Complexidade de tempo

Por que realizar análise de complexidade de algoritmos?

Por que realizar análise de complexidade de algoritmos?

- Estimar duração

Por que realizar análise de complexidade de algoritmos?

- Estimar duração
- Estimar tamanho máximo da entrada

Por que realizar análise de complexidade de algoritmos?

- Estimar duração
- Estimar tamanho máximo da entrada
- Comparar algoritmos

Por que realizar análise de complexidade de algoritmos?

- Estimar duração
- Estimar tamanho máximo da entrada
- Comparar algoritmos
- Partes do código para melhorar

Por que realizar análise de complexidade de algoritmos?

- Estimar duração
- Estimar tamanho máximo da entrada
- Comparar algoritmos
- Partes do código para melhorar
- Escolher algoritmo

Por que realizar análise de complexidade de algoritmos?

- Estimar duração
- Estimar tamanho máximo da entrada
- Comparar algoritmos
- Partes do código para melhorar
- Escolher algoritmo
  - Tamanho até 6 não precisa otimizar

Por que realizar análise de complexidade de algoritmos?

- Estimar duração
- Estimar tamanho máximo da entrada
- Comparar algoritmos
- Partes do código para melhorar
- Escolher algoritmo
  - Tamanho até 6 não precisa otimizar
  - Tamanho $\sim$1000, garanta que não é exponencial

Por que realizar análise de complexidade de algoritmos?

- Estimar duração
- Estimar tamanho máximo da entrada
- Comparar algoritmos
- Partes do código para melhorar
- Escolher algoritmo
    - Tamanho até 6 não precisa otimizar
    - Tamanho $\sim$1000, garanta que não é exponencial
    - Tamanho $\sim$1 bilhão, garanta tempo linear

Complexidades de tempo e espaço são funções

Complexidade de tempo $T(n)$: total de operações executadas pelo algoritmo, em função do tamanho da entrada n.

Complexidades de tempo e espaço são funções

Complexidade de tempo $T(n)$: total de operações executadas pelo algoritmo, em função do tamanho da entrada n.

- Assumimos que as operações gastam o mesmo tempo.

Complexidades de tempo e espaço são funções

Complexidade de tempo $T(n)$: total de operações executadas pelo algoritmo, em função do tamanho da entrada n.

- Assumimos que as operações gastam o mesmo tempo.
- Então basta contar o total de operações.

Complexidades de tempo e espaço são funções

Complexidade de tempo $T(n)$: total de operações executadas pelo algoritmo, em função do tamanho da entrada n.

- Assumimos que as operações gastam o mesmo tempo.
- Então basta contar o total de operações.
- Ex.: $T(n) = 3n^2 + 7n + 23$.

Complexidades de tempo e espaço são funções

Complexidade de tempo $T(n)$**:** total de operações executadas pelo algoritmo, em função do tamanho da entrada n.

- Assumimos que as operações gastam o mesmo tempo.
- Então basta contar o total de operações.
- Ex.: $T(n) = 3n^2 + 7n + 23$.

Complexidade de espaço $S(n)$**:** total de memória necessária para executar o algoritmo, em função do tamanho da entrada n.

Algoritmo P_1 demanda $T_1(n) = n^4$ operações, e P_2 demanda $T_2(n) = 2^n$ operações.

Algoritmo P_1 demanda $T_1(n) = n^4$ operações, e P_2 demanda $T_2(n) = 2^n$ operações.
A máquina executa 10^6 operações por segundo.

Algoritmo P_1 demanda $T_1(n) = n^4$ operações, e P_2 demanda $T_2(n) = 2^n$ operações.
A máquina executa 10^6 operações por segundo.
Se $n = 1000$, qual é o tempo de execução destes algoritmos?

Algoritmo P_1 demanda $T_1(n) = n^4$ operações, e P_2 demanda $T_2(n) = 2^n$ operações.
A máquina executa 10^6 operações por segundo.
Se $n = 1000$, qual é o tempo de execução destes algoritmos?

$$\text{Velocidade (oper/seg)} = \frac{\text{Complexidade (oper)}}{\text{Tempo (seg)}}$$

Algoritmo P_1 demanda $T_1(n) = n^4$ operações, e P_2 demanda $T_2(n) = 2^n$ operações.
A máquina executa 10^6 operações por segundo.
Se $n = 1000$, qual é o tempo de execução destes algoritmos?

$$\text{Velocidade (oper/seg)} = \frac{\text{Complexidade (oper)}}{\text{Tempo (seg)}}$$

$T_1(n)/10^6 = (1000)^4/10^6 = 10^6$ segundos (11,6 dias).

