

APRESENTAÇÃO DA DISCIPLINA

MC458 - Projeto e Análise de
Algoritmos I

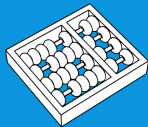
Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

08/25

00



UNICAMP



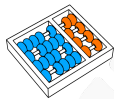


“Ciência da computação tem tanto a ver com o computador como a astronomia com o telescópio, a biologia com o microscópio, ou a química com os tubos de ensaio. A Ciência não estuda ferramentas, mas o que fazemos e o que descobrimos com elas.”

Atribuída a Edsger W. Dijkstra.



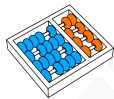
CONCEITOS BÁSICOS



Refletindo sobre a frase de Dijkstra

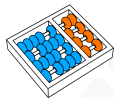
**O que estuda a Ciência da
Computação?**





Problemas computacionais

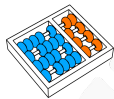
Um problema computacional é uma relação entre um conjunto de **INSTÂNCIAS** e um conjunto de **SOLUÇÕES**:



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.



Problemas computacionais

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

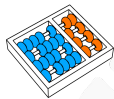
- ▶ Uma **INSTÂNCIA** é um conjunto de valores conhecidos.
- ▶ Uma **SOLUÇÃO** é um conjunto de valores a computar.



Problemas computacionais

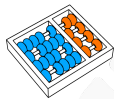
Um problema computacional é uma relação entre um conjunto de **INSTÂNCIAS** e um conjunto de **SOLUÇÕES**:

- ▶ 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

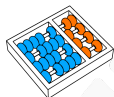
Problema: determinar se um dado número é primo.



Exemplo de problema: Teste de primalidade

Problema: determinar se um dado número é primo.

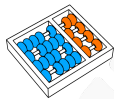
▶ **INSTÂNCIAS:** números inteiros.



Exemplo de problema: Teste de primalidade

Problema: determinar se um dado número é primo.

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

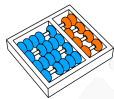


Exemplo de problema: Teste de primalidade

Problema: determinar se um dado número é primo.

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

Exemplo:



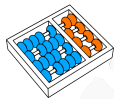
Exemplo de problema: Teste de primalidade

Problema: determinar se um dado número é primo.

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

Exemplo:

- ▶ Instância: 9411461.



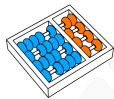
Exemplo de problema: Teste de primalidade

Problema: determinar se um dado número é primo.

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

Exemplo:

- ▶ Instância: 9411461.
- ▶ Solução: sim.



Exemplo de problema: Teste de primalidade

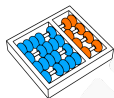
Problema: determinar se um dado número é primo.

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

Exemplo:

- ▶ Instância: 9411461.
- ▶ Solução: sim.

Exemplo:



Exemplo de problema: Teste de primalidade

Problema: determinar se um dado número é primo.

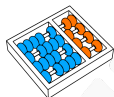
- ▶ **INSTÂNCIAS:** números inteiros.
- ▶ **SOLUÇÕES:** sim ou não.

Exemplo:

- ▶ Instância: 9411461.
- ▶ Solução: sim.

Exemplo:

- ▶ Instância: 8411461.



Exemplo de problema: Teste de primalidade

Problema: determinar se um dado número é primo.

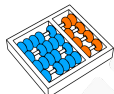
- ▶ **INSTÂNCIAS:** números inteiros.
- ▶ **SOLUÇÕES:** sim ou não.

Exemplo:

- ▶ Instância: 9411461.
- ▶ Solução: sim.

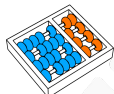
Exemplo:

- ▶ Instância: 8411461.
- ▶ Solução: não.



Exemplo de problema: Ordenação

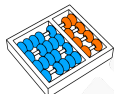
Problema: ordenar os elementos de um vetor de inteiros.



Exemplo de problema: Ordenação

Problema: ordenar os elementos de um vetor de inteiros.

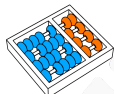
▶ **INSTÂNCIAS:** conjunto de vetores de inteiros.



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.

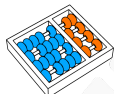


Exemplo de problema: Ordenação

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:



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	11	11	99	33	55	77



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	11	11	99	33	55	77

- ▶ Solução:

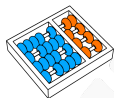
1										n
11	11	33	33	33	33	44	55	55	77	99



Refletindo sobre o estudo de problemas computacionais

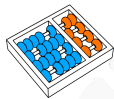


**Como estudamos (e solucionamos)
problemas computacionais?**



Algoritmos

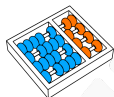
Um algoritmo é uma sequência de **INSTRUÇÕES** que:



Algoritmos

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

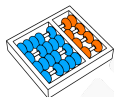
- ▶ Recebe uma instância de um problema computacional.



