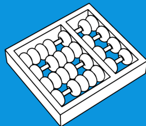
Apresentação da disciplina

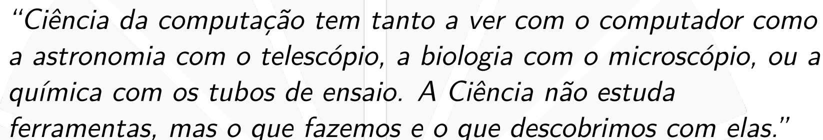
MO417 - Complexidade de Algoritmos I

02/24

0



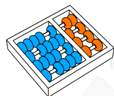
Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br



“Ciência da computação tem tanto a ver com o computador 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.”

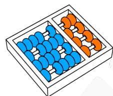
Atribuída a Edsger W. Dijkstra.

Conceitos básicos



Refletindo sobra a frase de Dijkstra

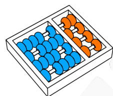
O que estuda a Ciência da Computação?



Problemas computacionais

Um problema computacional é uma relação entre um conjunto de **INSTÂNCIAS** e um conjunto de **SOLUÇÕES**:

- Uma **INSTÂNCIA** é um conjunto de valores conhecidos.
- Uma **SOLUÇÃO** é um conjunto de valores a computar.
- Cada instância corresponde a **UMA OU MAIS** soluções.



Exemplo de problema: Teste de primalidade

Problema: determinar se um dado número é primo.

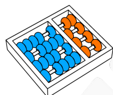
- **INSTÂNCIAS:** números inteiros.
- **SOLUÇÕES:** sim ou não.

Exemplo:

- Instância: 9411461.
- Solução: sim.

Exemplo:

- Instância: 8411461.
- Solução: não.



Exemplo de problema: Ordenação

Problema: ordenar os elementos de um vetor de inteiros.

- **INSTÂNCIAS:** conjunto de vetores de inteiros.
- **SOLUÇÕES:** conjunto de vetores de inteiros em ordem crescente.

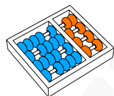
Exemplo:

- Instância:

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

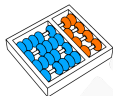
- Solução:

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



Refletindo sobre o estudo de problemas computacionais

Como estudamos (e solucionamos) problemas computacionais?



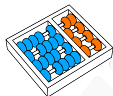
Algoritmos

Um algoritmo é uma sequência de **INSTRUÇÕES** que:

- Recebe uma instância de um problema computacional.
- Devolve uma solução correspondente à instância recebida.

Observações:

- A instância recebida é chamada de **ENTRADA**.
- A solução devolvida é chamada de **SAÍDA**.
- Toda instrução deve ser **BEM DEFINIDA**.

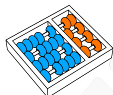


Descrição de algoritmos

Podemos escrever um algoritmo de várias maneiras:

- Em uma linguagem de programação, como C, Pascal, Java, Python...
- Em **PORTUGUÊS**, ou outra língua natural.
- Em **PSEUDOCÓDIGO**, como no livro de CLRS.

Usaremos apenas as duas últimas opções!

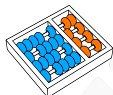


Exemplo de pseudocódigo

Um algoritmo para o problema da ordenação:

Algoritmo: INSERTION-SORT(A, n)

```
1 para j ← 2 até n
2     chave ← A[j]
3     i ← j − 1
4     enquanto i ≥ 1 e A[i] > chave
5         A[i + 1] ← A[i]
6         i ← i − 1
7         A[i + 1] ← chave
```



Más práticas

- **NÃO** misture código com pseudocódigo:

```
1 for (i = 0; i < n; i++) ...
```

```
1 if (A[i] >= chave)
2    break
```

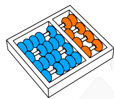
- **NÃO** escreva frases confusas:

```
1 se A[i] > chave
2     troque as posições dos elementos
```

```
1 chave ← pega o próximo elemento
2 i ← procura posição da chave
```

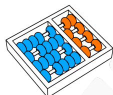
- **EVITE** algoritmos complicados ou com muitas variáveis:

```
1 se j = 1
2     A[2] ← A[1]
3     A[1] ← chave
4 senão
5     enquanto i ≥ 1 e A[i] ≥ chave
6         ...
```



Refletindo sobre algoritmos

Como sabemos se um algoritmo funciona?

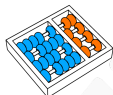


Modelo computacional

Só podemos escrever instruções **BEM DEFINIDAS**:

- O resultado de cada instrução é inambíguo e depende somente do estado corrente da execução.
- Deve ser possível executar cada instrução usando o computador adotado.

