

Provás Antigas - PAA

Sophia Carrozza

Analise as assertivas a seguir, assinalando V, se verdadeiras, ou F, se falsas.

- (v)  $2^{n-1} = O(2^n)$
- (d)  $2^{n+1} = O(2^n)$
- (f)  $2^{3n} = O(4^n)$
- (v)  $2^{3n} = O(8^n)$
- (v)  $10000n^2 + 10000n + n\log n = O(n^3)$
- ( )  $\log_3 n = O(\log n)$

A ordem correta, de cima para baixo, das respostas destas assertivas é:

- (a) V - F - F - V - V - V
- ~~(b)~~ V - V - F - V - V - V
- (c) V - F - V - V - F - V
- (d) V - V - V - F - V - V
- (e) F - F - F - V - V - F

C

$$2^{n-1} \leq C \cdot 2^n$$

$$C \geq 2^{n-1} \div 2^n$$

$$C \geq 2^{-1}$$

$$C \geq \frac{1}{2}$$

$$2^{n+1} \leq C \cdot 2^n$$

$$C \geq 2^{n+1} \div 2^n$$

$$C \geq 2^1$$

$$C \geq 2$$

$$2^{3n} \leq C \cdot 4^n$$

$$C \geq 2^{3n} \div 2^{2n}$$

$$C \geq 2^n$$

$$\lim_{n \rightarrow \infty} 2^n = \infty$$

Logo  $\not\models$  C

$$f(n) \underset{n \rightarrow \infty}{\sim} g(n)$$

a)  $2^{n+1} = O(2^n)$

$2^{n+1} \leq C \cdot 2^n, \forall n \geq n_0$

$\frac{2^{n+1}}{2^n} \leq C$

definição de  $O$ :  $f(n) = O(g(n))$  SSE:  
 $\exists c > 0, m \geq 0 \mid f(m) \leq c \cdot g(m), \forall n \geq m_0$

$$\frac{2^n \cdot 2^{-1}}{2^n} \leq c$$

$$2^{-1} \leq c$$

✓ faz que a afirmação é verdadeira. Sempre que  $c \geq 1/2$ , a afirmativa é verdadeira.

\* A definição afirma que, se finita  
 existe uma constante  $c > 0$  e um valor  $n_0$  tal que  $f(n) \leq c \cdot g(n)$ ,  
 $\forall n \geq n_0$ , então ela satisfaz a desigualdade para todos os valores de  $n_0 > 0$ .

b)  $2^{n+1} = O(2^n)$

$2^{n+1} \leq C \cdot 2^n$

$\frac{2^n \cdot 2}{2^n} \leq C$

$2 \leq C$

$C \geq 2$  satisfaz a definição. ✓

c)  $2^{3n} = O(4^n)$

$2^{3n} \leq C \cdot 4^n$

$\frac{2^{3n}}{2^{2n}} \leq C$

$2^n \leq C$

$\lim_{C \rightarrow \infty} 2^n \Rightarrow +\infty$

✗ → não existe um valor fixo/constante que satisfaz a definição F

d)  $2^{3n} = O(8^n)$

$$2^{3n} \leq C \cdot 8^n$$

$$\frac{2^{3n}}{2^{3n}} \leq C$$



$C \leq C$  *existe un valor constante finito que satisface la definición*

e)  $1000n^2 + 1000n + n \log n = O(n^3)$

$$1000n^2 + 1000n + n \log n \leq C \cdot n^3$$

$$\frac{1000n^2}{n^2} + \frac{1000n}{n^2} + \frac{n \log n}{n^2} \leq C$$

$$\frac{1000}{n} + \frac{1000}{n^2} + \frac{\log n}{n^2}$$



f)  $\log_3 n = O(\log n)$

$$\log_3 n = \frac{\log n}{\log 3}$$

$$\log_3 n \leq C \cdot \log n$$

$$\frac{\log n}{\log 3} \leq C \cdot \log n$$

$$\frac{1}{\log 3} \cdot \frac{\log n}{\log 3} \leq C$$

$\frac{1}{\log 3} \leq C$

*existe una constante*

## QUESTÃO 1

(25 %)

Responda as perguntas justificando suas respostas por meios das definições.

- 2 a) (7 %) Se  $f(n) = O(g(n))$  e  $h(n) = O(g(n))$  então podemos afirmar que  $f(n)$  é igual a  $h(n)$ ? *não, abus de notação*
- 4 b) (8 %) Se  $f(n) = O(g(n))$  e  $g(n) = O(h(n))$  então podemos afirmar que  $f(n) = O(h(n))$ ? *transitividade*
- 10 c) (10 %) Sejam  $h_1(n) = O(f(n))$  e  $h_2(n) = O(g(n))$ . Podemos afirmar que  $h_1(n) + h_2(n) = O(\max\{f(n), g(n)\})$ ? *sim*

## QUESTÃO 2

(30 %)

Apresente a relação de recorrência das funções abaixo (com relação ao número de comparações), assim como a ordem de complexidade das funções. Você deverá provar as ordens de complexidade apresentadas. Além disto, discorra sobre quais problemas poderiam ser resolvidos com esses algoritmos. Faça as considerações que julgar necessárias.

7 a) (7 %)

```
int misterio(int n){  
    if (n<=1) return 1;  
    else return misterio(n-1);  
}
```

busca sequencial

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

10 b) (10 %)

```
int misterio(int n){  
    if (n<=1) return 1;  
    else return misterio(n/3);  
}
```

busca termino

$$T(n) = \begin{cases} T(\frac{n}{3}) + 1, & n > 1 \\ 1, & n = 1 \end{cases}$$

3 c) (13 %)

```
int misterio(int n){  
    if (n<=1) return 1;  
    else return misterio(n/4)+misterio(n/4);  
}
```

$$T(n) = \begin{cases} 2 \cdot T(\frac{n}{4}) + 1, & n > 1 \\ 1, & n = 1 \end{cases}$$

**f-a)**  $\left. \begin{array}{l} f(n) = O(g(n)) \\ h(n) = O(g(n)) \end{array} \right\} f(n) = h(n) ? \rightarrow$  Não, pois mesmo que a ordem de complexidade seja a mesma, ou seja, as duas fôrman um crescimento assimtótico limitado por uma constante (vezes  $g(n)$ ), isso não implica que  $f(n) = h(n)$ . Isso significaria que as duas funções fôrman exatamente igual p/ todos valores de  $n$  (algo não garantido pela definição da notação).

b)  $\left. \begin{array}{l} f(n) = O(g(n)) \\ g(n) = O(h(n)) \end{array} \right\} f(n) = O(h(n)) ?$

$$\hookrightarrow f(n) = O(O(h(n))) \Rightarrow f(n) = O(h(n))$$

Sim, pois ela segue a transitividade da notação  $O$ , a qual afirma que, se uma função está limitada assimtóticamente por outra que está limitada por uma intermediária, isso significa que ela está limitada pela intermediária também.

