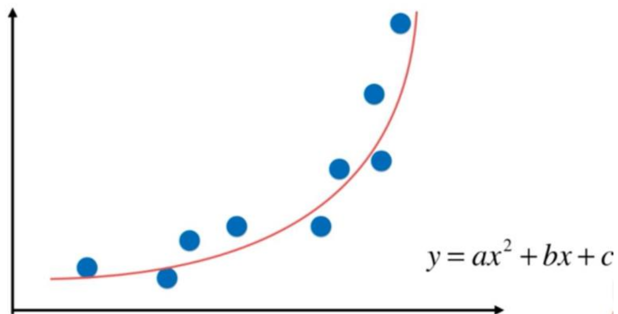


faz uma regressão:

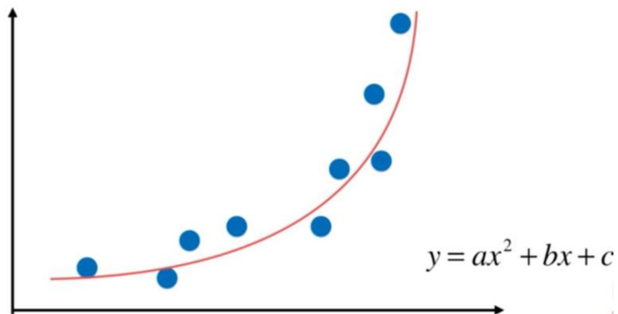



encontrou função quadrática! **Portanto, é um algoritmo de ordenação** $\approx \boldsymbol{n^2}$

Isso é tudo?

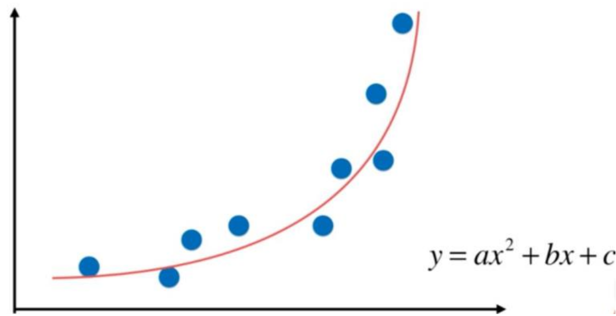



Isso é tudo?




Isso é tudo?

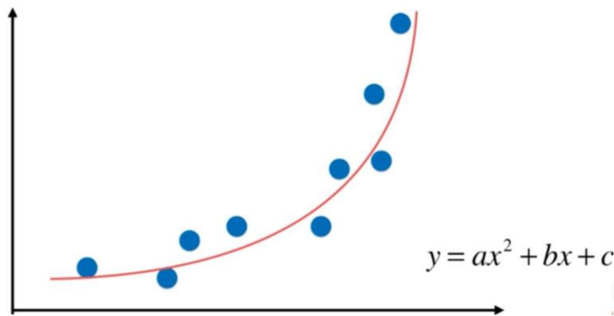
Há muitas outras coisas a serem estudadas:




Isso é tudo?

Há muitas outras coisas a serem estudadas:

Para mesmo $n$, há diferença no tempo de execução de entradas diferentes?

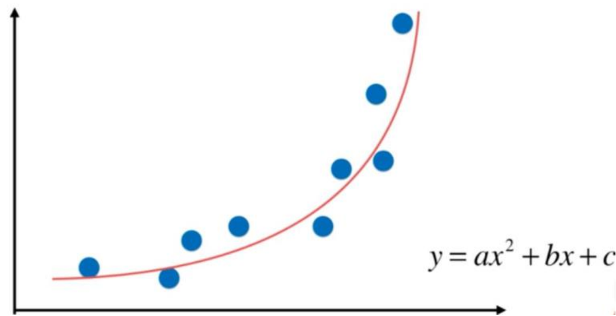



Isso é tudo?

Há muitas outras coisas a serem estudadas:

Para mesmo $n$, há diferença no tempo de execução de entradas diferentes?