O conjunto de instruções permitidas é determinado pelo que chamamos de **MODELO COMPUTACIONAL**.



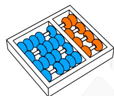
Correção de algoritmos

Um algoritmo está **CORRETO** se:

1. Só utiliza instruções do modelo de computação adotado.
2. Termina para toda instância do problema.
3. Devolve uma solução que correspondente à instância recebida.

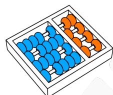
Ao escrever um algoritmo, sempre devemos:

1. Testar o algoritmo:
    - Com uma ou mais instâncias de exemplo.
    - Executando ou simulando o algoritmo.
2. Demonstrar que o algoritmo está correto:
    - Escrever uma prova formal genérica.
    - Vale para **TODA** instância do problema.



Refletindo sobre esforço computacional

Como medimos o esforço computacional de um algoritmo?



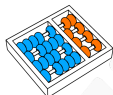
Modelo computacional e complexidade

Vamos estimar o tempo de execução de um algoritmo:

- Contamos o número de **INSTRUÇÕES ELEMENTARES** executadas.
- Supomos que cada instrução elementar consome um tempo contante.

A análise de complexidade depende sempre do modelo computacional adotado!

- Queremos um modelo computacional realista.
- O conjunto de **INSTRUÇÕES ELEMENTARES** deve ser compatível com os computadores modernos.
- Deve ser suficientemente genérico para as diferentes arquiteturas.



Máquinas RAM

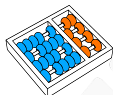
Usaremos o modelo abstrato **RAM** (Random Access Machine):

- Simula máquinas convencionais.
- Possui um único processador sequencial.
- Tipos básicos são números inteiros e pontos flutuantes.
- Cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários.

Instruções elementares:

- Operações aritméticas como soma, subtração, produto...
- Acesso direto às posições da memória.
- Comandos de fluxo de controle (**se**, **enquanto**...)

Operações como exponenciação não são elementares.



Tamanho da entrada

Um parâmetro importante é o **TAMANHO DA ENTRADA**:

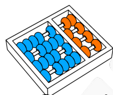
- Normalmente proporcional ao número de bits da entrada.
- Também usamos o número de elementos do vetor.

Problema da primalidade:

- Entrada: inteiro n.
- Tamanho: $\lceil \log_2 n \rceil$ bits.

Problema da ordenação:

- Entrada: vetor $A[1 \dots n]$.
- Tamanho: $n \lceil \log_2 M \rceil$ bits, onde M é o máximo em $A[1 \dots n]$.



Análise assintótica e de pior caso

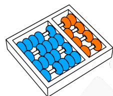
Consideramos apenas instâncias **GRANDES**:

- O número de instruções normalmente cresce com o tamanho da entrada n.
- Instâncias com tamanho limitado por constante gastam tempo constante.
- Para simplificar a análise, o número de instruções é dado em notação assintótica (O, Ω, Θ, o, ω).

Fazemos apenas análise de **PIOR CASO**:

- Restringimos a entradas com um dado tamanho n.
- Consideramos apenas uma instância para a qual o algoritmo executa o maior número de instruções.

Denotamos por $T(n)$ o número de instruções executadas no pior caso para entradas de tamanho n.



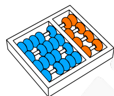
Características e limitações

Esse tipo de análise de complexidade:

- Normalmente **ESTIMA** bem tempo de execução real.
- Permite comparar diversos algoritmos para um problema.
- Continua relevante mesmo com evoluções tecnológicas.

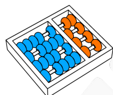
Limitações:

- É uma análise **PESSIMISTA** do tempo de execução.
- Em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso.
- Não fornece informação sobre tempo de execução médio.



Refletindo sobre eficiência

Como decidimos se um algoritmo é mais eficiente que outro?



Algoritmos bons e ruins

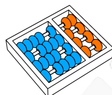
Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos:

1. Usando um computador A com velocidade $1GHz$ e com um algoritmo que executa $\mathbf{2n^2}$ instruções.
2. Usando um computador B com velocidade $10MHz$ e com um algoritmo que executa $\mathbf{50n}\log_2 \mathbf{n}$ instruções.

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B.
- A constante multiplicativa do segundo é bem maior:
  - Pode ser devido a uma linguagem de mais alto nível,
  - ou devido a um programador menos experiente.



Algoritmos bons e ruins

Que implementação ordena um milhão de elementos primeiro?