(exemplo: se  $f(n) = O(g(n))$  e  $g(n) = O(h(n))$ , então  $f(n) = O(h(n))$ )

$$f(n) \leq C_1 g(n) \leq C_1 C_2 h(n), \forall n \geq \max(n_1, n_2)$$

c)  $\left. \begin{array}{l} h_1(n) = O(f(n)) \\ h_2(n) = O(g(n)) \end{array} \right\} h_1(n) + h_2(n) = O(\max(f(n), g(n))) ?$

mesma definição

$\rightarrow$  Sim, pois somando duas funções assimtóticamente limitadas por 2 funções, o limite superior provêce como a função de menor crescimento assimtótico

Apresente a relação de recorrência das funções abaixo (com relação ao número de comparações), assim como a ordem de complexidade das funções. Você deverá provar as ordens de complexidade apresentadas. Além disto, discorra sobre quais problemas poderiam ser resolvidos com esses algoritmos. Faça as considerações que julgar necessárias.

27

a) (7%)

```
int misterio(int n){  
    if (n <= 1) return 1;  
    else return misterio(n-1);  
}
```

59

búca sequencial

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

chamada recursiva

comparação  
 $(if(n <= 1))$

5-23

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

fazer expansão iterativa

(substituirmos a definição da função recursiva várias vezes até identificar um padrão ou chegar ao caso base).

$$T(n-1) + 1$$

$$T(n-1) + 1 = (T(n-2) + 1) + 1$$

$$= ((T(n-3) + 1) + 1) + 1$$

$O(n)$

$$d = \frac{1}{(n-1) \cdot 2}$$

$$T(n) = (n-1) + d$$

## Equações de Recorrência

fatorial(n){

    if (n <= 1){  
        fat = 1;

    }

else {

fat = n \* fat(n-1);

}

$$T(n) = \begin{cases} d & n = 1 \\ c + T(n-1) & n > 1 \end{cases}$$

$$T(n) = C + \frac{T(n-1)}{C + \frac{C + T(n-2)}{C + \frac{C + C + T(n-3)}{\vdots}}}$$

$$\underbrace{C + C + C + \dots + C}_{(n-1)} + d$$

$$T(n) = c(n-1) + d$$

$O(n)$

serve para busca binária  
(quando se quer reduzir o problema exponencialmente)

b) (10 %)

```
int misterio(int n){  
    if (n<=1) return 1;  
    else return misterio(n/3);  
}
```

regr de log:  
 $\log_b a = x \rightarrow b^x = a$

$$T(n) = \begin{cases} T(n/3) + 1, & n > 1 \\ n = L, & 1 \end{cases}$$

\* caso base:  
 $T(n) = L \rightarrow n = L$

$$T(n) = n$$

$$T(n/3^k) + k \rightarrow \frac{1}{3^k} + k = 1$$

↓  
nunca vamos chegar exatamente em 1 (sempre vai ser decimal)  
 $= T(n/3^k) + k$

$$\frac{1}{3^k} = L \rightarrow 3^k = \frac{1}{L}$$

$$\log_3 L = -k \rightarrow O(\log(n))$$

c)

```
int misterio(int n){  
    if (n<=1) return 1;  
    else return misterio(n/4)+misterio(n/4);  
}
```

$$T(L) = 2T(n/4) + L$$

$$T(2) = 2 \cdot (2T(n/16) + 2) + 2$$

$$T(3) = 2 \cdot (2 \cdot (2T(n/64) + 3) + 2) + 3$$

$$T(n) = 2^n \cdot T(n/4^i) + i$$

$$T(n) = \begin{cases} T(n/4) + T(n/4) + L, & n > L \\ n = L, & L \end{cases}$$

$$T(n) = \begin{cases} 2 \cdot T(n/4) + L, & n > L \\ n = L, & L \end{cases}$$

$$T(1) \cdot T(2) \cdot T(3) \cdots T(n) =$$

chamada de função mas não pode, não  
vai chamando iterações

$$T(n) = \begin{cases} 2T(n/4) + 1, & n \geq 1 \\ n = L, & L \end{cases}$$

expansão  
telescópica

devemos  
derrotar  
a fórmula  
fechada

$$\begin{aligned} T(L) &= +1 \\ T(2) &= 2T(n/4) + 1 + 1 \\ T(3) &= 2T(n/4) \cdot 2T(n/4) + 1 + 1 + 1 \\ T(n) &= 2T(n/4) + 1 \end{aligned}$$

$$T(m) = 2T(n/4) + L$$

$$2 \cdot T(m/4) = 4T(n/16) + 1$$

nó rotina com  
verso depois  
de contate

caso genérico

$$2^i T(m/4^i) = 2^{i+1} T\left(\frac{m}{4^{i+1}}\right) + 2^i$$

$$T(m) = \sum_{i=0}^{\log_4(m)} 2^i \rightarrow i \in [0, ?]$$

conceitos em  $n$  e terminamos em 2  
(no 1 não chamamos + noda)

$$\frac{m}{4^i} = m ? \rightarrow i=0$$

$$\frac{m}{4^i} = 2 ? \rightarrow$$

$$m = 4^i \cdot 2 \rightarrow \frac{m}{2} = 4^i \rightarrow i = \log_4\left(\frac{m}{2}\right)$$

\* Formulas normales de PG

$$S_n = \frac{a(a^{m-1})}{a-1}$$

$$\log_{a^x} b = \frac{1}{x} \log_a b$$

$$\sum_{i=0}^{\log_2(n)} 2^i = 2^{\log_2^{1/2} - 1}$$

$$2^{\left[\frac{1}{2} \log_2^{n/2} - 1\right]}$$

$$\frac{2^{\left[\frac{1}{2} \log_2^{n/2}\right]}}{2} = 2^{\log_2\left(\frac{n}{4}\right)^{1/2}}$$

$$T(n) = \frac{1}{2} \cdot \frac{\sqrt{n}}{\sqrt{2}}$$

$$\frac{\sqrt{n}}{2\sqrt{2}} \leq c \cdot n \quad T(n) = O(n)$$

$$\lim_{n \rightarrow \infty} \frac{\sqrt{n}}{2\sqrt{n}} = \frac{1}{2\sqrt{2}}$$

exists

$$T(m) = 2T(m/4) + 1$$

~~$$2 \cdot T(m/4) \leq 2 \cdot 2T(m/4/4) + 2$$~~

~~$$\underbrace{2 \cdot 2 \cdot T(m/4/4)}_{2^i T(m/4^i)} \leq \underbrace{2 \cdot 2 \cdot 2T(m/4/4/4)}_{2^{i+1} T(m/4^{i+1})} + 4$$~~