Algoritmos

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

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

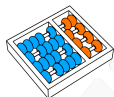


Algoritmos

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

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

Observações:



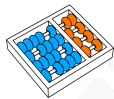
Algoritmos

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**.



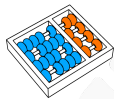
Algoritmos

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**.



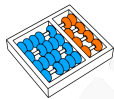
Algoritmos

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

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

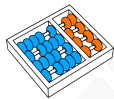
Observações:

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



Descrição de algoritmos

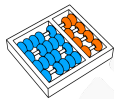
Podemos escrever um algoritmo de várias maneiras:



Descrição de algoritmos

Podemos escrever um algoritmo de várias maneiras:

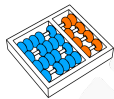
- ▶ Em uma linguagem de programação, como C, Pascal, Java, Python...



Descrição de algoritmos

Podemos escrever um algoritmo de várias maneiras:

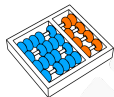
- ▶ Em uma linguagem de programação, como C, Pascal, Java, Python...
- ▶ Em **PORTUGUÊS**, ou outra língua natural.



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.

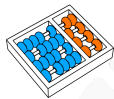


Descrição de algoritmos

Podemos escrever um algoritmo de várias maneiras:

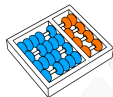
- ▶ Em uma linguagem de programação, como C, Pascal, Java, Python...
- ▶ Em **PORTUGUÊS**, ou outra língua natural.
- ▶ Em **PSEUDOCÓDIGO**, como no livro de CLRS.

Usaremos apenas as duas últimas opções!



Exemplo de pseudocódigo

Um algoritmo para o problema da ordenação:

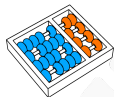


Exemplo de pseudocódigo

Um algoritmo para o problema da ordenação:

Algoritmo: INSERTION-SORT(A, n)

```
1 para  $j \leftarrow 2$  até  $n$ 
2   chave  $\leftarrow A[j]$ 
3    $i \leftarrow j - 1$ 
4   enquanto  $i \geq 1$  e  $A[i] > \text{chave}$ 
5      $A[i + 1] \leftarrow A[i]$ 
6      $i \leftarrow i - 1$ 
7      $A[i + 1] \leftarrow \text{chave}$ 
```

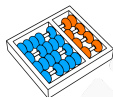


Más prácticas

► **NÃO** misture código com pseudocódigo:

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

```
1 if ( $A[i] \geq \text{chave}$ )  
2   break
```



Más prácticas

- ▶ **NÃO** misture código com pseudocódigo:

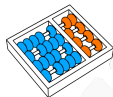
```
1 for (i = 0; i < n; i++) ...
```

```
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
```



Más prácticas

- ▶ **NÃO** misture código com pseudocódigo:

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

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

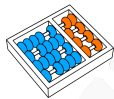
- ▶ **NÃO** escreva frases confusas:

```
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:

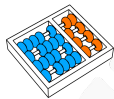
```
1 se j = 1
2   | A[2] ← A[1]
3   | A[1] ← chave
4 senão
5   | enquanto i ≥ 1 e A[i] ≥ chave
6   |   | ...
```



Refletindo sobre algoritmos

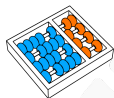


Como sabemos se um algoritmo funciona?



Modelo computacional

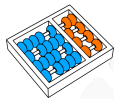
Só podemos escrever instruções **BEM DEFINIDAS**:



Modelo computacional

Só podemos escrever instruções **BEM DEFINIDAS**:

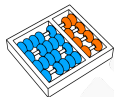
- ▶ O resultado de cada instrução é inambíguo e depende somente do estado corrente da execução.



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.

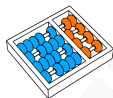


Modelo computacional

Só podemos escrever instruções **BEM DEFINIDAS**:

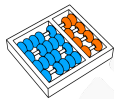
- ▶ O resultado de cada instrução é inambíguo e depende somente do estado corrente da execução.
- ▶ Deve ser possível executar cada instrução usando o computador adotado.

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



Correção de algoritmos

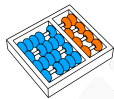
Um algoritmo está **CORRETO** se:



Correção de algoritmos

Um algoritmo está **CORRETO** se:

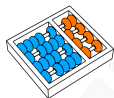
1. Só utiliza instruções do modelo de computação adotado.



Correção de algoritmos

Um algoritmo está **CORRETO** se:

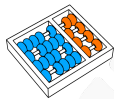
1. Só utiliza instruções do modelo de computação adotado.
2. Termina para toda instância do problema.



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.

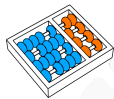


Correção de algoritmos

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:



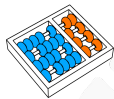
Correção de algoritmos

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:



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.



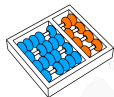
Correção de algoritmos

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.



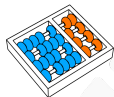
Correção de algoritmos

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:



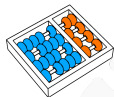
Correção de algoritmos

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.



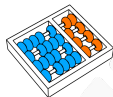
Correção de algoritmos

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:

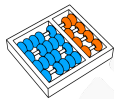
1. Testar o algoritmo:
 - ▶ Com uma ou mais instâncias de exemplo.
 - ▶ Executando ou simulando o algoritmo.
2. Demonstrar que o algoritmo está correto:
 - ▶ Escrever uma prova formal genérica.
 - ▶ Vale para **TODA** instância do problema.



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:



Modelo computacional e complexidade

Vamos estimar o tempo de execução de um algoritmo:

- ▶ Contamos o número de **INSTRUÇÕES ELEMENTARES** executadas.



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.



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!



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.



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.



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):



Máquinas RAM

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

- ▶ Simula máquinas convencionais.



Máquinas RAM

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

- ▶ Simula máquinas convencionais.
- ▶ Possui um único processador sequencial.



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.



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.



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:



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...



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.



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...**)



Máquinas RAM

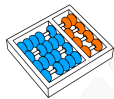
Usaremos o modelo abstrato **RAM** (Random Access Machine):

- ▶ 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:

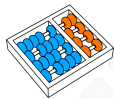
- ▶ Operações aritméticas como soma, subtração, produto...
- ▶ Acesso direto às posições da memória.
- ▶ Comandos de fluxo de controle (**se, enquanto...**)

Operações como exponenciação não são elementares.



Tamanho da entrada

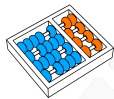
Um parâmetro importante é o **TAMANHO DA ENTRADA**:



Tamanho da entrada

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

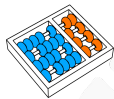
- ▶ Normalmente proporcional ao número de bits da entrada.



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.