Qual entrada faz o algoritmo ter a pior execução?



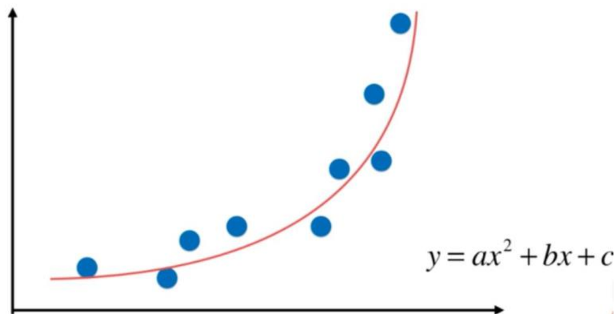

## Isso é tudo?

Há muitas outras coisas a serem estudadas:

Para mesmo $n$, há diferença no tempo de execução de entradas diferentes?

Qual entrada faz o algoritmo ter a pior execução?

Qual entrada faz o algoritmo ter a melhor execução?




## Isso é tudo?

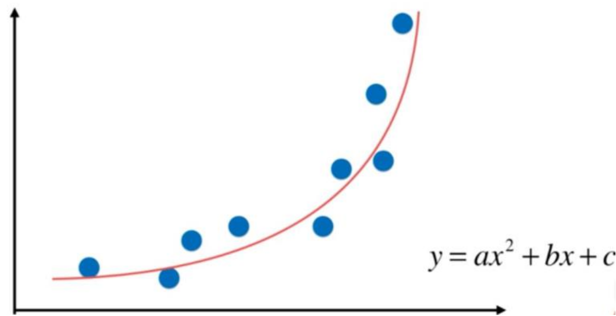
Há muitas outras coisas a serem estudadas:

Para mesmo $n$, há diferença no tempo de execução de entradas diferentes?

Qual entrada faz o algoritmo ter a pior execução?

Qual entrada faz o algoritmo ter a melhor execução?

As entradas ruins (boas) são raras ou comuns?




## Isso é tudo?

Há muitas outras coisas a serem estudadas:

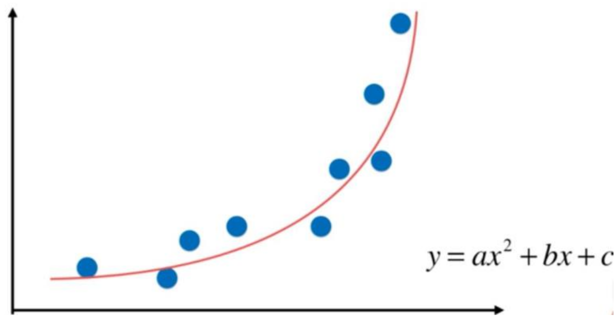
Para mesmo $n$, há diferença no tempo de execução de entradas diferentes?

Qual entrada faz o algoritmo ter a pior execução?

Qual entrada faz o algoritmo ter a melhor execução?

As entradas ruins (boas) são raras ou comuns?

Qual o trecho do algoritmo faz as entrada ruins executarem mal?




## Isso é tudo?

Há muitas outras coisas a serem estudadas:

Para mesmo $n$, há diferença no tempo de execução de entradas diferentes?

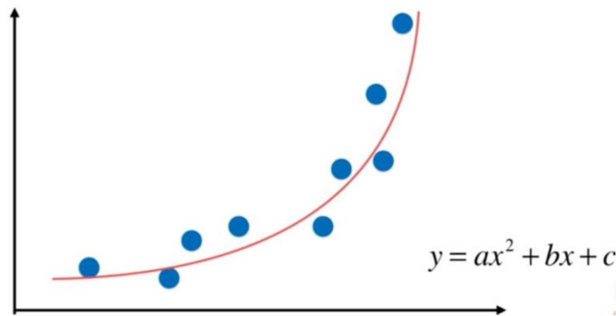
Qual entrada faz o algoritmo ter a pior execução?

Qual entrada faz o algoritmo ter a melhor execução?