$$T(L) = L$$

loop:

loop  $\rightarrow m$

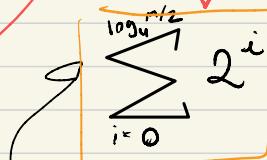
turno  $\rightarrow 2$  (no final chamavaos de nodes)

$$\frac{m}{4^i} = m \Rightarrow i=0$$

$$\frac{m}{4^0} = m \Rightarrow \frac{m}{1} = m$$

$$\frac{m}{4^i} = 2 \rightarrow \frac{m}{2} = 4^i \rightarrow \log_4 \frac{m}{2} = i$$

$$4^i = \frac{m}{2}$$



$$i \in [0, \frac{n}{2}]$$

raízes comuns

$$S_h = \frac{a_1(1 - r^m)}{1 - r}$$

$$a = L (1^{\text{a}} \text{ termo})$$

$r = 2$  (multiplicava de 2 em 2)

$$m = \log_4 \frac{m}{2} + 1$$

$$= \frac{L \cdot (L - 2^{\log_4 \frac{m}{2} + 1 - 1})}{L - 2 - 1}$$

$$= L - 2^{\log_4 \frac{m}{2}} \cdot (-1)$$

$$= 2^{\log_4 \frac{m}{2}} - L$$

$$= 2^{\log_2 \frac{m}{2}}$$

$$= \frac{1}{2} 2^{\log_2 \frac{m}{2}}$$

$O(n)$

$$\sqrt{\frac{n}{2}} \Leftarrow \frac{m}{2}^{1/2}$$

## PROPRIEDADES DOS LOGARITMOS

Logaritmando      Logaritmo  
 $\log_a b = x \rightarrow a^x = b$   
Base

LOGARITMO DO PRODUTO

$$\log_a(b \cdot c) = \log_a b + \log_a c$$

LOGARITMO DO QUOCIENTE

$$\log_a\left(\frac{b}{c}\right) = \log_a b - \log_a c$$

LOGARITMO DA POTÊNCIA

$$\log_a b^c = c \cdot \log_a b$$

LOGARITMO DE BASE COM POTÊNCIA

$$\log_{a^x} b = \frac{1}{x} \log_a b$$

MUDANÇA DE BASE

$$\log_b a = \frac{\log_c a}{\log_c b}$$



- a) (15 %) Seja  $h_1(n) = \mathcal{O}(f(n))$  e  $h_2(n) = \mathcal{O}(g(n))$ . Mostre que  $h_1(n) + h_2(n) = \mathcal{O}(\max\{f(n), g(n)\})$  em termos da definição da notação  $\mathcal{O}$ .
- b) (10 %)  $h(n) = \mathcal{O}(h(n))$ ? Justifique sua resposta em termos da definição da notação  $\mathcal{O}$ .

## QUESTÃO 2

(20 %)

Questões sobre tratabilidade:

- a) (10 %) Como provar que um problema X pertence a classe de problemas NP-Completo?
- b) (10 %) Discorra sobre reduções polinomiais entre dois problemas mostrando o grau de dificuldade relativo estes problemas.

## QUESTÃO 3

(25 %)

Dados um conjunto  $U$  de elementos, uma coleção  $S = \{S_1, S_2, \dots, S_m\}$  de subconjuntos de  $U$ .

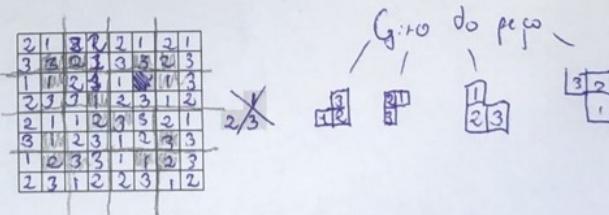
- Um subconjunto  $C \subseteq S$  é um set cover se a união de elementos de  $C$  é igual à  $U$ .
- Dados  $U$ ,  $S$ , e um inteiro  $k$ , existe uma coleção de tamanho  $\leq k$  destes conjuntos cuja união é igual à  $U$ ?

Mostre que VERTEX COVER pode ser reduzido polinomialmente para o SET COVER.

## QUESTÃO 4

(30 %)

Um **tromino** é um peça em formato de  $L$  formado por quadrados adjacentes 1-por-1, ilustrado na figura. O problema é **cobrir**, com *trominos*, qualquer tabuleiro de xadrez no formato  $2^n$ -por- $2^n$  com um quadrado faltando (em qualquer lugar do tabuleiro). Projete um algoritmo para que os trominos possam cobrir todos os quadrados do tabuleiro, exceto aquele marcado como faltante sem que haja sobre-posição.



**Crescimento Assintótico**  $\Rightarrow$  é uma forma de medir o comportamento de uma função quando a sua entrada se aproxima do infinito.

- a) (15 %) Seja  $h_1(n) = O(f(n))$  e  $h_2(n) = O(g(n))$ . Mostre que  $h_1(n) + h_2(n) = O(\max\{f(n), g(n)\})$  em termos da definição da notação O
- b) (10 %)  $h(n) = O(h(n))$ ? Justifique sua resposta em termos da definição da notação O.

a) Pela definição da notação O,  $f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$

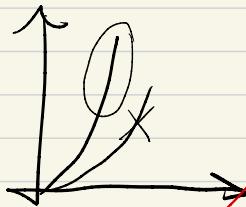
$$h_1(n) = O(f(n))$$

$$h_2(n) = O(g(n))$$

$$\exists c > 0, m \geq 0$$

se somarmos ambas as funções:

$$h_1(n) + h_2(n) \leq c \cdot f(n) + c \cdot g(n)$$



logo:

o crescente deixa uma soma determinada pelo maior entre  $f(n)$  e  $g(n)$  \*

ma análide assimptotica,  
o comportamento de uma  
soma de funções é governado  
pela função que cresce  
mais rapidamente à medida  
que  $n$  aumenta.

$$h_1(n) + h_2(n) \leq \max(f(n), g(n))$$

ex:  $h_1(n) + h_2(n) = n^2 + n$   
para valores grandes de  $n$ , o termo  $n^2$  domina a soma, porque cresce muito mais rápido do que  $n$ .  
(crescente assimptotic)

b)  $h(n) = O(h(n)) \rightarrow$  Não. Isto é falso, já que, pela definição da notação O:

$$\exists c > 0, m \geq 0, f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$



$$h(n) \leq c \cdot h(n) \Leftrightarrow$$

$$h(n) \leq 1 \cdot h(n) \checkmark$$

$$h(n) \leq 5 \cdot h(n)$$

vendidinho, pq  
o limite de  
 $\frac{h(n)}{h(n)}$   
sempre é constante

## QUESTÃO 2

(20 %)

Questões sobre tratabilidade:

- (10 %) Como provar que um problema X pertence a classe de problemas NP-Completo?
- (10 %) Discorra sobre reduções polinomiais entre dois problemas mostrando o grau de dificuldade relativo estes problemas.

a)

**Problema NP-Completo:** é um subconjunto de NP, o conjunto de todos os problemas de decisão cujas soluções podem ser verificadas em tempo polinomial.