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:



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 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.



Tamanho da entrada

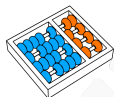
Um parâmetro importante é o **TAMANHO DA ENTRADA**:

- ▶ 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:



Tamanho da entrada

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

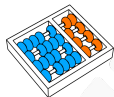
- ▶ Normalmente proporcional ao número de bits da entrada.
- ▶ Também usamos o número de elementos do vetor.

Problema da primalidade:

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

Problema da ordenação:

- ▶ Entrada: vetor $A[1 \dots n]$.



Tamanho 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

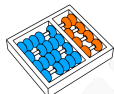
Consideramos apenas instâncias **GRANDES**:



Análise assintótica e de pior caso

Consideramos apenas instâncias **GRANDES**:

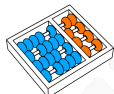
- ▶ O número de instruções normalmente cresce com o tamanho da entrada n .



Análise assintótica e de pior caso

Consideramos apenas instâncias **GRANDES**:

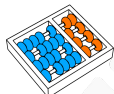
- ▶ O número de instruções normalmente cresce com o tamanho da entrada n .
- ▶ Instâncias com tamanho limitado por constante gastam tempo constante.



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).



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**:



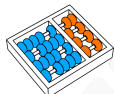
Análise assintótica e de pior caso

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 .



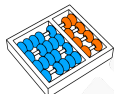
Análise assintótica e de pior caso

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.



Análise assintótica e de pior caso

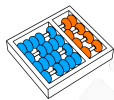
Consideramos apenas instâncias **GRANDES**:

- ▶ 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**:

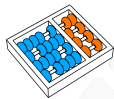
- ▶ Restringimos a entradas com um dado tamanho n .
- ▶ Consideramos apenas uma instância para a qual o algoritmo executa o maior número de instruções.

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



Características e limitações

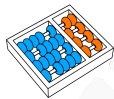
Esse tipo de análise de complexidade:



Características e limitações

Esse tipo de análise de complexidade:

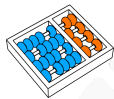
- ▶ Normalmente **ESTIMA** bem tempo de execução real.



Características e limitações

Esse tipo de análise de complexidade:

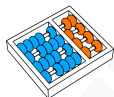
- ▶ Normalmente **ESTIMA** bem tempo de execução real.
- ▶ Permite comparar diversos algoritmos para um problema.



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.

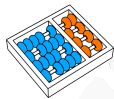


Características e limitações

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:



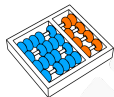
Características e limitações

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.



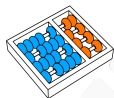
Características e limitações

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.



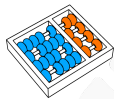
Características e limitações

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:

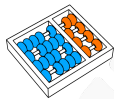
- ▶ É uma análise **PESSIMISTA** do tempo de execução.
- ▶ Em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso.
- ▶ Não fornece informação sobre tempo de execução médio.



Refletindo sobre eficiência

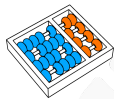


Como decidimos se um algoritmo é mais eficiente que outro?



Algoritmos bons e ruins

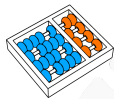
Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?



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:



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 $2n^2$ instruções.

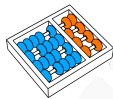


Algoritmos bons e ruins

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 $2n^2$ instruções.
2. Usando um computador B com velocidade $10MHz$ e com um algoritmo que executa $50n \log_2 n$ instruções.



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 $2n^2$ instruções.
2. Usando um computador B com velocidade $10MHz$ e com um algoritmo que executa $50n \log_2 n$ instruções.

Comparando as implementações:



Algoritmos bons e ruins

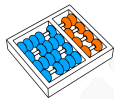
Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos:

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

Comparando as implementações:

- O computador A é 100 vezes mais rápido do que B .



Algoritmos bons e ruins

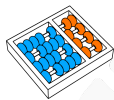
Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos:

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

Comparando as implementações:

- ▶ O computador A é 100 vezes mais rápido do que B .
- ▶ A constante multiplicativa do segundo é bem maior:



Algoritmos bons e ruins

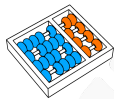
Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos:

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



Algoritmos bons e ruins

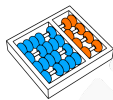
Que diferença faz ter um algoritmo $\Theta(n^2)$ ou $\Theta(n \log n)$?

Suponha que queremos ordenar um vetor de n elementos:

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



Algoritmos bons e ruins

Que implementação ordena um milhão de elementos primeiro?

1. A máquina A leva:

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



Algoritmos bons e ruins

Que implementação ordena um milhão de elementos primeiro?

1. A máquina *A* leva:

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

2. A máquina *B* leva:

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



Algoritmos bons e ruins

Que implementação ordena um milhão de elementos primeiro?

1. A máquina *A* leva:

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

2. A máquina *B* leva:

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

► A máquina *B* foi **VINTE VEZES** mais rápida do que *A*



Algoritmos bons e ruins

Que implementação ordena um milhão de elementos primeiro?

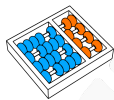
1. A máquina *A* leva:

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

2. A máquina *B* leva:

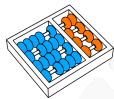
$$\frac{50 \cdot (10^6 \log_2 10^6) \text{instruções}}{10^7 \text{instruções /segundo}} \approx 100 \text{segundos}$$

- ▶ 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

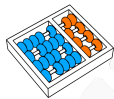
E se usarmos um supercomputador?



Algoritmos e tecnologia

E se usarmos um supercomputador?

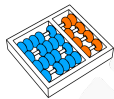
$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
--------	------------------	-----------------	------------------



Algoritmos e tecnologia

E se usarmos um supercomputador?

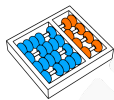
$f(n)$	Computador atual	$100\times$ mais rápido	$1000\times$ mais rápido
n	N_1	$100N_1$	$1000N_1$



Algoritmos e tecnologia

E se usarmos um supercomputador?

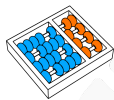
$f(n)$	Computador atual	$100\times$ mais rápido	$1000\times$ mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31,6N_2$



Algoritmos e tecnologia

E se usarmos um supercomputador?

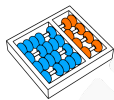
$f(n)$	Computador atual	$100\times$ mais rápido	$1000\times$ mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31,6N_2$
n^3	N_3	$4,64N_3$	$10N_3$



Algoritmos e tecnologia

E se usarmos um supercomputador?

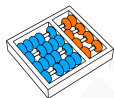
$f(n)$	Computador atual	$100\times$ mais rápido	$1000\times$ mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31,6N_2$
n^3	N_3	$4,64N_3$	$10N_3$
n^5	N_4	$2,5N_4$	$3,98N_4$



Algoritmos e tecnologia

E se usarmos um supercomputador?

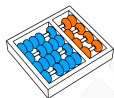
$f(n)$	Computador atual	$100\times$ mais rápido	$1000\times$ mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31,6N_2$
n^3	N_3	$4,64N_3$	$10N_3$
n^5	N_4	$2,5N_4$	$3,98N_4$
2^n	N_5	$N_5 + 6,64$	$N_5 + 9,97$



Algoritmos e tecnologia

E se usarmos um supercomputador?

$f(n)$	Computador atual	$100\times$ mais rápido	$1000\times$ 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	$2,5N_4$	$3,98N_4$
2^n	N_5	$N_5 + 6,64$	$N_5 + 9,97$
3^n	N_6	$N_6 + 4,19$	$N_6 + 6,29$

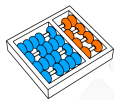


Algoritmos e tecnologia

E se usarmos um supercomputador?

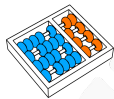
$f(n)$	Computador atual	$100\times$ mais rápido	$1000\times$ mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31,6N_2$
n^3	N_3	$4,64N_3$	$10N_3$
n^5	N_4	$2,5N_4$	$3,98N_4$
2^n	N_5	$N_5 + 6,64$	$N_5 + 9,97$
3^n	N_6	$N_6 + 4,19$	$N_6 + 6,29$

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



Conclusões

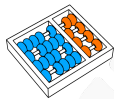
Algumas conclusões:



Conclusões

Algumas conclusões:

- ▶ Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.



Conclusões

Algumas conclusões:

- ▶ Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.
- ▶ Isso é tão importante quanto o projeto de hardware.



Conclusões

Algumas conclusões:

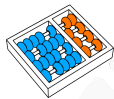
- ▶ Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.
- ▶ Isso é tão importante quanto o projeto de hardware.
- ▶ O ganho obtido ao melhorar a complexidade de um algoritmo não poderia ser obtida simplesmente com o avanço da tecnologia.



Conclusões

Algumas conclusões:

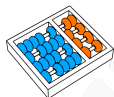
- ▶ Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.
- ▶ Isso é tão importante quanto o projeto de hardware.
