

Projeto e Análise de Algoritmos

Projeto de algoritmos recursivos

Lehilton Pedrosa

Primeiro Semestre de 2020

Projeto de algoritmos recursivos

Resolvendo um problema com recursão

1. **instâncias pequenas:** resolver diretamente
2. **instâncias grandes:**
 - ▶ construir uma instância menor do mesmo problema
 - ▶ resolver essa subinstância recursivamente
 - ▶ encontrar uma solução para a instância original

Uma estratégia é:

1. encontrar e definir um **subproblema** adequado
2. supor que sabemos resolver instâncias menores
3. construir o algoritmo recursivo para o subproblema

O subproblema escolhido:

- ▶ não precisar ser o problema o problema original
- ▶ costuma ser uma variante ligeiramente **mais geral**

Este processo é análogo a **demonstrações por indução**

- ▶ **caso básico:** instâncias pequenas
- ▶ **caso geral:** instâncias grandes
 - ▶ a chamada recursiva corresponde à **hipótese de indução**
 - ▶ a construção da solução corresponde ao **passo da indução**

Algumas vantagens

- ▶ A correção vem do próprio processo indutivo
- ▶ A complexidade é expressa numa recorrência

Projeto de algoritmos recursivos

- ▶ Cálculo de polinômios

Problema

Dada uma sequência de números reais $a_n, a_{n-1}, \dots, a_1, a_0$ e um número real x , calcular o valor do polinômio

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0.$$

- ▶ este é um problema bem simples
- ▶ mas queremos fazer poucas adições e multiplicações

Hipótese de indução

Dada uma sequência de números reais a_{n-1}, \dots, a_1, a_0 e um número real x , sabemos calcular o valor de

$$P_{n-1}(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0.$$

- ▶ **Caso base:** $n = 0$.
 - ▶ devolvemos a_0
- ▶ **Caso geral:** $n \geq 0$.
 - ▶ calculamos $P_{n-1}(x)$ recursivamente
 - ▶ somamos $a_n x^n$ ao resultado obtido

CÁLCULO-POLINÔMIO(A, x)

```
1  se  $n = 0$  então
2       $y \leftarrow a_0$ 
3  senão
4       $A' \leftarrow a_{n-1}, \dots, a_1, a_0$ 
5       $y' \leftarrow$  CÁLCULO-POLINÔMIO( $A', x$ )
6       $xn \leftarrow 1$ 
7      para  $i \leftarrow 1$  até  $n$  faça
8           $xn \leftarrow xn \cdot x$ 
9           $y \leftarrow y' + a_n \cdot xn$ 
10 devolva  $y$ 
```

Complexidade da primeira tentativa

Seja $T(n)$ o número de operações aritméticas realizadas

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + (n+1) \text{ multiplicações} + 1 \text{ adição}, & n > 0. \end{cases}$$

Utilizando o método da iteração

$$\begin{aligned} T(n) &= \sum_{i=1}^n [(i+1) \text{ multiplicações} + 1 \text{ adição}] \\ &= (n+3)n/2 \text{ multiplicações} + n \text{ adições.} \end{aligned}$$

Observações:

- ▶ pode-se diminuir o número de multiplicações
- ▶ para isso, calculamos x^n com exponenciação rápida

Segunda solução indutiva

Melhorando

- ▶ no primeiro algoritmo, recalculamos potências de x .
- ▶ podemos reaproveitar o cálculo de x^{n-1}
- ▶ para isso, **reforçamos** a hipótese de indução

Hipótese de indução reforçada:

Sabemos calcular $P_{n-1}(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0$ e também o valor de x^{n-1} .