Algoritmo P_1 demanda $T_1(n) = n^4$ operações, e P_2 demanda $T_2(n) = 2^n$ operações.
A máquina executa 10^6 operações por segundo.
Se $n = 1000$, qual é o tempo de execução destes algoritmos?

$$\text{Velocidade (oper/seg)} = \frac{\text{Complexidade (oper)}}{\text{Tempo (seg)}}$$

$T_1(n)/10^6 = (1000)^4/10^6 = 10^6$ segundos (11,6 dias).

$T_2(n)/10^6 = 2^{1000}/10^6 \approx 1{,}07 \times 10^{295}$ segundos ($1{,}24 \times 10^{290}$ dias).
Obs.: número de partículas no universo é cerca de 10^{80}.

Algoritmo P_1 demanda $T_1(n) = n^4$ operações, e P_2 demanda $T_2(n) = 2^n$ operações.
A máquina executa 10^6 operações por segundo.
Se $n = 1000$, qual é o tempo de execução destes algoritmos?

$$\text{Velocidade (oper/seg)} = \frac{\text{Complexidade (oper)}}{\text{Tempo (seg)}}$$

$T_1(n)/10^6 = (1000)^4/10^6 = 10^6$ segundos (11,6 dias).

$T_2(n)/10^6 = 2^{1000}/10^6 \approx 1{,}07 \times 10^{295}$ segundos ($1{,}24 \times 10^{290}$ dias).
Obs.: número de partículas no universo é cerca de 10^{80}.

Dizemos que P_1 é *viável* (polinomial), e que P_2 é *inviável* (exponencial).

Ignorando detalhes

Ignoramos constantes multiplicativas.

$$T(n) = \underbrace{\overbrace{a_1}^{\text{constantes}} \cdot f_1(n)}_{\text{termo}} + \underbrace{\overbrace{a_2}^{} \cdot f_2(n)}_{\text{termo}} + \underbrace{\overbrace{a_3}^{\text{multiplicativas}} \cdot f_3(n)}_{\text{termo}}$$

Ex.: $T(n) = \cancel{3}n^2 + \cancel{7}n + \cancel{23} \approx n^2 + n.$

Ignorando detalhes

Ignoramos constantes multiplicativas.

$$T(n) = \underbrace{\overbrace{a_1}^{\text{constantes}} \cdot f_1(n)}_{\text{termo}} + \underbrace{\overbrace{a_2}^{} \cdot f_2(n)}_{\text{termo}} + \underbrace{\overbrace{a_3}^{\text{multiplicativas}} \cdot f_3(n)}_{\text{termo}}$$

Ex.: $T(n) = \cancel{3}n^2 + \cancel{7}n + \cancel{23} \approx n^2 + n.$

Por que?

Ignorando detalhes

Ignoramos constantes multiplicativas.

$$T(n) = \underbrace{\overbrace{a_1}^{\text{constantes}} \cdot f_1(n)}_{\text{termo}} + \underbrace{\overbrace{a_2}^{} \cdot f_2(n)}_{\text{termo}} + \underbrace{\overbrace{a_3}^{\text{multiplicativas}} \cdot f_3(n)}_{\text{termo}}$$

Ex.: $T(n) = \cancel{3}n^2 + \cancel{7}n + \cancel{23} \approx n^2 + n.$

Por que?

- Dependem do modelo

Ignorando detalhes

Ignoramos constantes multiplicativas.

$$T(n) = \underbrace{\overbrace{a_1}^{\text{constantes}} \cdot f_1(n)}_{\text{termo}} + \underbrace{\overbrace{a_2}^{} \cdot f_2(n)}_{\text{termo}} + \underbrace{\overbrace{a_3}^{\text{multiplicativas}} \cdot f_3(n)}_{\text{termo}}$$

Ex.: $T(n) = \cancel{3}n^2 + \cancel{7}n + \cancel{23} \approx n^2 + n.$

Por que?

- Dependem do modelo
- Muito trabalho e pouco significativo (n^2 vs n^3 mais significativo que n^2 vs $3n^2$)

Ignorando detalhes

Ignoramos constantes multiplicativas.

$$T(n) = \underbrace{\overbrace{a_1}^{\text{constantes}} \cdot f_1(n)}_{\text{termo}} + \underbrace{\overbrace{a_2}^{\text{multiplicativas}} \cdot f_2(n)}_{\text{termo}} + \underbrace{\overbrace{a_3} \cdot f_3(n)}_{\text{termo}}$$

Ex.: $T(n) = \cancel{3}n^2 + \cancel{7}n + \cancel{23} \approx n^2 + n$.