- ▶ O ganho obtido ao melhorar a complexidade de um algoritmo não poderia ser obtida simplesmente com o avanço da tecnologia.
- ▶ Queremos estudar principalmente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



Refletindo sobre eficiência computacional

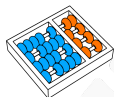


**Quais algoritmos consideramos
eficientes?**



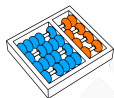
Eficiência computacional

- ▶ 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!!**



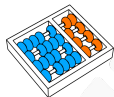
Eficiência computacional

- ▶ 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**.



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.



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.



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.



Relações entre problemas

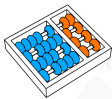
- ▶ Os problemas se podem agrupar em diferentes classes de complexidade como P e NP.
- ▶ Para um número grande de problemas (em NP) se desconhece se eles são tratáveis ou não.
- ▶ Contudo, os problemas de uma subclasse (NP-completo) tem uma propriedade interessante: todos são tratáveis ou nenhum é tratável.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



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.

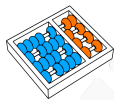
Veremos mais sobre esse tema em próximos semestres, na MC558...



Refletindo sobre limitações

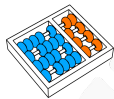
**Quais são as limitações da
Computação?**





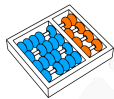
Limitações inerentes da Computação

- ▶ Impossibilidade de solucionar eficientemente alguns problemas.



Limitações inerentes da Computação

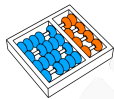
- ▶ Impossibilidade de solucionar eficientemente alguns problemas.
- ▶ Impossibilidade de solucionar alguns problemas.



Limitações inerentes da Computação

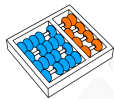
- ▶ Impossibilidade de solucionar eficientemente alguns problemas.
- ▶ Impossibilidade de solucionar alguns problemas.

Existem também limitações humanas.



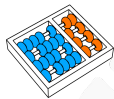
Por que estudar e pesquisar essas limitações?

- ▶ Para conhecer as limitações fundamentais da Ciência da Computação.



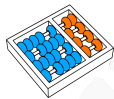
Por que estudar e pesquisar essas limitações?

- ▶ Para conhecer as limitações fundamentais da Ciência da Computação.
- ▶ Para analisar a dificuldade de solucionar novos problemas.



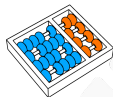
Por que estudar e pesquisar essas limitações?

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



Por que estudar e pesquisar essas limitações?

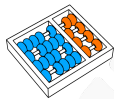
- ▶ Para conhecer as limitações fundamentais da Ciência da Computação.
- ▶ Para analisar a dificuldade de solucionar novos problemas.
- ▶ Para evitar esforços fúteis.
- ▶ Para satisfazer a curiosidade intelectual.



Refletindo sobre limitações

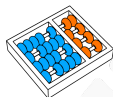


**Existem problemas que não podem
ser solucionados?**



Problema da parada de programas

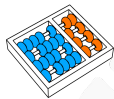
Um computador pode determinar o problema da parada de programas?



Problema da parada de programas

Um computador pode determinar o problema da parada de programas?

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

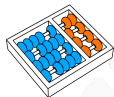


Problema da parada de programas

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.

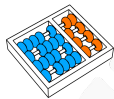


Problema da parada de programas

Um computador pode determinar o problema da parada de programas?

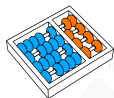
Problema: determinar se um dado programa para com alguma entrada.

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



Como solucionar o problema?

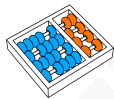
Executamos **P** com a entrada **I** até que pare ou até que algum limite de tempo ou memória seja atingido?



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?

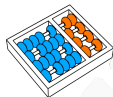


Como solucionar o problema?

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

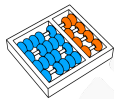
Que acontece se **P** para após consumir um ou mais bytes, ou após executar uma ou mais unidades de tempo?

Existem outras estratégias?



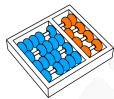
O que fazer?

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



Ideia da demonstração

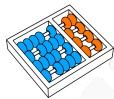
Suponha que existe um oráculo  que determina se um dado programa **P** para com entrada **I**, **P(I)** para.



Ideia da demonstração

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

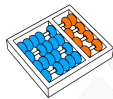
Consideremos o programa:



Ideia da demonstração

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