- ▶ podemos fazer hipóteses mais fortes sobre a recursão
- ▶ mas agora precisamos também devolver o valor de x^n
- ▶ i.e., precisamos resolver um problema **mais geral**

Segundo algoritmo

CÁLCULO-POLINÔMIO(A, x)

1 se $n = 0$ então

2 $y \leftarrow a_0$

3 $x_n \leftarrow 1$

4 senão

5 $A' \leftarrow a_{n-1}, \dots, a_1, a_0$

6 $y', x' \leftarrow \text{CÁLCULO-POLINÔMIO}(A', x)$

7 $x_n \leftarrow x * x'$

8 $y \leftarrow y' + a_n * x_n$

9 devolva y, x_n

Complexidade do segundo algoritmo

Para essa versão temos

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 2 \text{ multiplicações} + 1 \text{ adição}, & n > 0. \end{cases}$$

A solução da recorrência é

$$\begin{aligned} T(n) &= \sum_{i=1}^n (2 \text{ multiplicações} + 1 \text{ adição}) \\ &= 2n \text{ multiplicações} + n \text{ adições.} \end{aligned}$$

Terceira solução indutiva

Existem várias escolhas para a definição do subproblema

Outra hipótese de indução

Sabemos calcular $P'_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{n-2} \dots + a_1$.

- ▶ essa versão é mais simples
- ▶ e precisamos de menos operações

CÁLCULO-POLINÔMIO(A, x)

- 1 se $n = 0$ então
- 2 $y \leftarrow a_0$
- 3 senão
- 4 $A' \leftarrow a_{n-1}, \dots, a_1$
- 5 $y' \leftarrow \text{CÁLCULO-POLINÔMIO}(A', x)$
- 6 $y \leftarrow x * y' + a_0$
- 7 devolva y

Complexidade do algoritmo melhorado

Finalmente, temos

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 1 \text{ multiplicação} + 1 \text{ adição}, & n > 0. \end{cases}$$

A solução é

$$\begin{aligned} T(n) &= \sum_{i=1}^n (1 \text{ multiplicação} + 1 \text{ adição}) \\ &= n \text{ multiplicações} + n \text{ adições.} \end{aligned}$$

Esse algoritmo chamado de **regra de Horner**.

Projeto de algoritmos recursivos

- ▶ Fator de balanceamento em árvores binárias

Fator de balanceamento em árvores binárias

Vamos relembrar árvore binárias de busca

- ▶ cada nó v tem uma chave
- ▶ a subárvore esquerda tem chaves menores
- ▶ a subárvore direita tem chaves maiores

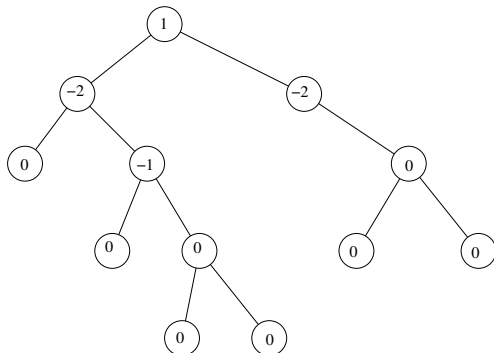
Definição

O **fator de balanceamento** (f.b.) de um nó v é a diferença entre as alturas da subárvores esquerda e direita.

Uma árvores binárias de busca é **balanceada** se todo nó v tem f.b. limitado.

- ▶ e.g., em uma árvore **AVL**, todo nó tem f.b. $-1, 0$ ou $+1$.
- ▶ uma árvore vazia tem f.b. igual a zero

Exemplo



Uma árvore e o f.b. de cada nó.

Calculando o fator de balanceamento

Problema

Dada uma árvore binária A com n nós, calcular os fatores de balanceamento de cada nó de A .

Vamos projetar o algoritmo indutivamente.

Hipótese de indução

Sabemos como calcular fatores de balanceamento de árvores com menos que n nós.

Uma dificuldade

- ▶ a definição de f.b. **não** é recursiva!
- ▶ o f.b. de um nó depende das alturas das subárvores
- ▶ mas sabemos apenas o f.b. das subárvores
- ▶ conclusão: é necessário uma h.i. mais forte!

Nova hipótese de indução:

Para um $n > 0$ qualquer, sabemos como calcular fatores de balanceamento e alturas de árvores com menos que n nós.

