# SCC5900 Projeto de Algoritmos

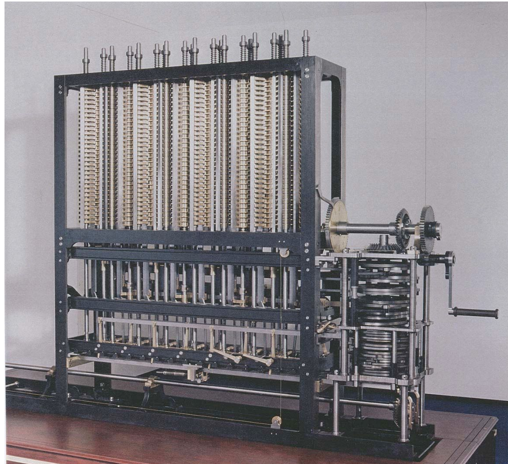
Complexidade de Algoritmos

Joao Batista
ICMC - USP

# Por que se preocupar com complexidade?

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? ”

Charles Babbage (1864)



Quantas vezes teremos que girar a manivela?

# Uma primeira tentativa em se definir **eficiência**

- Definição 1: um algoritmo é eficiente se, quando implementado, roda rapidamente para instâncias reais de entradas.

    - Você acha que esta definição é boa o suficiente?
    - Se não acha, destaque alguns problemas que ela apresenta….

# Uma primeira tentativa em se definir **eficiência**

- Onde este algoritmo rodou?
  - o hardware é ultra rápido ou muito lento?
- O quão bem este algoritmo foi implementado
  - será que a estrutura de dados escolhida foi adequada?
  - que abordagem foi usada na implementação?
- O que de fato significa uma instância real de entrada
  - se for uma instância que nada exige do seu código ???
- Como seu algoritmo escala à medida que o tamanho as instâncias de entrada crescem?

# Para boa definição para **eficiência** requer

- Independência da plataforma
- Independência das instâncias de entrada
- Um valor preditivo do comportamento do seu algoritmo com relação ao crescimento de suas entradas.
- Em resumo, é preciso uma abordagem um pouco mais matemática para resolver esta questão.

# O que é um algoritmo Eficiente?

- Além de se preocupar se um programa é ou não **rápido**.
- Alguns podem se preocupar com a quantidade de memória requerida para a execução
- Como afinal, medir a eficiência de um algoritmo? Qual o principal quesito
- Obs:  Existem problemas para os quais não se conhece nenhum algoritmo eficiente para obter a solução: *NP*-Completos > não trataremos disso aqui

# Uma segunda definição para **eficiência**

- Definição 2: Um algoritmo é eficiente se tem, qualitativamente, melhor desempenho, **no pior caso**, se comparado à qq solução força bruta em nível analítico.

- O que vc acha dessa definição?
    - É correto ou é muito draconiano aferir a eficiência do algoritmo sempre pelo pior caso?

# Uma segunda definição para **eficiência**

- Análise pelo pior caso pode ser muito ‘injusta’:
  - meu código é muito bom para a maioria dos casos e só fica lento para poucas instâncias.
- No entanto
  - é um bom parâmetro, quando ‘provado’ matematicamente. Encontrar uma alternativa para a análise do caso médio (para um conjunto de instâncias tomadas aleatoriamente) pode ser muito complicado. Mesmo porque é difícil definir o que seria um conjunto médio de entradas para um algoritmo!
- Mas o que dizer a respeito de uma comparação com a força bruta?
  - Voltemos ao exemplo do Casamento Estável.
  - Temos n! casamentos perfeitos possíveis. Podemos gerar todos eles e verificar qual(is) é/são estável(is).
  - No entanto, vimos que basta um tempo proporcional a N para achar a nossa solução.
  - Chegamos a esta conclusão em nível analítico. Não foi preciso implementar a força bruta.

# Uma segunda definição para **eficiência**

- Ok. Esta segunda definição é boa, mas ainda há algo vago acerca dela. Dizer que um algoritmo é eficiente porque é **qualitativamente melhor** não é o ideal.

# Uma terceira (e definitiva) definição para **eficiência**

- Def. 3: Um algoritmo é eficiente se ele tem tempo de execução polinomial.
  - $n$, $n \log n$, $n^2$, $n^3$.