Por que?

- Dependem do modelo
- Muito trabalho e pouco significativo (n^2 vs n^3 mais significativo que n^2 vs $3n^2$)

Obs.: Constantes são importantes em algoritmos muito utilizados.

Ignorando detalhes

Ignoramos termos de menor crescimento.

$$T(n) = \underbrace{a_1 \cdot f_1(n)}_{\text{termo}} + \underbrace{a_2 \cdot f_2(n)}_{\text{termo}} + \underbrace{a_3 \cdot f_3(n)}_{\text{termo}}$$

Ignorando detalhes

Ignoramos termos de menor crescimento.

$$T(n) = \underbrace{a_1 \cdot f_1(n)}_{\text{termo}} + \underbrace{a_2 \cdot f_2(n)}_{\text{termo}} + \underbrace{a_3 \cdot f_3(n)}_{\text{termo}}$$

Ex.: $T(n) = \cancel{3}n^2 + \cancel{7}n + \cancel{23} \approx n^2 + \cancel{n} \approx n^2$.

Ignorando detalhes

Ignoramos termos de menor crescimento.

$$T(n) = \underbrace{a_1 \cdot f_1(n)}_{\text{termo}} + \underbrace{a_2 \cdot f_2(n)}_{\text{termo}} + \underbrace{a_3 \cdot f_3(n)}_{\text{termo}}$$

Ex.: $T(n) = \cancel{3}n^2 + \cancel{7}n + \cancel{23} \approx n^2 + \cancel{n} \approx n^2.$

As duas simplificações são formalizadas pelas notações assintóticas (ex.: Θ, O).

Definição de tamanho da entrada

- **Tamanho da entrada:** quantidade de bits necessários para codificá-la.

Definição de tamanho da entrada

- **Tamanho da entrada:** quantidade de bits necessários para codificá-la.
- Quantos bits são necessário para codificar o valor inteiro positivo v?

Definição de tamanho da entrada

- **Tamanho da entrada:** quantidade de bits necessários para codificá-la.
- Quantos bits são necessário para codificar o valor inteiro positivo v?
  Como 2^{n-1} é o menor valor v com n bits,

$$2^{n-1} \leq v < 2^n$$

Definição de tamanho da entrada

- **Tamanho da entrada:** quantidade de bits necessários para codificá-la.
- Quantos bits são necessário para codificar o valor inteiro positivo v?
  Como 2^{n-1} é o menor valor v com n bits,

$$2^{n-1} \leq v < 2^n \quad \Rightarrow \quad n - 1 \leq \log_2 v < n$$

Definição de tamanho da entrada

- **Tamanho da entrada:** quantidade de bits necessários para codificá-la.
- Quantos bits são necessário para codificar o valor inteiro positivo v?
  Como 2^{n-1} é o menor valor v com n bits,

$$2^{n-1} \leq v < 2^n \quad \Rightarrow \quad n-1 \leq \log_2 v < n \quad \Rightarrow \quad n = \lfloor \log_2 v \rfloor + 1 \approx \log_2 v$$

Definição de tamanho da entrada

- **Tamanho da entrada:** quantidade de bits necessários para codificá-la.
- Quantos bits são necessário para codificar o valor inteiro positivo v?
  Como 2^{n-1} é o menor valor v com n bits,

$$2^{n-1} \leq v < 2^n \quad \Rightarrow \quad n - 1 \leq \log_2 v < n \quad \Rightarrow \quad n = \lfloor \log_2 v \rfloor + 1 \approx \log_2 v$$

Ex.: Entrada com valor 1 bilhão tem tamanho $n = \lfloor \log_2 10^9 \rfloor + 1 = 30$ bits.

Definição de tamanho da entrada

- **Tamanho da entrada:** quantidade de bits necessários para codificá-la.
- Quantos bits são necessário para codificar o valor inteiro positivo v?
  Como 2^{n-1} é o menor valor v com n bits,

$$2^{n-1} \leq v < 2^n \quad \Rightarrow \quad n-1 \leq \log_2 v < n \quad \Rightarrow \quad n = \lfloor \log_2 v \rfloor + 1 \approx \log_2 v$$

- Se recebe valor v como entrada e gasta $T(n) = v$, seu tempo de execução é linear?