Encontrando a solução do subproblema

1. se $n = 0$, a árvore a árvore é vazia, então
 - ▶ o f.b. é igual 0.
 - ▶ a altura é igual 0.
2. se $n > 0$, a árvore tem pelo menos um nó
 - ▶ recursivamente, obtemos a altura da árvore esquerda h_e
 - ▶ do mesmo modo, obtemos a altura da árvore direita h_d
 - ▶ por definição, f.b. é $h_e - h_d$
 - ▶ e a altura é $\max(h_e, h_d) + 1$

Observações

- ▶ repare que não precisamos de f.b. para das subárvores
- ▶ mas obtemos a altura recursivamente
- ▶ de novo, fortalecer a h.i. simplificou o projeto do algoritmo

Pseudocódigo do algoritmo

FATORALTURA(A)

1 se $n = 0$ então

2 $f \leftarrow 0$

3 $h \leftarrow 0$

4 senão

5 $f_e, h_e \leftarrow \text{FATORALTURA}(A_e)$

6 $f_d, h_d \leftarrow \text{FATORALTURA}(A_d)$

7 $f \leftarrow h_e - h_d$

8 $h \leftarrow \max(h_e, h_d) + 1$

9 devolva f, h

Seja $T(n)$ o número de operações executadas no pior caso,

$$T(n) = \begin{cases} \Theta(1), & n = 0 \\ T(n_e) + T(n_d) + \Theta(1), & n > 0, \end{cases}$$

onde n_e, n_d são os números de nós das subárvores.

Estimando o pior caso

- ▶ O pior caso parece acontecer se $n_e = 0$ e $n_d = n - 1$
- ▶ Nesse caso, temos

$$\begin{aligned}T(n) &= T(n-1) + T(0) + \Theta(1) \\ &= T(n-2) + 2T(0) + 2\Theta(1) \\ &\quad \vdots \\ &= T(0) + nT(0) + n\Theta(1) \\ &= (n+1)\Theta(1) + n\Theta(1) = \Theta(n).\end{aligned}$$

Exercício: argumente que este é, de fato, um pior caso.

Projeto de algoritmos recursivos

- ▶ Problema da celebridade

Definição

Em um conjunto S de n pessoas, uma **celebridade** é alguém que é conhecido por todas as pessoas de S mas que não conhece ninguém.

- ▶ nem sempre existe uma celebridade
- ▶ mas só pode existir uma celebridade em S
- ▶ queremos saber se existe uma celebridade em S

O problema da celebridade

Formalizando o problema:

- ▶ queremos descrever o conhecimento entre n pessoas
- ▶ representamos isso com uma matriz binária $n \times n$
- ▶ $M[i,j] = 1$ se i conhece j e $M[i,j] = 0$ caso contrário.

Problema

Dado um conjunto de n pessoas e a matriz associada M , encontrar uma celebridade no conjunto se existir.

Observações

- ▶ para uma celebridade k , vale $M[i,k] = 1$ para todo i
- ▶ analogamente, vale $M[k,i] = 0$ para todo i
- ▶ um algoritmo simples verifica se cada pessoa é celebridade, usando $2n(n-1)$ operações

Argumento indutivo

Um argumento indutivo pode ser:

Hipótese de indução:

Sabemos encontrar uma celebridade em um conjunto de $n - 1$ pessoas ou decidir que não existe.

- ▶ se $n = 1$, a única pessoa no conjunto é uma celebridade
- ▶ se $n \geq 2$
 - ▶ represente o conjunto de pessoas como $S = \{1, 2, \dots, n\}$
 - ▶ podemos encontrar celebridade em $S' = \{1, 2, \dots, n - 1\}$
 - ▶ como isso ajuda na instância original?

Passo indutivo

Dois casos:

1. Existe celebridade k em S'
 - ▶ se $M[n, k] = 1$ e $M[k, n] = 0$, então k é celebridade em S
 - ▶ senão, n ainda poderia ser celebridade
 2. Não existe celebridade em S'
 - ▶ nesse caso, apenas n poderia ser celebridade
-
- ▶ em qualquer caso, precisamos verificar se n é celebridade
 - ▶ temos que verificar se $M[n, j] = 0$ e $M[j, n] = 1$ para $j < n$
 - ▶ também obtemos um algoritmo quadrático

Segunda tentativa

Suponha que s não é celebridade

- ▶ podemos fazer a chamada recursiva para $S' = S \setminus \{s\}$
- ▶ e não precisaríamos verificar s depois

Como encontrar alguém que **não** é celebridade?