NP → Problema em que até ainda não consegui criar um algoritmo para sua solução em tempo polinomial.

→ Pode ser que ele esteja em P (polinomial) e eu só não encontrei uma resposta boa pra ele, ou pode ser que ele não esteja em P.

Um problema Q é NP-completo se Q ∈ NP e todo outro problema de NP se reduz polinomialmente a Q.

$$\forall X \in NP \rightarrow X \text{ reduz } Q$$

Se X é NP-completo, podemos dizer que P = NP se e somente se X pertence a P.

Um problema Q é NP-difícil se todo outro problema de NP se reduz polinomialmente a Q.

Um problema Q é NP-completo se Q ∈ NP e Q é NP-difícil

a) Se provarmos que  $X$  é NP e que todo outro problema de NP se reduz polinomialmente a  $X$  (e NP-difícil), então ele é NP-Completo.

Ou seja:

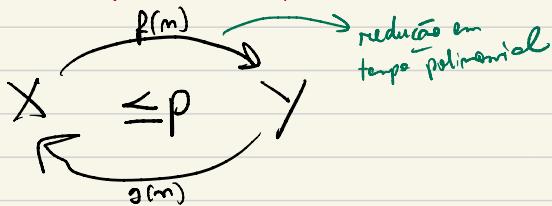
- mostrarmos que  $X$  está em NP

- enumerarmos todos os problemas de NP e fazermos redução polinomial de todos para  $X$  (mostrando que ele é NP-difícil)

\*anotação dmo  
no caderno!!!

ou seja, se suas soluções podem ser verificadas em tempo polinomial (dada uma solução  $X$ , é possível verificar sua veracidade em tempo polinomial)

b)



Se  $X$  se reduz para  $Y$ , significa que  $Y$  é pelo menos tão difícil quanto  $X$  (se resolvemos  $Y$  eficientemente, também conseguimos resolver  $X$ ).

$$X \leq P Y \text{ e } Y \leq P X \text{ então } X \equiv P Y$$

Damais forma, se  $X$  se reduz polinomialmente para  $Y$  e  $Y$  se reduz polinomialmente p/  $X$ ,  $X$  e  $Y$  são igualmente fáceis ou difíceis.

### QUESTÃO 3

(25 %)

Dados um conjunto  $U$  de elementos, uma coleção  $S = \{S_1, S_2, \dots, S_m\}$  de subconjuntos de  $U$ .

- Um subconjunto  $C \subseteq S$  é um set cover se a união de elementos de  $C$  é igual à  $U$ .
- Dados  $U$ ,  $S$ , e um inteiro  $k$ , existe uma coleção de tamanho  $\leq k$  destes conjuntos cuja união é igual à  $U$ ?

Mostre que VERTEX COVER pode ser reduzido polinomialmente para o SET COVER.

(NP completo)  $\rightarrow$  SATISFIABILITY, ou SAT  $\rightarrow$  primeiro problema classificado como pertencente à classe de complexidade NP-completo

**Satisfabilidade:** Dado uma fórmula booleana (resposta Vou F) em sua forma conjuntiva normal (CNF), determinar se existe uma determinada valoração para suas variáveis tal que ela satisfaça essa fórmula em questão.

**LITERAL**  $\rightarrow$  (termo)

Literal: Uma variável booleana ou sua negação.  $X$  or  $\bar{X}$

**cláusula**: Uma disjunção <sup>(COR)</sup> de literais/termos  $C = X_1 \vee \bar{X}_2 \vee X_3$

**fórmula booleana em CNF**: Uma conjunção <sup>(AND)</sup> de cláusulas  $\Phi = C_1 \wedge C_2 \wedge \dots \wedge C_3$

$$\Phi = (X_1 \vee \bar{X}_2 \vee X_3) \wedge (X_1 \vee X_2 \vee \bar{X}_3) \wedge (X_1 \vee X_2 \vee X_3)$$

**Definição do Problema - Decisão 3SAT**: Igualando ao SAT, mas cada cláusula tem 3 termos (ou literais), cada um envolvendo variáveis diferentes

↓ logo:

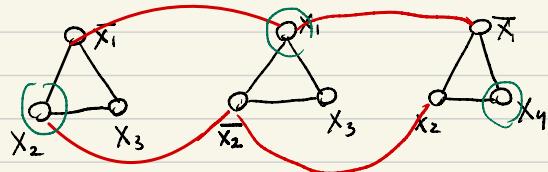
$$\Phi = (\bar{X}_1 \vee X_2 \vee X_3) \wedge (X_1 \vee \bar{X}_2 \vee \bar{X}_3) \wedge (X_1 \vee \bar{X}_2 \vee X_3) \wedge (\bar{X}_1 \vee \bar{X}_2 \vee \bar{X}_3)$$

**Claim:  $3SAT \leq P$  Conjunto Independente**

**Construction:**

$$\Phi = (\bar{X}_1 \vee X_2 \vee X_3) \wedge (X_1 \vee \bar{X}_2 \vee X_3) \wedge (\bar{X}_1 \vee \bar{X}_2 \vee X_3)$$

o problema de satisfabilidade 3-SAT pode ser reduzido em tempo polinomial ao problema de Conjunto Independente  
↓ exemplos



tempo polinomial  
 $O(n^2)$   
 $3n = O(n)$   
 $O(n^2)$   
 $O(n^2)$   
 $O(n^2)$

- 1- create 3 vertices for each clause
- 2- connect these 3 vertices to a triangle
- 3- connect every literal to each of its negation
- 4- set K equal to number of clauses.

$O(n^2)$  está dentro do tempo polinomial

- $X_2, X_1, X_4$
- $X_3, X_1, X_2$
- $\bar{X}_1, \bar{X}_2, X_4$

ou seja, esse "algoritmo" de redução  
de um problema a outro foi  
feito em tempo polinomial.

↳ proof: Let  $S$  be independent set of size  $K$ :

- $S$  must contain a vertex from each triangle.
- Set corresponding literals to true
- Truth assignment is consistent and satisfies  $\Phi$

Given a satisfying assignment for  $\Phi$ , pick one true literal from each clause corresponding set of vertices in  $G$  are an independent set of size  $K$ .