As entradas ruins (boas) são raras ou comuns?

Qual o trecho do algoritmo faz as entrada ruins executarem mal?

Se eu alterar esse trecho do algoritmo, quão melhor ele executará?




# Isso é tudo?

Há muitas outras coisas a serem estudadas:

Para mesmo $n$, há diferença no tempo de execução de entradas diferentes?

Qual entrada faz o algoritmo ter a pior execução?

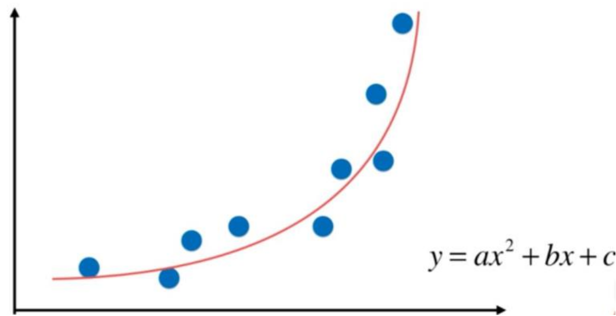
Qual entrada faz o algoritmo ter a melhor execução?

As entradas ruins (boas) são raras ou comuns?

Qual o trecho do algoritmo faz as entrada ruins executarem mal?

Se eu alterar esse trecho do algoritmo, quão melhor ele executará?

Existe alguma entrada que o algoritmo devolve resposta errada?




# Isso é tudo?

Há muitas outras coisas a serem estudadas:

Para mesmo $n$, há diferença no tempo de execução de entradas diferentes?

Qual entrada faz o algoritmo ter a pior execução?

Qual entrada faz o algoritmo ter a melhor execução?

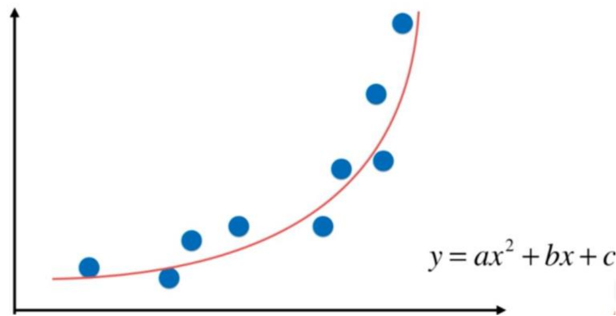
As entradas ruins (boas) são raras ou comuns?

Qual o trecho do algoritmo faz as entrada ruins executarem mal?

Se eu alterar esse trecho do algoritmo, quão melhor ele executará?

Existe alguma entrada que o algoritmo devolve resposta errada?

Qual é o ponto fraco do problema? Como bolar um algoritmo para isso?




Isso é tudo?

Há muitas outras coisas a serem estudadas:

Para mesmo $n$, há diferença no tempo de execução de entradas diferentes?

Qual entrada faz o algoritmo ter a pior execução?

Qual entrada faz o algoritmo ter a melhor execução?

As entradas ruins (boas) são raras ou comuns?

Qual o trecho do algoritmo faz as entrada ruins executarem mal?

Se eu alterar esse trecho do algoritmo, quão melhor ele executará?

Existe alguma entrada que o algoritmo devolve resposta errada?

Qual é o ponto fraco do problema? Como bolar um algoritmo para isso?

**isso tudo** é Análise de Algoritmos

# PROBLEMA COMPUTACIONAL

um **problema computacional** é um problema formulado em termos de

dado . . .

obtenha . . .

ou,

entrada: . . .

saída: . . .

ou ainda,

instância: . . .

resposta: . . .