# Uma terceira (e definitiva) definição para **eficiência**

- Espaços de busca para problemas combinatoriais tendem a crescer exponencialmente.
- Queremos algoritmos com escala de crescimento melhor.
  - Se N cresce por um fator constante (digamos 2), o algoritmo deve rodar mais lentamente por um fator constante C
- Matematicamente, podemos formular este comportamento de escala como:
  - Sejam c > o e d > 0 e para toda entrada N o tempo de execução de um algoritmo é limitado superiormente à $cN^d$ **passos de primitivas computacionais*.**
- Dizemos que o tempo de execução é proporcional à $N^d$

primitiva: uma instrução assembly ou uma linha de código C, etc..

# Uma terceira (e definitiva) definição para **eficiência**

- Dizemos que o tempo de execução é proporcional à $N^d$
- Se o input cresce de N para 2N, o tempo de execução
  - $cN^d$ para $c(aN)^d = c\ 2^d\ N^d$, que é um decréscimo de $2^d$ na execução.
  - Dado que d é uma constante, $2^d$ também é uma constante
  - E claro, com esperamos, polinômios de mais baixa ordem tem melhor comportamento de escala que polinômios de alta ordem.
- Sim, esta definição tem que ser vista com bom senso:
  - $n^{100}$ não é eficiente.
- Mas esta definição tem a vantagem de nos permitir definir que para certos problemas não há algoritmos eficientes

primitiva: uma instrução assembly ou uma linha de código C, etc..

# O que é um algoritmo Eficiente?

**Table 2.1** The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

| | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $1.5^n$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| $n = 10$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 4 sec |
| $n = 30$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 18 min | $10^{25}$ years |
| $n = 50$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 11 min | 36 years | very long |
| $n = 100$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | 1 sec | 12,892 years | $10^{17}$ years | very long |
| $n = 1{,}000$ | $< 1$ sec | $< 1$ sec | 1 sec | 18 min | very long | very long | very long |
| $n = 10{,}000$ | $< 1$ sec | $< 1$ sec | 2 min | 12 days | very long | very long | very long |
| $n = 100{,}000$ | $< 1$ sec | 2 sec | 3 hours | 32 years | very long | very long | very long |
| $n = 1{,}000{,}000$ | 1 sec | 20 sec | 12 days | 31,710 years | very long | very long | very long |

# O que é um algoritmo Eficiente?

- Vamos analisar a afirmação abaixo:

  “Desenvolvi um novo algoritmo chamado TripleX que leva 14,2 segundos para processar 1.000 números, enquanto o método SimpleX leva 42,1 segundos.”

- Você que usa SimpleX o trocaria por TripleX ???

# Análise de Algoritmos

- Há vários fatores a se considerar nesta afirmação:
  - Linguagem de programação utilizada
    - Compilada/Interpretada
    - Alto nível/ Baixo Nível
  - A implementação do SImpleX foi mal feita, enquanto TripleX é muito elaborada
  - Qual máquina em que testei os algoritmos
    - O que dizer quanto à quantidade de memória que possui?
  - Quantidade de dados processador, acima de tudo:
    - Se TripleX  é mais rápido para 1000 números, este continua sendo mais rápido para uma quantidade maior?

# Complexidade de Algoritmos

- Em função destas questões, uma forma justa de predizer a eficiência de um algoritmo é desconsiderar:
    - Hardware
    - Linguagem
    - Habilidade do Programador.
- O que devemos é comparar algoritmos e não programas
    - A esta “arte” dá-se o nome de análise/complexidade de algoritmos.

# SimpleX vs. TripleX

- Dos argumentos citados anteriormente, o número de operações é uma boa medida de eficiência.

- TripleX: para uma entrada de tamanho n, o algoritmo realiza $n^2 + n$ operações. Pensando em termos de função: $f(n) = n^2 + n$.
- SimpleX: para uma entrada de tamanho n, o algoritmo realiza 1.000n operações. Em termos de função, $g(n) = 1.000n$.