Um resumo mostra que existe uma correspondência entre a satisfatibilidade da fórmula e da existência de um conjunto independente de tamanho  $K$  no grafo  $G$ .

Direção 1: Se  $S$  é um conjunto independente de tamanho  $K$ , então  $\Phi$  é satisfatível.

- Um conjunto independente é um conjunto de vértices no qual nenhum par está conectado por uma aresta.
- Como cada triângulo no grafo representa uma cláusula, o conjunto independente deve conter exatamente um vértice de cada triângulo (pois escolher mais de um violaria a independência).
- O vértice escolhido dentro de cada triângulo corresponde a um literal que satisfaz a cláusula correspondente.
- Assim, os literais escolhidos formam uma atribuição que satisfaz todas as cláusulas da fórmula  $\Phi$ .

Direção 2: Se  $\Phi$  é satisfatível, então existe um conjunto independente de tamanho  $K$ .

- Suponha que existe uma atribuição satisfatória para a fórmula  $\Phi$ .
- Para cada cláusula, escolha um literal verdadeiro na atribuição.
- No grafo, selecione o vértice correspondente a esse literal dentro do triângulo da cláusula.
- Esse conjunto de vértices será independente (nenhum par estará conectado), pois:
  - Dentro de cada triângulo, apenas um vértice foi escolhido.
  - Literais complementares ( $x_i$  e  $\neg x_i$ ) não podem ser escolhidos simultaneamente devido à conexão entre eles.
- Como há  $K$  cláusulas, o conjunto independente terá exatamente tamanho  $K$ .

## Resumo

A construção e prova mostram que resolver o problema 3-SAT equivale a encontrar um conjunto independente no grafo construído. Essa redução demonstra que o problema Conjunto Independente é NP-difícil, já que o 3-SAT também é NP-completo.

Para verificar se uma redução é polinomial:

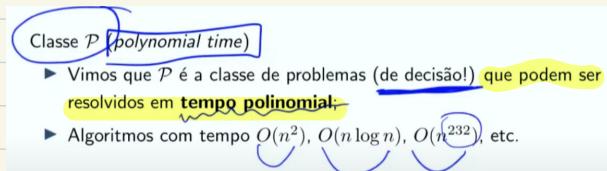
1. Liste todas as operações realizadas na transformação.
2. Estime a complexidade computacional de cada etapa com base no tamanho da entrada (número de variáveis ou cláusulas).
3. Certifique-se de que nenhuma etapa tenha complexidade maior que polinomial ( $O(n^k)$ , onde  $k$  é uma constante).

Se todas as etapas forem polinomiais, a redução está correta dentro desse critério.

# Classe de Problemas em Optimização

classe  $P \rightarrow$  para problemas de decisão (booleana - um problema de resposta sim ou não).

ex: Existe um clique em tal Grafo  $G = (V, E)$ ?  $\rightarrow$  Sim ou Não  
por isso verificando da por fórmula booleana!!!



classe  $NP \rightarrow$  nondeterministic polynomial time. Problemas de decisão em que um "resultado não" possui uma solução de tamanho polinomial que pode ser checada em tempo polinomial.

Uma regra: resolvendo o problema é muito difícil, mas tendo uma resposta em mãos, podemos checar ela em tempo polinomial

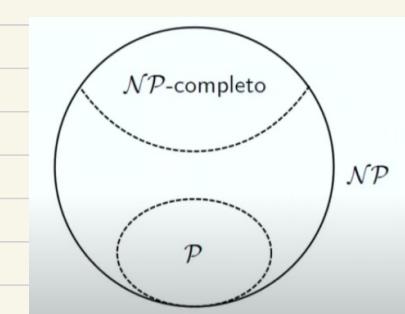
→ não sei um algoritmo eficiente pra resolvê-lo.

## Classe $NP$ (nondeterministic polynomial time)

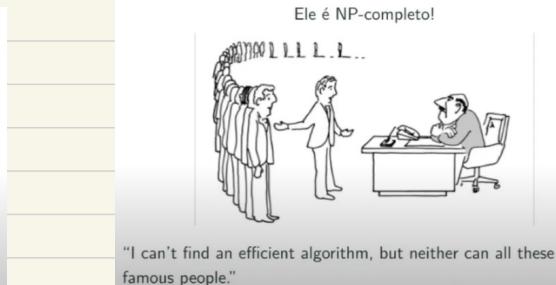
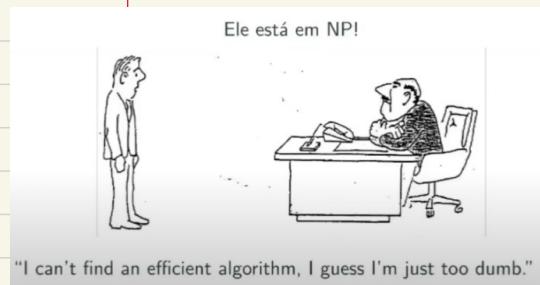
- ▶ Problemas de decisão para os quais qualquer resultado **sim** possui um certificado ("solução") de tamanho polinomial que pode ser checado em tempo polinomial;
- ▶ Assim, sempre que a decisão é **sim**, é possível verificar a resposta em tempo polinomial;
- ▶  $P \subset NP$ ;
- ▶ **Nondeterministic:** solucionados em tempo polinomial em uma Máquina de Turing não-determinística;

As máquinas não determinísticas

uma máquina que atua de forma aleatória e por um acaso pode achar a resposta ótima



**classe NP-completo:** Problema  $B$  que está em NP e, dado qualquer  $A$  em NP, eu consigo reduzir  $A$  em  $B$ .  
São os "mais difíceis em NP"



Dados  $A$  e  $B$  em NP, de modo que  $A \leq^{\text{redução}} B$ :

- se  $B \in P$ , então  $A \in P$
- se  $A \in$  NP-completo, então  $B \in$  NP-completo

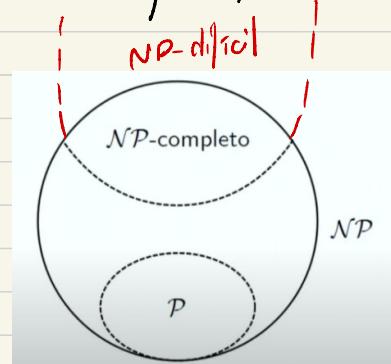
(ou seja, eu pego um problema que já sei que é NP-completo e transformo no meu)

**classe NP-difícil:** Problemas que não são tão difíceis quanto o problema mais difícil em NP.

- um NP-difícil não precisa estar em NP
- um problema NP-difícil que está em NP é um problema NP completo

• NP-difícil recorre ao oráculo

Um problema  $A$  é NP-difícil se existe um algoritmo de tempo polinomial para um problema  $B$  NP-completo quando esse algoritmo tem  $A$  como oráculo;