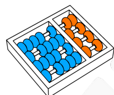
1. A máquina A leva:

$$\frac{2 \cdot (10^6)^2 \text{instruções}}{10^9 \text{instruções } / \text{segundo}} \approx 2000 \text{segundos}$$

2. A máquina B leva:

$$\frac{50 \cdot (10^6 \log_2 10^6) \text{instruções}}{10^7 \text{instruções } / \text{segundo}} \approx 100 \text{segundos}$$

- A máquina B foi **VINTE VEZES** mais rápida do que A
- Se fossem 10 milhões de elementos, a razão seria de **2,3 DIAS** para **20 MINUTOS**

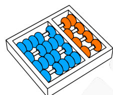


Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
$\mathbf{2^n}$	$\mathbf{N_5}$	$\mathbf{N_5 + 6{,}64}$	$\mathbf{N_5 + 9{,}97}$
$\mathbf{3^n}$	$\mathbf{N_6}$	$\mathbf{N_6 + 4{,}19}$	$\mathbf{N_6 + 6{,}29}$

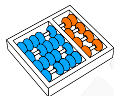
Se só tivermos um algoritmo ruim, não iremos resolver problemas muito maiores.



Conclusões

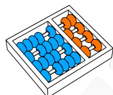
Algumas conclusões:

- Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.
- Isso é tão importante quanto o projeto de hardware.
- O ganho obtido ao melhorar a complexidade de um algoritmo não poderia ser obtida simplesmente com o avanço da tecnologia.
- Queremos estudar principalmente algoritmos fundamentais, que produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, classificadores, etc).



Refletindo sobre eficiência computacional

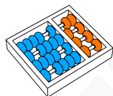
Quais algoritmos consideramos eficientes?



Eficiência computacional

- Se um algoritmo requer 2^n espaço para entradas de tamanho n, então para $n \geq 247$ precisará mais que o **número de átomos (estimado) no universo conhecido!!**

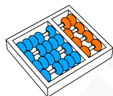
- Se um algoritmo requer 2^n operações para entradas de tamanho n, então para $n \geq 74$ precisará executar pelo menos **um século em um computador que realiza** 1000000 **de operações por milissegundo!!**



Eficiência computacional

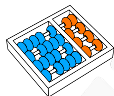
Um algoritmo cuja complexidade de tempo é limitada por um **POLINÔMIO** no tamanho da entrada é considerado **EFICIENTE**.

Um problema é considerado **TRATÁVEL** se pode ser solucionado por um algoritmo eficiente. Em outro caso, é considerado **INTRATÁVEL**.



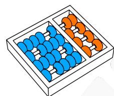
Relações entre problemas

- Os problemas se podem agrupar em diferentes classes de complexidade como P e NP.
- Para um número grande de problemas (em NP) se desconhece se eles são tratáveis ou não.
- Contudo, os problemas de uma subclasse (NP-completo) tem uma propriedade interessante: todos são tratáveis ou nenhum é tratável.
- Ainda, se algum problema NP-completo for tratável, então todos os problemas em NP também o são.



Refletindo sobre limitações

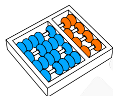
Quais são as limitações da Computação?



Limitações inerentes da Computação

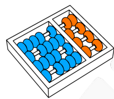
- Impossibilidade de solucionar eficientemente alguns problemas.
- Impossibilidade de solucionar alguns problemas.

Existem também limitações humanas.



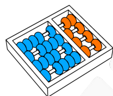
Por que estudar e pesquisar essas limitações?

- Para conhecer as limitações fundamentais da Ciência da Computação.
- Para analisar a dificuldade de solucionar novos problemas.
- Para evitar esforços fúteis.
- Para satisfazer a curiosidade intelectual.



Refletindo sobre limitações

Existem problemas que não podem ser solucionados?

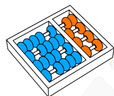


Problema da parada de programas

Um computador pode determinar o problema da parada de programas?

Problema: determinar se um dado programa para com alguma entrada.

- **INSTÂNCIAS:** conjunto de programas e suas entradas.
- **SOLUÇÕES:** sim ou não.

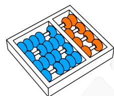


Como solucionar o problema?

Executamos **P** com a entrada **I** até que pare ou até que algum limite de tempo ou memória seja atingido?

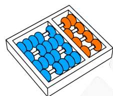
Que acontece se **P** para após consumir um ou mais bytes, ou após executar uma ou mais unidades de tempo?

Existem outras estratégias?

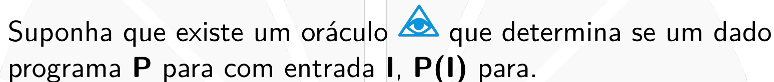


O que fazer?

Podemos tentar provar que um computador não pode determinar esse problema usando **uma demonstração matemática!**



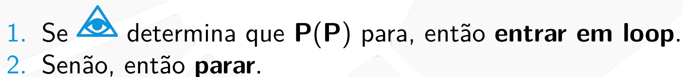
Demonstração



Suponha que existe um oráculo que determina se um dado programa **P** para com entrada **I**, **P(I)** para.