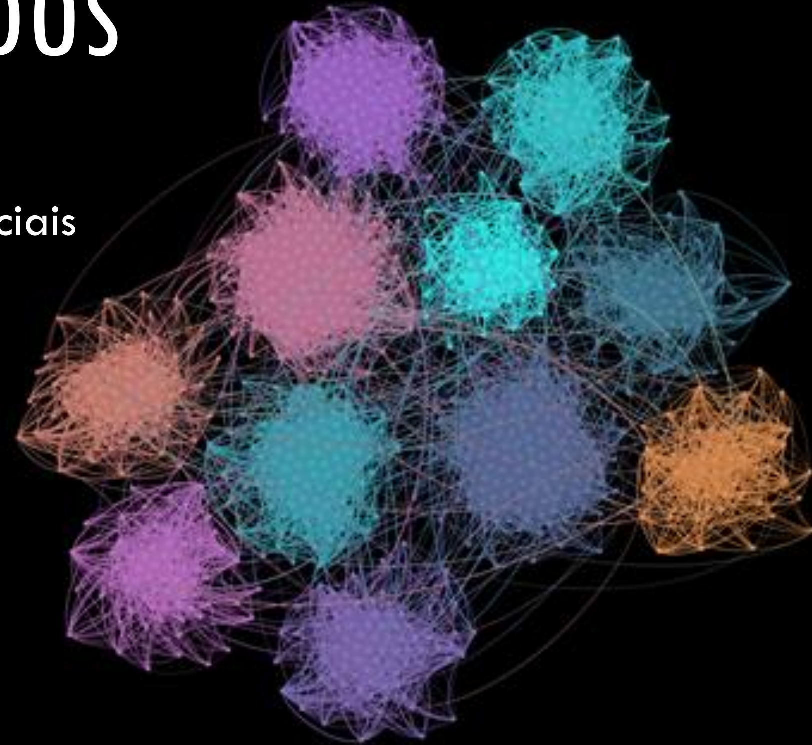
# PROBLEMAS MAL-DEFINIDOS

**problema:** detectar comunidades em redes sociais

- o que é uma comunidade?

**solução:** definir precisamente!

1. método do corte mínimo
2. método de agrupamento hierárquico
3. algoritmo de Girvan–Newman
4. maximização de modularidade
5. baseado em cliques
6. ...



e quem escolhe a definição precisa?

# PROBLEMAS MAL-DEFINIDOS

alguns casos são **ainda mais problemáticos**...

**problema:** classificar uma imagem como cão ou gato

- o que é uma imagem de cão?
- o que é uma imagem de gato?

**solução:** definições “precisas” (entre aspas)

1. redes neurais
2. classificadores lineares
3. ...

ainda não se sabe exatamente porque eles funcionam!!!

# ALGORITMOS

a **solução** de um problema computacional é chamado de **algoritmo**

um algoritmo é:

- a *descrição* de uma *computação* (passo-a-passo)
- para cada instância do problema, resulta numa resposta

# ALGORITMOS - DESCRIÇÃO

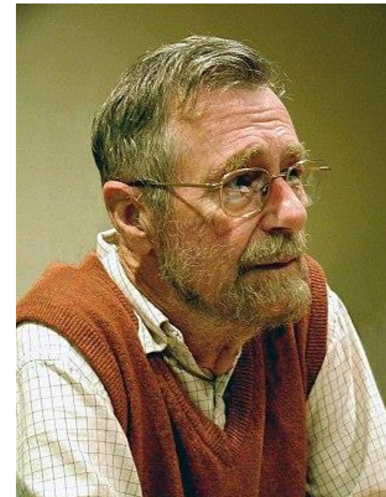
vamos descrever algoritmos usando **pseudo-código**

- parece código, mas não é!
- é mais simples (omite detalhes específicos usados para o computador entender (declarar variáveis, delimitadores, ...)
- a maioria dos livros de algoritmos usa pseudo-código

## note que não é preciso um computador para executar a computação descrita pelo algoritmo!

uma pessoa, um animal treinado, ou um dispositivo mecânico movido a querosene são capazes de executá-lo.