**Set-Cover:** Let  $U$  be a set of elements and  $C = \{S_1, S_2, \dots, S_n\}$  be a collection of subsets of  $U$ .

A set cover is a subcollection  $C' \subseteq C$  whose union is  $U$ .

$$\bigcup_{x \in C'} x = U$$

$$\bigcup_{x \in C'} x = U$$

$$U = \{x, \cancel{x}, \cancel{x}, \cancel{x}, 6, 7, \cancel{x}\}$$

$$\begin{array}{ll} S_1 = \{1, 2, 3\}, & S_2 = \{1, 4, 5, 8\} \\ S_3 = \{2, 3, 4\}, & S_4 = \{6, 7\} \\ S_5 = \{2, 4, 6\}, & S_6 = \{2, 3, 5, 8\} \end{array}$$

- $\{S_1, S_2, S_4, S_6\}$  is a set cover.
- $\{S_2, S_3, S_4\}$  is a set cover.
- $\{S_1, S_3, S_6\}$  is not a set cover. missing G and F.

**Set Cover Problem:** Given a set  $U$ , a collection  $S_1, S_2, \dots, S_n$  of subsets of  $U$ , and integer  $k$ , is there a set cover of  $U$  of size  $k$ ?

**Vertex Cover Problem:** Given a graph  $G$  and integer  $M$ , does  $G$  have a vertex cover

(cobertura de vértices = conjunto de vértices tal que cada aresta do grafo é incidente a pelo menos um vértice do conjunto.)

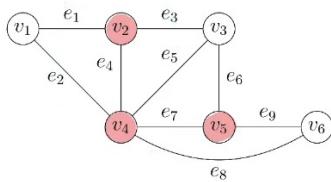


(SSC)

Se existe um problema NP-Completo L que é reduzível em tempo polinomial com uma máquina turing não determinística (racóculo), ela é NP-difícil.

1. Vertex Cover
  - Dado um grafo  $G = (V, E)$  e um inteiro  $k$ , o objetivo é determinar se existe um subconjunto de vértices  $C \subseteq V$  com  $|C| \leq k$  tal que cada aresta em  $E$  tem pelo menos um de seus extremos em  $C$ .
2. Set Cover
  - Dado um universo  $U$  e uma coleção de subconjuntos  $S = \{S_1, S_2, \dots, S_m\}$ , o objetivo é determinar se existe um subconjunto  $C \subseteq S$  com  $|C| \leq k$  tal que a união dos conjuntos em  $C$  cobre todos os elementos de  $U$ .

Given a graph  $G = (V, E)$ , let  $U = E$  and  $S_i$  correspond to vertex  $v_i \in V$ .



- $U = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$
- $S_1 = \{e_1, e_2\}$
- $S_2 = \{e_1, e_3, e_4\}$
- $S_3 = \{e_3, e_5, e_6\}$
- $S_4 = \{e_2, e_4, e_5, e_7, e_8\}$
- $S_5 = \{e_6, e_7, e_9\}$
- $S_6 = \{e_8, e_9\}$

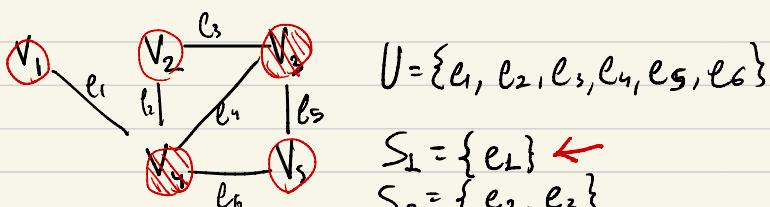
**Proposition 10.** The Vertex Cover Problem can be reduced to the Set Cover Problem in polynomial time.

*Proof.* Let graph  $G = (V, E)$  and integer  $M$  be an instance of the vertex cover problem. Let  $U = E$  and let  $S_i = \{e_j : v_i \text{ is incident on } e_j\}$  and  $C$  be the collection  $S_1, S_2, \dots, S_n$ . Graph  $G$  has a vertex of size  $M$  if and only if  $U$  has a set cover of size  $M$ . Mapping  $(G, M)$  to  $(U, C, M)$  is a reduction of the vertex cover problem to the set cover problem and constructing  $U$  and  $C$  takes  $O(mn)$  time which is a polynomial in  $m$  and  $n$ .  $\square$

Dados um conjunto  $U$  de elementos, uma coleção  $S = \{S_1, S_2, \dots, S_m\}$  de subconjuntos de  $U$ .

- Um subconjunto  $C \subseteq S$  é um set cover se a união de elementos de  $C$  é igual à  $U$  ← Set Cover
- Dados  $U$ ,  $S$ , e um inteiro  $k$ , existe uma coleção de tamanho  $\leq k$  destes conjuntos cuja união é igual à  $U$  ← Vertex Cover problem

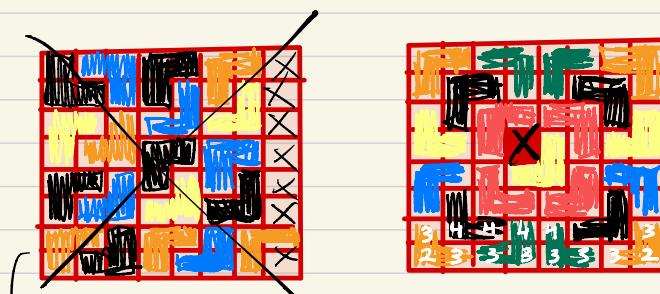
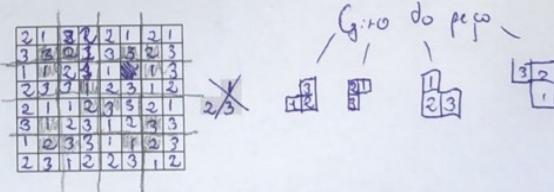
Mostre que VERTEX COVER pode ser reduzido polinomialmente para o SET COVER.