# SimpleX vs. TripleX

- Faça os cálculos do desempenho de cada algoritmo para cada tamanho de entrada

| tamanho da entrada $n$ | 1 | 10 | 100 | 1.000 | 10.000 |
|---|---|---|---|---|---|
| $f(n) = n^2 + n$ | | | | | |
| $g(n) = 1.000n$ | | | | | |

# SimpleX vs. TripleX

- Faça os cálculos do desempenho de cada algoritmo para cada tamanho de entrada

| tamanho da entrada $n$ | 1 | 10 | 100 | 1.000 | 10.000 |
|---|---|---|---|---|---|
| $f(n) = n^2 + n$ | 2 | 110 | 10.100 | **1.001.000** | **100.010.000** |
| $g(n) = 1.000n$ | **1.000** | **10.000** | 100.000 | 1.000.000 | 10.000.000 |

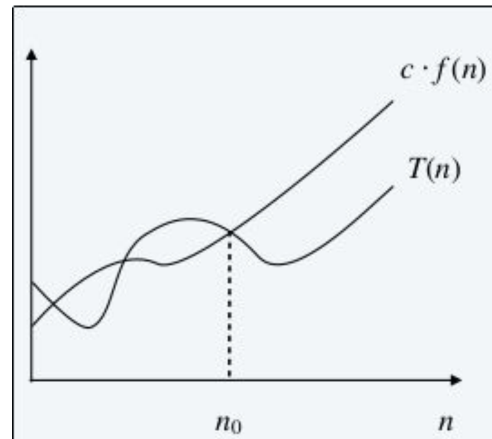
- A partir de n = 1.000, f(n) mantém-se maior e cada vez mais distante de g(n): **Diz-se que f(n) cresce mais rápido do que g(n).**

# Análise Assintótica: **big-oh, omega, theta**

- Se entendemos eficiência como algo relacionado a quão rápido um algoritmo executa, então devemos considerar apenas quando o tamanho de n for **grande**.
- A **eficiência assintótica** de um algoritmo descreve a sua eficiência relativa quando *n* torna-se grande.
- Portanto, para comparar 2 algoritmos, determinam-se as taxas de crescimento de cada um: o algoritmo com menor taxa de crescimento rodará mais rápido quando o tamanho do problema for grande.

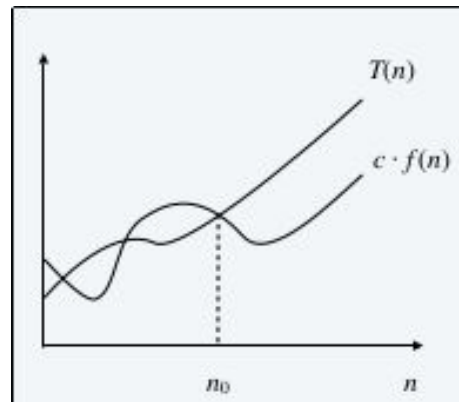
# Big - O

- T(n) é O (f(n)) : leia-se T(n) é ordem (f(n))




- Limite Superior. T(n) é O (f(n)) se houver constantes c > 0 e $n_0$ > 0 tal que T(n) ≤ c f(n) para todo n ≥ $n_0$. Ou seja T é limitada (superiormente) por f.
- Ex: T(n) = $32n^2$ + 17n + 1.
    - T(n) é O($n^2$). <<<<< escolha c = 50, $n_0$ = 1
    - T(n) é também O($n^3$).
    - T(n) não é O(n) e nem O( n log n)
- Uso típico. Ordenação por inserção faz O($n^2$) comparações para ordenar n elementos.

# Big-Omega




- Limite inferior (lower bound). $T(n)$ é $\Omega$ $(f(n))$ se houver constantes $c > 0$ e $n_0 > 0$ tal que $T(n) \geq c\, f(n)$ para todo $n \geq n_0$. Ou seja T é limitada (inferiormente) por f.
- Ex: $T(n) = 32n^2 + 17n + 1$.
  - $T(n)$ é $\Omega$ $(n^2)$. e $\Omega(n)$ <<<<< escolha $c = 32$, $n_0 = 1$
  - $T(n)$ não é $\Omega(n^3)$ e nem $\Omega(n^3 \log n)$
- Uso típico. Qq algoritmo de ordenação baseado em comparações requer $\Omega$ $(n \log n)$ comparações no pior caso