Consideremos o programa:

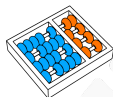


Ideia da demonstração

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

Consideremos o programa:

D (**P**):



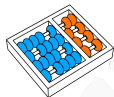
Ideia da demonstração

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**.



Ideia da 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**.



Ideia da demonstração

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

Consideremos o programa:

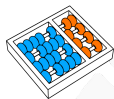
$D(P)$:

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

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



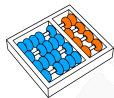
ESCOPO E AVALIAÇÃO



Escopo e avaliação

Objetivos

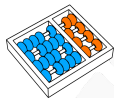
O que veremos nesta disciplina?



Objetivos

O que veremos nesta disciplina?

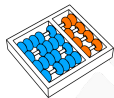
1. Demonstrar **CORREÇÃO** de um algoritmo:



Objetivos

O que veremos nesta disciplina?

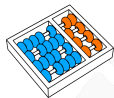
1. Demonstrar **CORREÇÃO** de um algoritmo:
 - ▶ Como ter certeza de que a saída está correta?



Objetivos

O que veremos nesta disciplina?

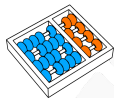
1. Demonstrar **CORREÇÃO** de um algoritmo:
 - ▶ Como ter certeza de que a saída está correta?
 - ▶ Como convencer outras pessoas disso?



Objetivos

O que veremos nesta disciplina?

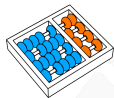
1. Demonstrar **CORREÇÃO** de um algoritmo:
 - ▶ Como ter certeza de que a saída está correta?
 - ▶ Como convencer outras pessoas disso?
2. Analisar a **COMPLEXIDADE** de um algoritmo:



Objetivos

O que veremos nesta disciplina?

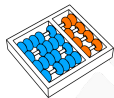
1. Demonstrar **CORREÇÃO** de um algoritmo:
 - ▶ Como ter certeza de que a saída está correta?
 - ▶ Como convencer outras pessoas disso?
2. Analisar a **COMPLEXIDADE** de um algoritmo:
 - ▶ Como estimar a quantidade de recursos utilizados?



Objetivos

O que veremos nesta disciplina?

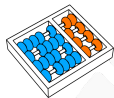
1. Demonstrar **CORREÇÃO** de um algoritmo:
 - ▶ Como ter certeza de que a saída está correta?
 - ▶ Como convencer outras pessoas disso?
2. Analisar a **COMPLEXIDADE** de um algoritmo:
 - ▶ Como estimar a quantidade de recursos utilizados?
 - ▶ Recursos podem ser tempo, memória, acesso a rede, etc.



Objetivos

O que veremos nesta disciplina?

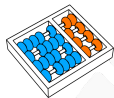
1. Demonstrar **CORREÇÃO** de um algoritmo:
 - ▶ Como ter certeza de que a saída está correta?
 - ▶ Como convencer outras pessoas disso?
2. Analisar a **COMPLEXIDADE** de um algoritmo:
 - ▶ Como estimar a quantidade de recursos utilizados?
 - ▶ Recursos podem ser tempo, memória, acesso a rede, etc.
3. Utilizar técnicas conhecidas de **PROJETO** de algoritmos:



Objetivos

O que veremos nesta disciplina?

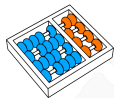
1. Demonstrar **CORREÇÃO** de um algoritmo:
 - ▶ Como ter certeza de que a saída está correta?
 - ▶ Como convencer outras pessoas disso?
2. Analisar a **COMPLEXIDADE** de um algoritmo:
 - ▶ Como estimar a quantidade de recursos utilizados?
 - ▶ Recursos podem ser tempo, memória, acesso a rede, etc.
3. Utilizar técnicas conhecidas de **PROJETO** de algoritmos:
 - ▶ Divisão e conquista, programação dinâmica, etc.



Objetivos

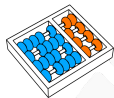
O que veremos nesta disciplina?

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.



Carga horária

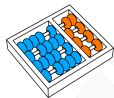
4 horas cada semana, divididas em duas aulas de duas horas cada:



Carga horária

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

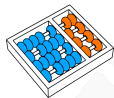
- ▶ 1 aula nas segundas.



Carga horária

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

- ▶ 1 aula nas segundas.
- ▶ 1 aula nas quartas.

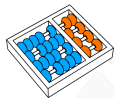


Carga horária

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

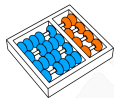
- ▶ 1 aula nas segundas.
- ▶ 1 aula nas quartas.

Totalizando: 60 horas em 30 aulas!



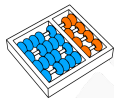
Avaliação e notas

- (T) Quinzenalmente, serão aplicados (presencialmente) testes teóricos com duração de 30 minutos no final das aulas.



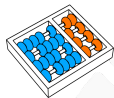
Avaliação e notas

- (T) Quinzenalmente, serão aplicados (presencialmente) testes teóricos com duração de 30 minutos no final das aulas.
- (L) Ao longo do semestre serão orientadas listas de exercícios de programação, com prazo de entrega de uma semana.



Avaliação e notas

- (T) Quinzenalmente, serão aplicados (presencialmente) testes teóricos com duração de 30 minutos no final das aulas.
- (L) Ao longo do semestre serão orientadas listas de exercícios de programação, com prazo de entrega de uma semana.