*"Ciência da computação está relacionada com o computador assim como a Astronomia com o telescópio, a Biologia com o microscópio, ou a Química com os tubos de ensaio. A Ciência não estuda ferramentas, mas o que fazemos e o que descobrimos com elas."* - **Edsger Dijkstra**



Edsger Dijkstra
(1930 – 2002)

# COMPUTAÇÃO $\neq$ TECNOLOGIA

já vimos que:

- computador não é necessário para fazer computação
- linguagens de programação não são necessárias para descrever algoritmos

nossa abordagem é mais "purista",
não dependente da tecnologia

essa abordagem purista é o **padrão** de abordagem na área de algoritmos

# CONHEÇA O ADVERSÁRIO

um mundo cruel:

- entrada **diabolicamente planejada**
- abordagem de **pior caso**
- seremos **honestos**: se algoritmo é bom, nenhuma entrada tem mal desempenho

**abordagens alternativas:**

- **melhor caso**
- **caso médio**

} ignora o adversário

- **caso esperado** } engana o adversário

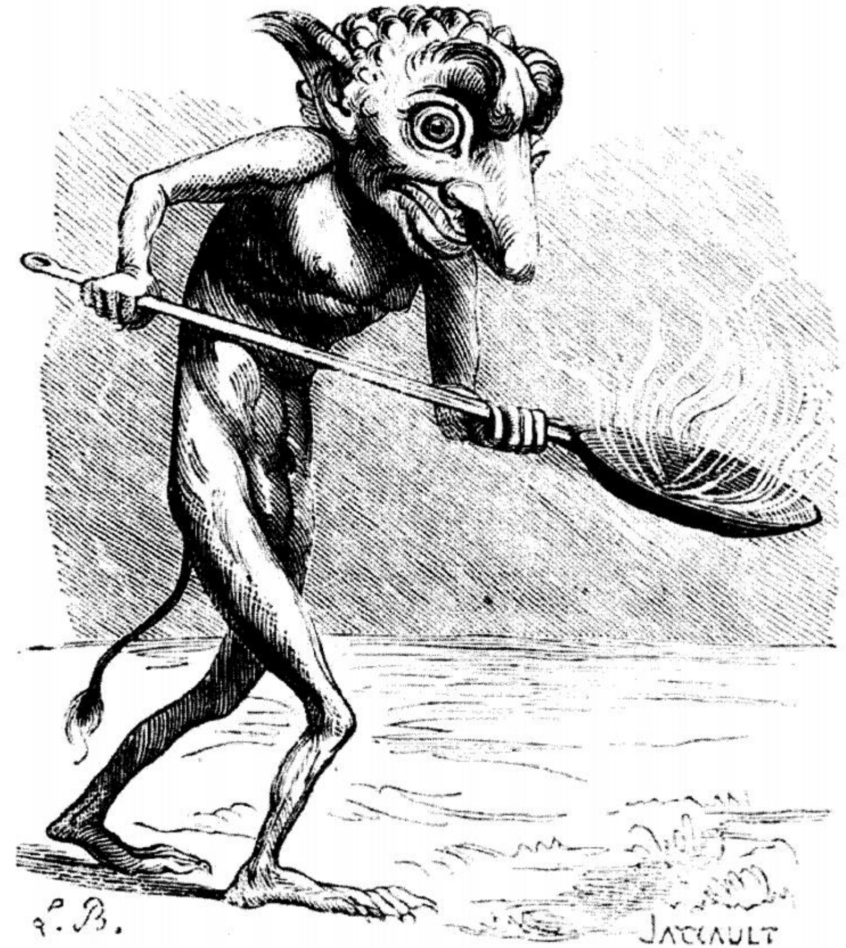


FIGURE 2.1: The adversary bringing us a really stinky instance.