# Algumas considerações

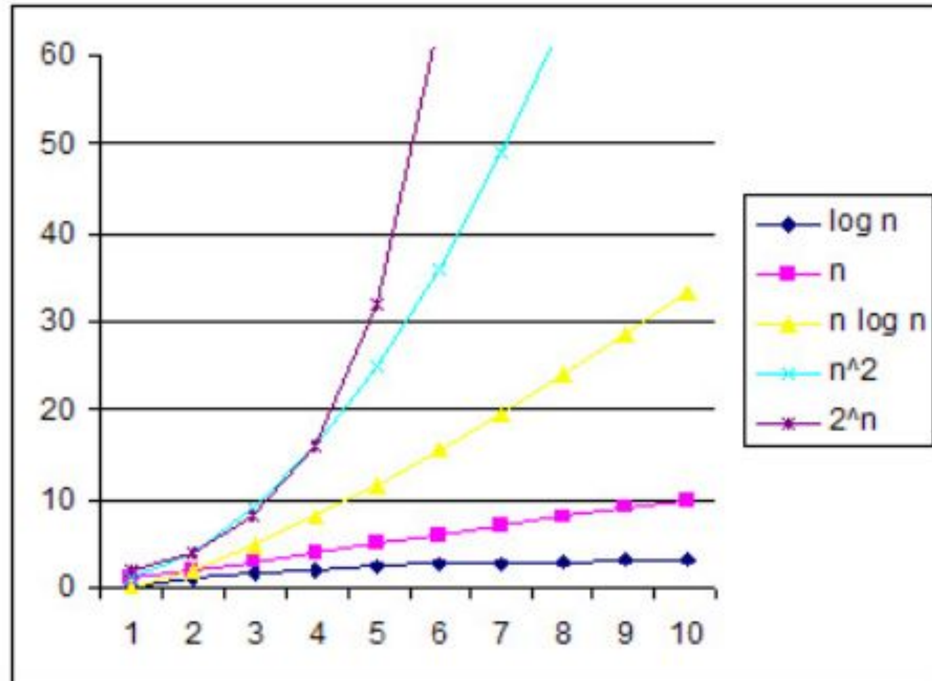
- O uso das notações permite comparar a taxa de crescimento das funções correspondentes aos algoritmos:
    - Não faz sentido comparar pontos isolados das funções, já que podem não corresponder ao comportamento assintótico.
- Ao dizer que $g(n) = O(f(n))$, garante-se que $g(n)$ cresce numa taxa não maior do que $f(n)$, ou seja, $f(n)$ é seu limite superior.
- Ao dizer que $f(n) = \Omega(g(n))$, tem-se que $g(n)$ é o limite inferior de $f(n)$.

# Funções e Taxas de Crescimento

- As mais comuns

| $c$ | constante |
|---|---|
| $log\ n$ | logarítmica |
| $log^2\ n$ | logarítmica ao quadrado |
| $n$ | linear |
| $n\ log\ n$ | |
| $n^2$ | quadrática |
| $n^3$ | cúbica |
| $2^n$ | |
| $a^n$ | exponencial |

# Funções e Taxas de Crescimento



# Taxas de Crescimento

- Apesar de às vezes ser importante, não é comum incluir constantes ou termos de menor ordem em taxas de crescimento:
  - Queremos medir a taxa de crescimento da função, o que torna os “termos menores” irrelevantes,
  - As constantes também dependem do tempo exato de cada operação; como ignoramos os custos reais das operações, ignoramos também as constantes.
- Não se diz que $T(n) = O(2n^2)$ ou que $T(n) = O(n^2 + n)$:
  - Diz-se apenas $T(n) = O(n^2)$.

# Análise de Algoritmos

- Considera-se somente o algoritmo e suas entradas (de tamanho n).
- Para uma entrada de tamanho n, pode-se calcular $T_{melhor}(n)$, $T_{media}(n)$ e $T_{pior}(n)$, ou seja, o melhor tempo de execução, o tempo médio e o pior, respectivamente:
- Obviamente, $T_{melhor}(n) \leq T_{media}(n) \leq T_{pior}(n)$.