- ▶ considere duas pessoas i e j
- ▶ se $M[i, j] = 1$, então i não é celebridade
- ▶ mas se $M[i, j] = 0$, então j não é celebridade

Vamos usar a mesma hipótese anterior

- ▶ i.e., consideramos o mesmo subproblema
- ▶ mas escolhemos a subinstância mais cuidadosamente

Encontrando uma celebridade

O algoritmo devolve NIL se não houver celebridade.

```
CELEBRIDADE( $S, M$ )
1  se  $|S| = 1$  então
2      seja  $k$  a única pessoa em  $S$ 
3  senão
4      sejam  $i, j$  duas pessoas em  $S$ 
5      se  $M[i, j] = 1$ 
6          então  $s \leftarrow i$ 
7          senão  $s \leftarrow j$ 
8       $S' \leftarrow S \setminus \{s\}$ 
9       $k \leftarrow \text{CELEBRIDADE}(S', M)$ 
10     se  $k = \text{NIL}$  ou  $M[s, k] = 0$  ou  $M[k, s] = 1$ 
11         então  $k \leftarrow \text{NIL}$ 
12     devolva  $k$ 
```


Exemplo 4 - Complexidade

Seja $T(n)$ o número de operações executadas

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ T(n-1) + \Theta(1), & n > 1. \end{cases}$$

A solução da recorrência é

$$\sum_1^n \Theta(1) = n\Theta(1) = \Theta(n).$$

Projeto de algoritmos recursivos

- ▶ Subsequência consecutiva máxima

Subsequência consecutiva máxima (SCM)

Problema:

Dada uma sequência $X = x_1, x_2, \dots, x_n$ de números reais encontrar uma **subsequência consecutiva** $Y = x_i, x_{i+1}, \dots, x_j$, onde $1 \leq i, j \leq n$, cuja soma seja máxima.

Exemplos:

$$X = [4, 2, -7, 3, 0, -2, 1, 5, -2] \quad \text{Resp: } Y = [3, 0, -2, 1, 5]$$

$$X = [-1, -2, 0] \quad \text{Resp: } Y = [0] \text{ ou } Y = []$$

$$X = [-3, -1] \quad \text{Resp: } Y = []$$

Hipótese de indução

Sabemos calcular a SCM de sequências de comprimento $n - 1$.

- ▶ considere uma sequência $X = x_1, x_2, \dots, x_n$
- ▶ removendo x_n , obtemos uma sequência menor X'
- ▶ usando a h.i., obtemos uma SCM $Y' = x_i, x_{i+1}, \dots, x_j$ de X'

Há três casos a analisar

1. $Y' = \emptyset$: se $x_n \geq 0$, faça $Y = x_n$, senão faça $Y = \emptyset$
2. $j = n - 1$: se $x_n \geq 0$, faça $Y = Y' \parallel x_n$, senão faça $Y = Y'$
3. $j < n - 1$: consideramos dois subcasos
 - ▶ se Y' for uma SCM de X , então basta fazer $Y = Y'$
 - ▶ caso contrário, x_n faz parte de uma SCM de X e fazemos $Y = x_k, x_{k+1}, \dots, x_n$, para algum $k \leq n - 1$.

- ▶ podemos executar os dois primeiros casos rapidamente
- ▶ mas para o último caso, a h.i. não fornece o valor de k
- ▶ mas, nesse caso, sabemos que uma SCM é

$$Y = x_k, x_{k+1}, \dots, x_{n-1}, x_n,$$

- ▶ ou seja, Y é um sufixo de X
- ▶ basta conhecer também o **sufixo de maior soma** de X'

Hipótese de indução reforçada

Hipótese de indução reforçada

Sabemos calcular a *SCM* e o sufixo de maior soma de sequências de comprimento $n - 1$.