# COMO ANALISAR: TAMANHO IMPORTA

- buscar em bilhões de palavras > buscar em 2 palavras
- análise depende do **tamanho $n$ da entrada**
- tamanho $n$ é
  - geralmente: #bits da codificação binária da entrada
  - às vezes: explicitamente definido como sendo outra coisa

**GRANDE IDEIA # 1: ANÁLISE É UMA FUNÇÃO EM $n$**

# COMO ANALISAR: VALORES IMPORTAM

- diferentes entradas tem diferentes valores!
- para entradas de (mesmo) tamanho $n$:
  - algumas levam 2 passos,
  - algumas levam bilhões de passos

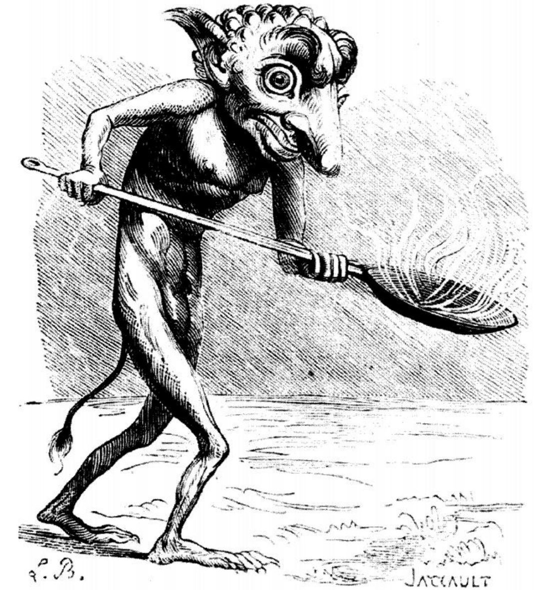


FIGURE 2.1: The adversary bringing us a really stinky instance.

**GRANDE IDEIA # 2: ANÁLISE PELA PIOR ENTRADA DE TAMANHO $n$**

# COMO ANALISAR: MODELO IMPORTA

exemplo: é palíndromo?

- MT com 1 fita $\sim n^2$ **passos**
  - cabeça de leitura fica alternando entre os extremos
- MT com 2 fitas $\sim n$ **passos**
  - copia a string na 2ª fita, então não precisa ficar alternando entre os extremos

C, Java, Python, MT 1 fita, MT 2 fitas ...

- **computabilidade**: são iguais
- **complexidade**: são diferentes

# COMO ANALISAR: MODELO IMPORTA

porque o **modelo** importa?

```
boolean palindromo (char[] v) {
    int n = v.length();
    for (int i = 0; i < n; i++)
        if (v[i] != v[n-i-1])
            return false;
    return true;
}
```

no final,

$$\boldsymbol{i} = \boldsymbol{n}$$

#bits para guardar $\boldsymbol{i}$?

$$\sim \log_2 \boldsymbol{n}$$

# COMO ANALISAR: MODELO IMPORTA

porque o **modelo** importa?

```
boolean palindromo (char[] v) {
    int n = v.length();
    for (int i = 0; i < n; i++)
        if (v[i] != v[n-i-1])
            return false;
    return true;
}
```

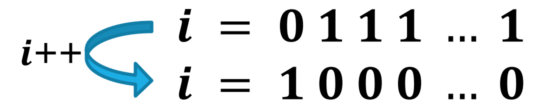
no final,

$$i = n$$

#bits para guardar $i$?

$$\sim \log_2 n$$

se $n$ é potência de **2:**



$$\sim \log_2 n$$

passos?

# COMO ANALISAR: MODELO IMPORTA

porque o **modelo** importa?

```
boolean palindromo (char[] v) {
    int n = v.length();
    for (int i = 0; i < n; i++)
        if (v[i] != v[n-i-1])
            return false;
    return true;
}
```

inicialmente, acessa

$$v[0] \text{ e } v[n-1]$$

é necessário $n$ passos para ir de $v[0]$ até $v[n-1]$?

# COMO ANALISAR: MODELO IMPORTA

porque o **modelo** importa?

```
boolean palindromo (char[] v) {
    int n = v.length();
    for (int i = 0; i < n; i++)
        if (v[i] != v[n-i-1])
            return false;
    return true;
}
```

inicialmente, acessa

$$v[0] \text{ e } v[n-1]$$

é necessário $n$ passos para ir de $v[0]$ até $v[n-1]$?