- (P) Haverá um projeto final a ser desenvolvido em duplas.

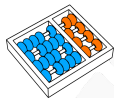


Avaliação e notas

- (*T*) Quinzenalmente, serão aplicados (presencialmente) testes teóricos com duração de 30 minutos no final das aulas.
- (*L*) Ao longo do semestre serão orientadas listas de exercícios de programação, com prazo de entrega de uma semana.
- (*P*) Haverá um projeto final a ser desenvolvido em duplas.

A nota final será uma média pesada:

$$M = 0,5T + 0,35L + 0,15P.$$



Avaliação e notas

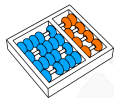
- (*T*) Quinzenalmente, serão aplicados (presencialmente) testes teóricos com duração de 30 minutos no final das aulas.
- (*L*) Ao longo do semestre serão orientadas listas de exercícios de programação, com prazo de entrega de uma semana.
- (*P*) Haverá um projeto final a ser desenvolvido em duplas.

A nota final será uma média pesada:

$$M = 0,5T + 0,35L + 0,15P.$$

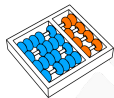
Contudo, se $T < 5$ ou $L < 5$ ou $P < 5$, a média final será:

$$M = \min\{T, L, P\}.$$



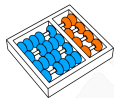
Aprovação e exame

- ▶ Para aprovar, precisa que $M \geq 5$ e o número de presenças seja no mínimo 75%.



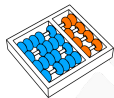
Aprovação e exame

- ▶ Para aprovar, precisa que $M \geq 5$ e o número de presenças seja no mínimo 75%.
- ▶ Se $2,5 \leq M < 5$ e tiver pelo menos 75% de presenças, então terá direito a exame.



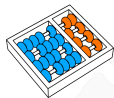
Aprovação e exame

- ▶ Para aprovar, precisa que $M \geq 5$ e o número de presenças seja no mínimo 75%.
- ▶ Se $2,5 \leq M < 5$ e tiver pelo menos 75% de presenças, então terá direito a exame.
- ▶ O exame (**E**) consistirá em uma prova teórica sobre todo o conteúdo do semestre.



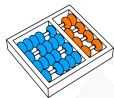
Aprovação e exame

- ▶ Para aprovar, precisa que $M \geq 5$ e o número de presenças seja no mínimo 75%.
- ▶ Se $2,5 \leq M < 5$ e tiver pelo menos 75% de presenças, então terá direito a exame.
- ▶ O exame (E) consistirá em uma prova teórica sobre todo o conteúdo do semestre.
- ▶ Após o exame a nota final será: $\max\{M, \min\{5, E\}\}$.



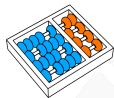
Uso de IA e fraudes

- ▶ É proibido o uso de Ferramentas de IA Generativa.



Uso de IA e fraudes

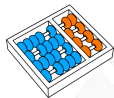
- ▶ É proibido o uso de Ferramentas de IA Generativa.
- ▶ É proibido qualquer tipo de fraude em atividades avaliativas.



Uso de IA e fraudes

- ▶ É proibido o uso de Ferramentas de IA Generativa.
- ▶ É proibido qualquer tipo de fraude em atividades avaliativas.

Se detectados fraude ou o uso de IA, a nota do semestre será **zero**, reprovando a disciplina.

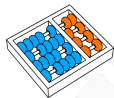


Uso de IA e fraudes

- ▶ É proibido o uso de Ferramentas de IA Generativa.
- ▶ É proibido qualquer tipo de fraude em atividades avaliativas.

Se detectados fraude ou o uso de IA, a nota do semestre será **zero**, reprovando a disciplina.

- ▶ Caso se arrependa de realizar fraude ou usar IA, contate imediatamente o professor, que terá só zerada a atividade em questão.

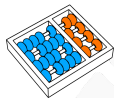


Uso de IA e fraudes

- ▶ É proibido o uso de Ferramentas de IA Generativa.
- ▶ É proibido qualquer tipo de fraude em atividades avaliativas.

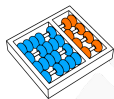
Se detectados fraude ou o uso de IA, a nota do semestre será **zero**, reprovando a disciplina.

- ▶ Caso se arrependa de realizar fraude ou usar IA, contate imediatamente o professor, que terá só zerada a atividade em questão.
- ▶ O arrependimento só vale se for antes de ser detectado pelo professor ou monitores.



Ferramentas online

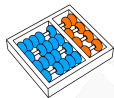




Ferramentas online



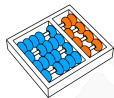
► [Google Sala de Aula.](#)



Ferramentas online



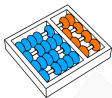
- ▶ [Google Sala de Aula.](#)
- ▶ [Página da disciplina.](#)



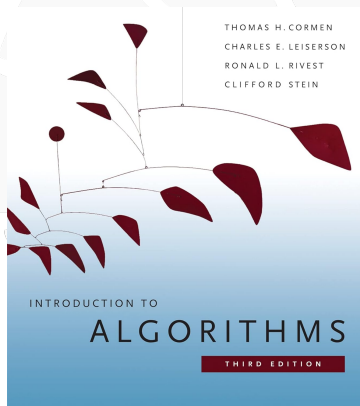
Ferramentas online



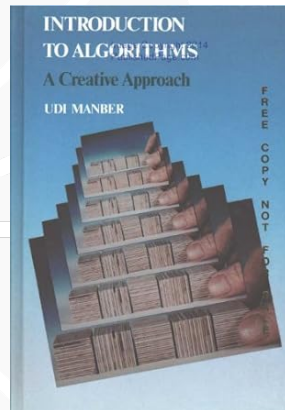
- ▶ [Google Sala de Aula.](#)
- ▶ [Página da disciplina.](#)
- ▶ [beecrowd](#) (chave: **MXeliWO**).



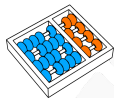
Bibliografia principal



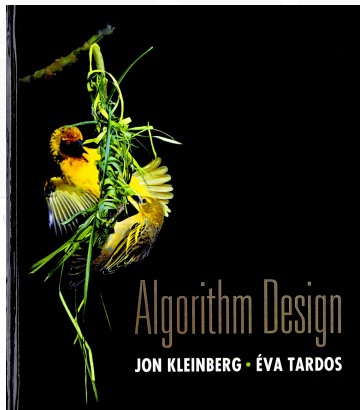
T. Cormen, C. Leiserson, R. Rivest, C. Stein. *"Introduction to Algorithms"*. MIT Press (MA); 3rd ed. (2009).



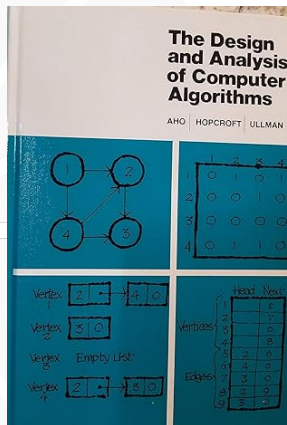
U. Manber. *"Introduction to Algorithms: A Creative Approach"*. Addison-Wesley. (1989).



Bibliografia complementar



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



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

APRESENTAÇÃO DA DISCIPLINA

MC458 - Projeto e Análise de
Algoritmos I

Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

08/25

00



UNICAMP