Modelando os conjuntos de arestas incidentes a cada vértice como um conjunto  $S_i$ , e realizando a união de uma coleção de  $K$  conjuntos, consegue-se chegar a uma resposta "sim" em tempo polinomial. Logo, Vertex Cover pode ser reduzido polinomialmente p/ o Set Cover.

$$S_1 \cup S_2 \cup S_4 = U | K = 3$$

Um **tromino** é uma peça em formato de L formado por quadrados adjacentes 1-por-1, ilustrado na figura. O problema é **cobrir**, com **trominos**, qualquer tabuleiro de xadrez no formato  $2^n$ -por- $2^n$  com um quadrado faltando (em qualquer lugar do tabuleiro). Projete um algoritmo para que os trominos possam cobrir todos os quadrados do tabuleiro, exceto aquele marcado como faltante sem que haja sobre-posição.



$$\Rightarrow 8 \ (2^3 \text{ por } 2^3) \rightarrow 8 \cdot 8 = 64 \rightarrow 63 \text{ preenchidos}$$

- Considera os quadrados como vértices e as adjacências como arestas
- Começa criando os Ls com 3 vértices sempre encostada a de menor grau de adjacência primeiro até finalizar os de grau 3.
- Depois Ls que contém vértices de grau 4,

- Deveremos dividir o quadro em 4 subquadros
- Colocar um L no centro de quadro, de forma que há um quadrado em cada subquadro, mas no que está faltando 1.
- Agora, chamamos recursivamente a pintura dos novos Ls, levando em conta que cada L deve ter o maior n de adjacência possível com os quadrados que já estão pintados (a pintar novos Ls em 3 quadrados que ainda não foram pintados)
- No quadro que tem o quadrado faltando, riga a mesma regra, mas proibindo a pintura do quadrado faltante.

You are given a board of size  $n \times n = 2^k \times 2^k$  which has one square missing (the missing square can be at any position). Board sizes are thus  $2 \times 2, 4 \times 4, 8 \times 8, 16 \times 16, 32 \times 32$ , etc. You are given as many trominos as you need.

A tromino is an object made up of three squares:



Show that no matter where the missing square is, the board can be tiled with trominoes!

Note that no two trominos can overlap and every square (except the missing one) must be tiled.

The smallest board is a  $2 \times 2$  board with one square missing. Any such  $2 \times 2$  board can be tiled using one tromino as shown below.



In our illustrations, gray squares represent the untiled board, a white square indicates the missing square, and trominos are blue.

It is easy to convince yourself that a  $2 \times 2$  board can be tiled with one tromino. It is a little harder for a  $4 \times 4$  board, but the number possible locations for the missing square is still quite small. What about a board of size  $1024 \times 1024 = 2^{10} \times 2^{10}$ ?

Recursion will help us understand how the tiling can be generated for an arbitrary sized board. You are given a board of size  $2^{k+1} \times 2^{k+1}$  which has one tile missing (**Figure 1**). The board consists of four boards of size  $2^k \times 2^k$  each (**Figure 2**). To figure out how to tile the board of size  $2^{k+1} \times 2^{k+1}$ , we will tile four boards of  $2^k \times 2^k$ .

Figure 1	Figure 2	Figure 3
Figure 4	Figure 5	Figure 6

We do this using recursion. Keep in mind that every time we tile a smaller board, the board **must** have one square marked as missing. We tile a board of size  $2^{k+1} \times 2^{k+1}$  as follows:

- Conceptually split the board into four boards of size  $2^k \times 2^k$ . (**See Figure 2**) One of the four smaller boards has a square missing. The other three boards have no squares missing.
- For the smaller board with the missing tile we make a recursive call to tile it. (**See Figure 3**)
- How do we tile the remaining three smaller boards? We don't know how to tile boards with no tile missing!
  - We remove one square from each of the remaining three smaller boards – the square removed is in the "center" of the original board. (**See Figure 4**)
  - We can now make three recursive calls, each call tiling one of the three smaller boards which now each have a missing tile. (**See Figure 5**)
  - The three center squares removed by us form a tromino and we place a tromino to cover them. (**See Figure 6**)
- Now we are done tiling a board of size  $2^{k+1} \times 2^{k+1}$ .

## QUESTÃO 3

(30 %)

Seja uma competição de ciclismo de velocidade por equipes, com que todos os membros estão divididos em  $m$  equipes. Um dos desafios das equipes é fazer um percurso mais rapidamente possível. Vou é o técnico de uma das equipes, e que o seu desafio é escolher adequadamente as bicicletas de cada membro da equipe. Para garantir a velocidade média máxima de cada ciclista saiba-se que quanto mais próximos os pesos da bicicleta e do ciclista, maior a velocidade que pode ser atingida pela bicicleta, e consequentemente, maior a velocidade média da par ciclista/bicicleta. Considere que a sua equipe possui  $n$  membros, com pesos  $a_1, a_2, \dots, a_n$ , e que existam bicicletas com pesos  $c_1, c_2, \dots, c_m$ .

Problema: o seu objetivo é escrever um algoritmo, em  $O(n^2)$ , para atribuir uma bicicleta a um ciclista de forma a tentar minimizar o tempo de percurso. Seu algoritmo é ótimo? Dê argumentos que mostre sua otimalidade ou um contra-exemplo que mostre que não é ótimo. Justifique suas escolhas e dize claro como chegou ao algoritmo com o custo desejado.

## QUESTÃO 4

(15 %)

Analice as assertivas a seguir, assinalando V se a assertiva for verdadeira, ou F, se a assertiva for falsa.

15

- (F) Algoritmos gulosos têm como principal característica a construção incremental da solução, por meio da otimização mísse de algum critério local, e por causa disto, não é possível obter soluções ótimas.
- (F) Algoritmos gulosos têm como principal característica a construção incremental da solução, por meio da otimização mísse de algum critério global, sempre obtendo soluções ótimas.
- (M) Algoritmos com ordem de complexidade  $O(n)$  também possuem ordem de complexidade  $O(n^2)$ .
- (F) Algoritmos com ordem de complexidade  $O(n)$  são sempre mais rápidos que algoritmos com ordem de complexidade  $O(n^2)$  para o mesmo tamanho de instância.
- (M) Encontrar elementos em um vetor, de forma exata, pode ser feita em  $O(n)$ . búzio seguiu
- (M) É possível fazer uma busca binária, desenvolvida por meio de recursão, e uma busca binária, desenvolvida por meio de uma estrutura de repetição, terem a mesma ordem de complexidade.

Qual será a ordem correta, de cima para baixo, das respostas destas assertivas? Justifique todas as suas respostas. Caso não tenha justificativa, o item será zerado.

- (a) V - V - V - F - F - V
- (b) F - F - V - F - F - V
- (c) F - F - F - V - F - V
- (d) F - F - V - F - V - V
- (e) V - F - V - F - V - F

3- - Bicicleta e Ciclista  $\rightarrow$  pesos iguais  $\rightarrow +$  velocidade média  
 - N membros  
 - Peso  $a_1, a_2, \dots, a_n$

► - Lista de tuplas ciclista - peso  
 - P/ cada bicicleta, calculemos a diferença entre  
 o peso dela e o peso de todos os ciclistas recorrendo  
 $(n \cdot n) \rightarrow O(n^2)$  e adicionemos a ciclista com a  
 menor diferença p/ essa bicicleta e adicionarmos à  
 lista.  
 - Assim, temos uma lista das bicicletas adequadas  
 p/ cada membro da equipe

30  
32  
53  
72  
31  
57  
70  
45

calcule  
diferença  
bicicleta p/  
cada um e  
atribuir aik

calculo dif.  $(n-1)$   
atribuição de menor dif.