Consideremos o programa:

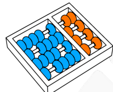
D (**P**):



1. Se determina que **P**(**P**) para, então **entrar em loop**.
2. Senão, então **parar**.

O que acontece com o chamado **D(D)**?

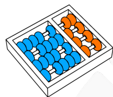
Escopo e avaliação



Objetivos

O que veremos nesta disciplina?

1. Demonstrar **CORREÇÃO** de um algoritmo:
   - Como ter certeza de que a saída está correta?
   - Como convencer outras pessoas disso?
2. Analisar a **COMPLEXIDADE** de um algoritmo:
   - Como estimar a quantidade de recursos utilizados?
   - Recursos podem ser tempo, memória, acesso a rede etc.
3. Utilizar técnicas conhecidas de **PROJETO** de algoritmos:
   - Divisão e conquista, programação dinâmica etc.
   - Utilizar **RECURSÃO** adequadamente.
4. Entender a **DIFICULDADE** intrínseca de alguns problemas:
   - Inexistência de algoritmos eficientes.
   - Identificar os problemas intratáveis.

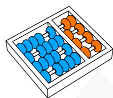


Carga horária

4 horas cada semana, dividas em duas aulas de duas horas cada:

- 1 aula nas terças.
- 1 aula nas quintas.

Totalizando: 60 horas em 30 aulas!



Notas

(P_1) Primeira prova, sobre projeto e análise de algoritmos.

(P_2) Segunda prova, sobre algoritmos em grafos.

(P_3) Terceira prova, sobre problemas e classes de complexidade computacional.

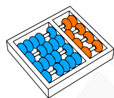
(E_1) Primeira entrega de exercícios, sobre projeto e análise de algoritmos.

(E_2) Segunda entrega de exercícios, sobre algoritmos em grafos.

(R) Relatório (com possível apresentação/discussão) sobre problemas NP-completos.

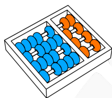
A nota final será uma média pesada:

$$\mathbf{M} = 0{,}2\mathbf{P_1} + 0{,}2\mathbf{P_2} + 0{,}3\mathbf{P_3} + 0{,}1\mathbf{E_1} + 0{,}1\mathbf{E_2} + 0{,}1\mathbf{R}.$$

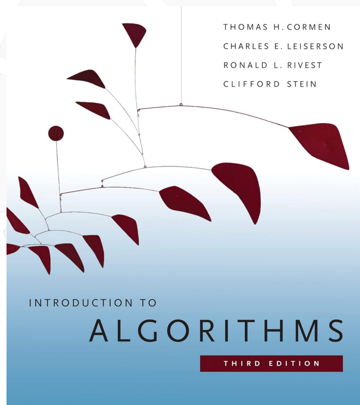


Notas

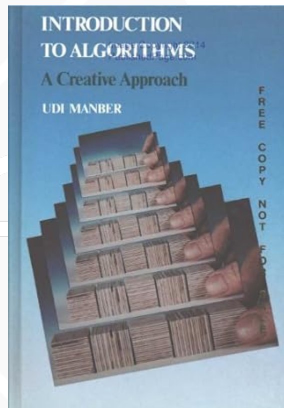
M	Conceito final
$[8.5, 10.0]$	A
$[7.0, 8.5)$	B
$[5.0, 7.0)$	C
$[0.0, 5.0)$	D



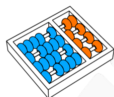
Bibliografia principal



T. Cormen, C. Leiserson, R. Rivest, C. Stein. *"Introduction to Algorithms"*. MIT Press (MA); 3rd ed. (2009).



U. Manber. *"Introduction to Algorithms: A Creative Approach"*. Addison-Wesley. (1989).



Bibliografia complementar

C. Papadimitriou, K. Steiglitz. "Combinatorial Optimization: Algorithms and Complexity". Dover Publications. (1998).

M. Sipser. "Introduction to the Theory of Computation". Cengage Learning; 3rd ed. (2012).

J. Kleinberg, E. Tardos. "Algorithm Design". Pearson; 1st ed. (2005).

M. Garey e D. Johnson. "Computers and Intractability: a Guide to the Theory of NP- Completeness". Freeman. (1979).

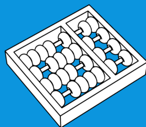
A. Aho, J. Hopcroft, J. Ullman. "The Design and Analysis of Computer Algorithms". Addison-Wesley. (1974).

Apresentação da disciplina

MO417 - Complexidade de Algoritmos I

02/24

0



Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br