- ▶ consideramos também sequência de tamanho $n = 0$
- ▶ nesse caso, o sufixo de maior soma e a *SCM* são vazias

Entrada e saída do subproblema

Entrada:

- ▶ um inteiro n
- ▶ n números reais $X = x_1, x_2, \dots, x_n$

Saída:

- ▶ inteiros i, j, k tais que
 - ▶ $Y = x_i, x_{i+1}, \dots, x_j$ é uma SCM de X
 - ▶ $S = x_k, x_{k+1}, \dots, x_n$ é um sufixo de soma máxima de X
- ▶ números reais $MaxSeq, MaxSuf$ tais que
 - ▶ a soma de Y é $MaxSeq$
 - ▶ a soma de S é $MaxSuf$

SCM(X, n):

```
1  se  $n = 0$ 
2    então  $i, j, k \leftarrow 1, \text{MaxSeq}, \text{MaxSuf} \leftarrow 0$ 
3  senão
4     $i, j, k, \text{MaxSeq}, \text{MaxSuf} \leftarrow \text{SCM}(X, n - 1)$ 
5     $\text{MaxSuf} \leftarrow \text{MaxSuf} + x_n$ 
6    se  $\text{MaxSuf} < 0$ 
7      então  $k \leftarrow n + 1, \text{MaxSuf} \leftarrow 0$ 
8    se  $\text{MaxSuf} > \text{MaxSeq}$ 
9      então  $i \leftarrow k, j \leftarrow n, \text{MaxSeq} \leftarrow \text{MaxSuf}$ 
10  devolva  $i, j, k, \text{MaxSeq}, \text{MaxSuf}$ 
```

O tempo de execução $T(n)$ de SCM é

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ T(n-1) + \Theta(1), & n > 1. \end{cases}$$

A solução desta recorrência é

$$\sum_1^n \Theta(1) = n\Theta(1) = \Theta(n).$$

Divisão e conquista

Projeto de algoritmos por divisão e conquista

Relembre que um algoritmo de **divisão e conquista**

- ▶ divide uma instância do problema considerado em partes
- ▶ combina as várias soluções parciais

Algumas características

- ▶ normalmente criamos pelo menos duas subinstâncias
- ▶ elas podem ser muito menores que a instância original

Algoritmo genérico

DIVISAO-CONQUISTA(x)

- 1 se a instância x é suficientemente pequena então
- 2 retorne SOLUCAO(x)
- 3 senão
- 4 decomponha x em instâncias menores x_1, x_2, \dots, x_k
- 5 para i de 1 até k faça
- 6 $y_i \leftarrow$ DIVISAO-CONQUISTA(x_i)
- 7 combine as soluções y_i para obter uma solução y
- 8 retorne y

Divisão e conquista

- ▶ Exponenciação rápida

Problema:

Calcular a^n , para todo real a e inteiro $n \geq 0$.

Um estratégia recursiva simples

- ▶ se $n = 0$:
 - ▶ devolva $a^0 = 1$
- ▶ se $n > 0$:
 - ▶ suponha que sabemos calcular a^{n-1}
 - ▶ calculamos $a^n = a^{n-1} \cdot a$

Algoritmo com recursão simples

EXPONENCIACAO(a, n)

```
1  se  $n = 0$  então
2     $an \leftarrow 1$ 
3  senão
4     $an' \leftarrow \text{EXPONENCIACAO}(a, n - 1)$ 
5     $an \leftarrow an' * a$ 
6  retorne  $an$ 
```


Complexidade da recursão simples

Seja $T(n)$ o número de operações executadas

$$T(n) = \begin{cases} \Theta(1), & n = 0 \\ T(n-1) + \Theta(1), & n > 0, \end{cases}$$

A solução da recorrência é

$$T(n) = \Theta(1) + \sum_{i=1}^n \Theta(1) = \Theta(n).$$

Pergunta: o algoritmo é linear no tamanho da entrada?

- ▶ o número de bits para representar n é $m = \lfloor \log_2 n \rfloor + 1$
- ▶ então o tempo do algoritmo é $\Theta(2^m)$
- ▶ que é **exponencial** no tamanho da entrada

Usando divisão e conquista

Utilizamos a mesma hipótese

Hipótese de indução

Suponha que, para qualquer inteiro $n > 0$ e real a , sei calcular a^k , para todo $k < n$.