**GRANDE IDEIA # 3: MODELO IMPORTA PARA O TEMPO DE EXECUÇÃO**

# O MELHOR MODELO COMPUTACIONAL

**P**: “se o modelo importa, qual é o **melhor** modelo?”

**R1**: depende

**R2**: não importa

# O MELHOR MODELO COMPUTACIONAL

**P**: “se o modelo importa, qual é o **melhor** modelo?”

**R1**: depende

**R2**: não importa

## GRANDE IDEIA # 4: MODELOS DETERMINÍSTICOS SÃO POLINOMIALMENTE EQUIVALENTES

- esteja ciente do modelo que está sendo usado!
- esteja ciente do que constitui um passo no modelo!

# RESUMINDO: COMO ANALISAR?

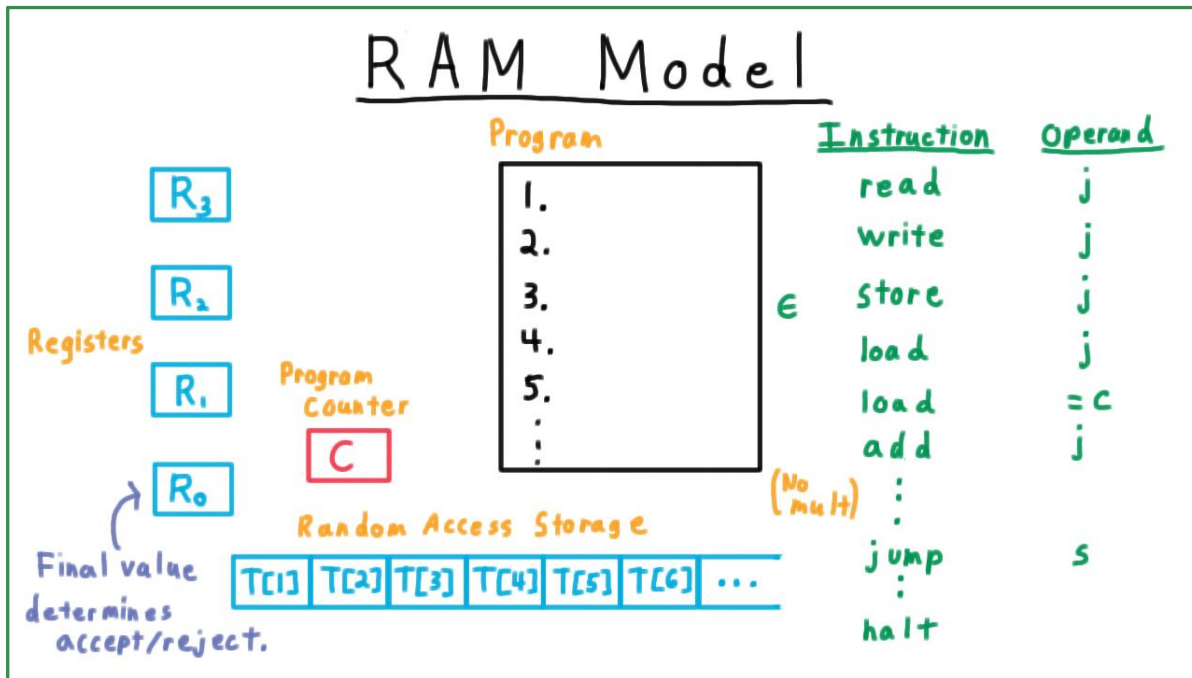
e se o vetor é pequeno/grande? e se eu tiver sorte? o que é considerado um passo?

- **tamanho importa!**
- **valores importam!**
- **modelo importa!**

**qual modelo usar?**

# MODELO RANDOM ACCESS MACHINE (RAM)

boa combinação de simplicidade/realidade




leva 1 passo (constante):

- operações aritméticas +, - , *, /
- operações lógicas ==, !=, >, <
- atribuições: x = y
- saltos condicionais: if... then... else
- acesso à memória: v[n]

**na verdade:** levam 1 passo (constante) se os números são limitados por $poli(n)$