Definição de tamanho da entrada

- **Tamanho da entrada:** quantidade de bits necessários para codificá-la.
- Quantos bits são necessário para codificar o valor inteiro positivo v?
  Como 2^{n-1} é o menor valor v com n bits,

$$2^{n-1} \leq v < 2^n \quad \Rightarrow \quad n-1 \leq \log_2 v < n \quad \Rightarrow \quad n = \lfloor \log_2 v \rfloor + 1 \approx \log_2 v$$

- Se recebe valor v como entrada e gasta $T(n) = v$, seu tempo de execução é linear?
  Não, pois $T(n) = v \approx 2^n$ (exponencial em n).

Considere os algoritmos:

- P_1: soma os V inteiros em um array.
- P_2: checa se um inteiro V é divisível por 2, por 3, ... até V.

Considere os algoritmos:

- P_1: soma os V inteiros em um array.
- P_2: checa se um inteiro V é divisível por 2, por 3, ... até V.

Os dois realizam $V - 1$ operações (adições ou divisões).

Considere os algoritmos:

- P_1: soma os V inteiros em um array.
- P_2: checa se um inteiro V é divisível por 2, por 3, ... até V.

Os dois realizam $V - 1$ operações (adições ou divisões).
P_1 é considerado *viável*, enquanto P_2 é considerado *inviável*. Por que?

Considere os algoritmos:

- P_1: soma os V inteiros em um array.
- P_2: checa se um inteiro V é divisível por 2, por 3, ... até V.

Os dois realizam $V - 1$ operações (adições ou divisões).
P_1 é considerado *viável*, enquanto P_2 é considerado *inviável*. Por que?

Supondo inteiros de 64 bits, o tamanho da entrada para P_1 vale $n = 64V$.

Considere os algoritmos:

- P_1: soma os V inteiros em um array.
- P_2: checa se um inteiro V é divisível por 2, por 3, ... até V.

Os dois realizam $V - 1$ operações (adições ou divisões).
P_1 é considerado *viável*, enquanto P_2 é considerado *inviável*. Por que?

Supondo inteiros de 64 bits, o tamanho da entrada para P_1 vale $n = 64V$.
Concluímos que $T_1(n) = V - 1 = \frac{1}{64}n - 1$ (linear).

Considere os algoritmos:

- P_1: soma os V inteiros em um array.
- P_2: checa se um inteiro V é divisível por 2, por 3, ... até V.

Os dois realizam $V - 1$ operações (adições ou divisões).
P_1 é considerado *viável*, enquanto P_2 é considerado *inviável*. Por que?

Supondo inteiros de 64 bits, o tamanho da entrada para P_1 vale $n = 64V$.
Concluímos que $T_1(n) = V - 1 = \frac{1}{64}n - 1$ (linear).

Em P_2 o tamanho da entrada é o número de bits de V, ou seja $n \approx \log_2 V$.

Considere os algoritmos:

- P_1: soma os V inteiros em um array.
- P_2: checa se um inteiro V é divisível por 2, por 3, ... até V.

Os dois realizam $V - 1$ operações (adições ou divisões).
P_1 é considerado *viável*, enquanto P_2 é considerado *inviável*. Por que?

Supondo inteiros de 64 bits, o tamanho da entrada para P_1 vale $n = 64V$.
Concluímos que $T_1(n) = V - 1 = \frac{1}{64}n - 1$ (linear).

Em P_2 o tamanho da entrada é o número de bits de V, ou seja $n \approx \log_2 V$.
Concluímos que $T(n)_2 = V - 1 \approx 2^n - 1$ (exponencial).

Definição de tamanho da entrada

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Definição de tamanho da entrada

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Quantidade de dígitos de um número inteiro v.

Definição de tamanho da entrada

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Quantidade de dígitos de um número inteiro v.
Quantidade de bits $\approx \log_2 v$
Quantidade de dígitos $\approx \log_{10} v$

$$\log_{10} v = \left(\frac{1}{\log_2 10}\right) \times \log_2 v$$

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Tupla $\langle x_1, x_2, \ldots, x_n \rangle$ com n elementos, cada um com aproximadamente C bits.

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Tupla $\langle x_1, x_2, \ldots, x_n \rangle$ com n elementos, cada um com aproximadamente C bits.

- Total de bits $\approx C \cdot n$.

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Tupla $\langle x_1, x_2, \ldots, x_n \rangle$ com n elementos, cada um com aproximadamente C bits.

- Total de bits $\approx C \cdot n$.
- Então podemos usar n como tamanho da entrada.

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Tupla $\langle x_1, x_2, \ldots, x_n \rangle$ com n elementos, cada um com aproximadamente C bits.