Mas agora dividimos a potência em várias partes

- ▶ se $n = 0$:
 - ▶ devolva $a^0 = 1$
- ▶ se $n > 0$:
 - ▶ calculamos recursivamente $a^{\lfloor \frac{n}{2} \rfloor}$
 - ▶ podemos calcular a potência da seguinte forma

$$a^n = \begin{cases} (a^{\lfloor \frac{n}{2} \rfloor})^2, & \text{se } n \text{ é par} \\ a \cdot (a^{\lfloor \frac{n}{2} \rfloor})^2, & \text{se } n \text{ é ímpar} \end{cases}$$

Exponenciação com divisão e conquista

EXPONENCIACAO-DC(a, n)

1 se $n = 0$ então

2 $an \leftarrow 1$

3 senão

4 $an' \leftarrow \text{EXPONENCIACAO-DC}(a, \lfloor \frac{n}{2} \rfloor)$

5 $an \leftarrow an' \cdot an'$

6 se n é ímpar então

7 $an \leftarrow an \cdot a$

8 retorne an

O tempo de execução é dado por

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + c_2, & n > 0, \end{cases}$$

A solução da recorrência é $T(n) = \Theta(\log n)$

- ▶ esse algoritmo é linear no número de bits da entrada!

Divisão e conquista

- ▶ Busca binária

Problema:

Dado um vetor de números ordenado A com n elementos e número x , determinar alguma posição i tal que $A[i] = x$, ou decidir que x não está no vetor.

- ▶ se utilizássemos uma recursão simples, teríamos um algoritmo linear (qual seria a hipótese de indução?)
- ▶ vamos decompor o vetor em duas partes com cerca de **metade** dos elementos

Como representar subvetores?

Entrada:

- ▶ vetor A com n números
- ▶ índice e do elemento mais à esquerda do subvetor
- ▶ índice d do elemento mais à direita do subvetor
- ▶ número x

Saída:

- ▶ se x estiver no vetor, índice i tal que $A[i] = x$
- ▶ senão, NIL

BUSCA-BINARIA(A, e, d, x)

1 se $e > d$ então

2 $i \leftarrow \text{NIL}$

3 senão

4 $i \leftarrow \lfloor \frac{e+d}{2} \rfloor$

5 se $A[i] > x$ então

6 $i \leftarrow \text{BUSCA-BINARIA}(A, e, i - 1, x)$

7 se $A[i] < x$ então

8 $i \leftarrow \text{BUSCA-BINARIA}(A, i + 1, d, x)$

9 retorne i

O tempo de execução é:

$$T(n) = \begin{cases} c_1, & n = 1 \\ T(\lceil \frac{n}{2} \rceil) + c_2, & n > 1, \end{cases}$$

Resolvendo-se a recorrência, temos

$$T(n) = \Theta(\log n)$$

- ▶ esse algoritmo é melhor que a busca linear
- ▶ e se o vetor **não** estivesse ordenado, qual seria melhor?

Divisão e conquista

- ▶ Um exemplo real

Problema de um mestrando do IC

Entrada:

- ▶ m listas L_i com n_i inteiros ordenados do menor para maior
- ▶ um número k

Saída:

- ▶ k tuplas (a_1, a_2, \dots, a_m) com **menores somas**

Observação:

- ▶ a soma de uma tupla (a_1, a_2, \dots, a_m) é $a_1 + a_2 + \dots + a_m$
- ▶ também poderíamos usar outra operação associativa

Algoritmo trivial

Um algoritmo simples é testar todas as tuplas

1. Enumerar todas as tuplas
2. Ordenar as tuplas em ordem de soma usando MERGE-SORT
3. Selecionar as k primeiras

Esse algoritmo é inviável

- ▶ existem $m = n_1 \cdot n_2 \cdot \dots \cdot n_m$ tuplas
- ▶ o tempo de execução é $O(m \log m)$
- ▶ mas cada n_i é da ordem de **dezenas de milhares**
- ▶ e temos algumas **dezenas de listas**

Resolvendo com divisão e conquista

Podemo fazer melhor com divisão e conquista:

- ▶ caso base:
 - ▶ $m = 1$
 - ▶ $m = 2$ também!
- ▶ caso geral:
 - ▶ **divisão**: dividimos em sub-somas
 - ▶ **conquista**: é o mesmo problema, mas para $m = 2$!

Vamos tentar fazer juntos no quadro.