- Atenção: para mais de uma entrada, essas funções teriam mais de um argumento.

# Análise de Algoritmos

- Geralmente, utiliza-se somente a análise do pior caso $T_{pior}(n)$, pois ela fornece os limites para todas as entradas, incluindo particularmente as entradas ruins:

# Alguns tipos importantes (e comuns) de tempo de processamento

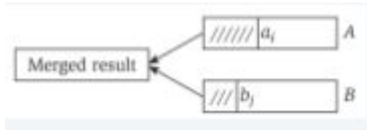
# Linear: O(n)

- Tempo linear. O tempo de execução é proporcional ao tamanho da entrada
- Ex: calcule o valor máximo de n números $a_1 \dots a_n$

```
max ← a1
for i = 2 to n {
        if (ai > max)
                max ← ai
}
```

# Linear: O(n)

- Merge (intercalar). Combine duas listas $A = a_1, a_2, \ldots, a_n$ e $B = b_1, b_2, \ldots , b_n$ em uma lista ordenada




```
i = 1, j = 1
while (both lists are nonempty) {
   if (a_i ≤ b_j) append a_i to output list and increment i
   else           append b_j to output list and increment j
}
append remainder of nonempty list to output list
```

Definição: Intercalar 2 listas de tamanho n é de ordem O(n)

Prova: a cada comparação, o comprimento da saída é incrementado em uma unidade !

# Tempo “Linearrithmic”: O(n log n)

- Aparece em algoritmos do tipo dividir e conquistar
- Ordenação
  - Mergesort e heapsort são algoritmos de ordenação de complexidade O (n log n)

- O Maior Intervalo vazio
  - Dado n time-stamps $x_1, \ldots, x_n$ que representam o tempo em que cópias de um arquivo chega em um servidor, qual é o maior intervalo entre a chegada de arquivos?
  - Ordene os time-stamps. Percorra a lista identificando o maior intervalo entre sucessivos time-stamps.

# Tempo Quadrático: $O(n^2)$

- Ex: O par coordenadas mais próximas
  - Dada uma lista de n coordenadas no plano $(x_1, y_1), \ldots, (x_n, y_n)$, encontre o par mais próximo
- Solução $O(n^2)$. Compute a distância entre cada par de pontos

```
min ← (x1 - x2)² + (y1 - y2)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)² + (yi - yj)²
      if (d < min)
         min ← d
   }
}
```

# Tempo Cúbico: $O(n^3)$

- Conjuntos disjuntos
  - dados n conjuntos $S_1, \ldots, S_n$, cada qual sendo um subconjunto de números 1, … ,n , existe um par de conjuntos que seja disjuntos?
- Solução $O(n^3)$: para cada par de conjuntos determine se são disjuntos

```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```

# Tempo Polinomial: $O(n^k)$

- Conjunto independente de tamanho k
  - Dado um grafo com N nós, existe um subconjunto S de k nós, tal que nenhum par de vértices em S seja ligado por uma aresta?
- Solução O $(n^k)$. enumerar todos os subconjuntos de tamanho k do grafo

```
foreach subset S of k nodes {
   check whether S is an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Verificar se S é independente é $O(n^2)$
- Número de Subconjuntos de k elementos:
- $O(k^2 n^k / k!) = O(n^k)$.

$$\binom{n}{k} = \frac{n(n-1)(n-2)\times\cdots\times(n-k+1)}{k(k-1)(k-2)\times\cdots\times 1} \le \frac{n^k}{k!}$$

# Tempo exponencial:

- Dado um grafo, qual é a cardinalidade máximo de um conjunto independente
- Solução $O(n^2 2^n)$
- Número de subconjuntos no grafo de n nós: $2^n$

```
S* ← φ
foreach subset S of nodes {
   check whether S is an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

# Tempo sub-linear

- Busca em vetor ordenado: dado um vetor V ordenado de n números,  existe um elemento x em V?
- Solução O(log n): busca binária

```
lo ← 1, hi ← n
while (lo ≤ hi) {
   mid ← (lo + hi) / 2
   if      (x < A[mid]) hi ← mid - 1
   else if (x > A[mid]) lo ← mid + 1
   else return yes
}
return no
```