- Total de bits $\approx C \cdot n$.
- Então podemos usar n como tamanho da entrada.
- Ex.: vetor de inteiros, cada um com 32 bits.

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Tupla $\langle x_1, x_2, \ldots, x_n \rangle$ com n elementos, cada um com aproximadamente C bits.

- Total de bits $\approx C \cdot n$.
- Então podemos usar n como tamanho da entrada.
- Ex.: vetor de inteiros, cada um com 32 bits.
- Ex.: strings, cada caractere com 8 bits.

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Grafo com n vértices e m arestas.

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Grafo com n vértices e m arestas.

- Tamanho depende da implementação.

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Grafo com n vértices e m arestas.

- Tamanho depende da implementação.
- Array de arestas ocupa $2 \log_2 n$ bits por aresta, totalizando $m \times 2 \log_2(n)$ bits.

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Grafo com n vértices e m arestas.

- Tamanho depende da implementação.
- Array de arestas ocupa $2\log_2 n$ bits por aresta, totalizando $m \times 2\log_2(n)$ bits.
- Matriz de adjacências ocupa n^2 bits.

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Grafo com n vértices e m arestas.

- Tamanho depende da implementação.
- Array de arestas ocupa $2 \log_2 n$ bits por aresta, totalizando $m \times 2 \log_2(n)$ bits.
- Matriz de adjacências ocupa n^2 bits.
- Lista de adjacências ocupa $P \cdot n$ bits no array de ponteiros, e $(\log_2(n) + P) \times m$ nos elementos das listas, totalizando $P \cdot n + (\log_2(n) + P)m$ bits.

Definição de tamanho da instância

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits.

Ex.: Grafo com n vértices e m arestas.

- Tamanho depende da implementação.
- Array de arestas ocupa $2 \log_2 n$ bits por aresta, totalizando $m \times 2 \log_2(n)$ bits.
- Matriz de adjacências ocupa n^2 bits.
- Lista de adjacências ocupa $P \cdot n$ bits no array de ponteiros, e $(\log_2(n) + P) \times m$ nos elementos das listas, totalizando $P \cdot n + (\log_2(n) + P)m$ bits.
- Para simplificar, costuma-se usar m ou $n + m$ como tamanho do grafo.

Quando a complexidade depende da instância de entrada

Ex.: Ordenação por inserção: $\sim n$ para ordem crescente, e $\sim n^2$ para ordem decrescente.

Quando a complexidade depende da instância de entrada

Ex.: Ordenação por inserção: $\sim n$ para ordem crescente, e $\sim n^2$ para ordem decrescente.

Das 2^n entradas possíveis, qual considerar na análise?

Quando a complexidade depende da instância de entrada

Ex.: Ordenação por inserção: $\sim n$ para ordem crescente, e $\sim n^2$ para ordem decrescente.

Das 2^n entradas possíveis, qual considerar na análise?

- **Entrada típica:** para qual aplicação?

Quando a complexidade depende da instância de entrada

Ex.: Ordenação por inserção: $\sim n$ para ordem crescente, e $\sim n^2$ para ordem decrescente.

Das 2^n entradas possíveis, qual considerar na análise?

- **Entrada típica:** para qual aplicação?
- **Caso médio:** todas têm a mesma chance de ocorrer?

Quando a complexidade depende da instância de entrada

Ex.: Ordenação por inserção: $\sim n$ para ordem crescente, e $\sim n^2$ para ordem decrescente.

Das 2^n entradas possíveis, qual considerar na análise?

- **Entrada típica:** para qual aplicação?
- **Caso médio:** todas têm a mesma chance de ocorrer?
- **Pior caso:** é um caso típico?

Quando a complexidade depende da instância de entrada

Ex.: Ordenação por inserção: $\sim n$ para ordem crescente, e $\sim n^2$ para ordem decrescente.

Das 2^n entradas possíveis, qual considerar na análise?

- **Entrada típica:** para qual aplicação?
- **Caso médio:** todas têm a mesma chance de ocorrer?
- **Pior caso:** é um caso típico?

Costuma-se usar a análise de pior caso.

Complexidade de tempo de um problema

Menor número possível de operações para resolver uma instância do problema com tamanho n.

Complexidade de tempo de um problema

Menor número possível de operações para resolver uma instância do problema com tamanho n.

Ex.: Todo algoritmo de ordenação baseado em comparações realiza pelo menos $n \log_2 n$ operações. Mergesort está entre os melhores, pois o total de operações é da ordem de $n \log_2 n$.