► Considerando que a diferença de peso é o único fator determinante da velocidade de cada ciclista, a comparação da diferença entre pesos de todos os bicicletas maximiza a velocidade média de cada ciclista.

E como a velocidade média geral é a média das velocidades individuais (depende diretamente das velocidades), estavam também maximizada a velocidade média do time inteiro.

4-

Analise as assertivas a seguir, assinalando V se a assertiva for verdadeira, ou F, se a assertiva for falsa.

- F Algoritmos gulosos têm como principal característica a construção incremental da solução, por meio da otimização mísse de algum critério local, e por causa disto, não é possível obter soluções ótimas. → e para el him
- F Algoritmos gulosos têm como principal característica a construção incremental da solução, por meio da otimização mísse de algum critério global, sempre obtendo soluções ótimas.
- V Algoritmos com ordem de complexidade  $O(n)$  também possuem ordem de complexidade  $O(n^2)$ .  $n \leq n^2$  ✓ devido a definição do
- F Algoritmos com ordem de complexidade  $O(n)$  são sempre mais rápidos que algoritmos com ordem de complexidade  $O(n^2)$  para o mesmo tamanho de instância. X mas, para iso repete
- V Encontrar elementos em um vetor, de forma exata, pode ser feita em  $O(n)$ . Busca sequencial simpler ✓
- V É possível fazer uma busca binária, desenvolvida por meio de recursão, e uma busca binária, desenvolvida por meio de uma estrutura de repetição, terem a mesma ordem de complexidade.

Qual será a ordem correta, de cima para baixo, das respostas destas assertivas? Justifique todas as suas respostas. Caso não tenha justificativa, o item será zerado.

- (a) V - V - V - F - F - V
- (b) F - F - V - F - F - V
- (c) F - F - F - V - F - V
- F - F - V - F - V - V
- (e) V - F - V - F - V - F

da velocidade de entrada  
do algoritmo. A análise  
quantitativa é importante  
para explicar a grande  
ordem de complexidade  
de  $O(n^2)$

A busca binária e a com repetição  
podem possuir  $O(\log n)$

EXPLICAR DEPOIS



## QUESTÃO 2

(10 %)

Analise as assertivas a seguir, assinalando V, se verdadeiras, ou F, se falsas.

- (1) Seja um vetor ordenado com  $n$  elementos. Seja um algoritmo A para encontrar um elemento neste vetor usando uma estratégia de busca de binária. A relação de recorrência deste algoritmo será

$$T(n) = \begin{cases} T(n/2) + c, & \text{se } n > 1 \\ c, & \text{se } n = 1 \end{cases}$$

okay log<sub>2</sub> n

- (1) Seja um vetor com  $n$  elementos. Seja um algoritmo A para encontrar o maior elemento do vetor. Este algoritmo A, com ordem de complexidade  $O(n)$ , pode possuir a seguinte relação de recorrência

$$T(n) = \begin{cases} 2 \times T(n/2) + c, & \text{se } n > 1 \\ c, & \text{se } n = 1 \end{cases}$$

busca binária em  
dos lados okay

- (1) Seja um vetor com  $n$  elementos. Seja um algoritmo A para encontrar o maior elemento do vetor. Este algoritmo A, com ordem de complexidade  $O(\log n)$ , pode possuir a seguinte relação de recorrência

$$T(n) = \begin{cases} 2 \times T(n/2) + c, & \text{se } n > 1 \\ c, & \text{se } n = 1 \end{cases}$$

- (1) Sejam dois vetores ordenados  $A$  e  $B$  contendo  $m$  e  $n$  elementos, respectivamente. Seja o seguinte algoritmo: (i) selecione o menor elemento de  $A$  e compare com o menor elemento de  $B$ ; (ii) copie o menor elemento encontrado para um vetor  $C$  e remova este elemento do vetor que ele está presente; (iii) repita os procedimentos (i) e (ii) enquanto houver elemento ainda não copiado para  $C$ . O custo computacional deste algoritmo é  $O(m+n)$ .

(f)  $f'(n) = O(g'(n))$  e  $f''(n) = O(g''(n))$ , então  $f'(n) + f''(n) = O(\max\{g'(n), g''(n)\})$ .

(g)  $f(n) = O(g(n))$  se  $\exists c > 0, n_0 \geq 0$  tal que  $g(n) \leq c f(n), \forall n \geq n_0$ .

A ordem correta, de cima para baixo, das respostas destas assertivas é:

- (a) F - F - V - F - F - F
- (b) F - V - F - V - V - F
- (c) V - V - F - V - V - F
- (d) V - V - F - F - V - F
- (e) F - F - F - F - F - V

for (int i=0, i<(n+m); i++) {

x = seleção\_menor (A)

y = seleção\_menor (B)

if (x < y)

2-a) busca binária  $\rightarrow n \geq 1 \Rightarrow n/2$  ✓  
 $n = 1 \Rightarrow$  aceita

b) algoritmo p/ encontrar o maior  
elemento do vetor

$$T(n) = \begin{cases} 2 \cdot T(n/2) + C, & n \geq 1 \\ C, & n = 1 \end{cases}$$

$$2T(n/2) = 2 \cdot 2T(n/4) + 2C$$

$$\cancel{2} \cdot \cancel{2}T(n/4) = 2 \cdot 2 \cdot 2T(n/8) + \cancel{2} \cdot \cancel{2} \cdot C$$

$$2^i T(n/2^i) = 2^i T(n/2^{i+1}) + 2^i \cdot C$$

$$2T(1) = 2^{i+1}$$

TEOREMA MESTRE (versão simplificada): Sejam  $a \geq 1$ ,  $b > 1$  e  $k > 0$  constantes. Para  $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k)$  vale que

- (1) se  $a > b^k$ , então  $T(n) = \Theta(n^{\log_b a})$
- (2) se  $a = b^k$ , então  $T(n) = \Theta(n^k \log n)$
- (3) se  $a < b^k$ , então  $T(n) = \Theta(n^k)$

$$T(n) = 2T\left(\frac{n}{2}\right) + m$$

$$T(n) = 4T\left(\frac{n}{4}\right) + 5\sqrt{n}$$

$$T(n) = 5T\left(\frac{3n}{4}\right) + 10n^3$$

$$T(n) = 3T\left(\frac{n}{4}\right) + n \log n$$

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k)$$

$$\begin{matrix} \left(\frac{n}{2}\right) + m \\ / \quad \backslash \\ \frac{n}{2^2} + 2m \quad \frac{n}{2^2} + 2m \end{matrix}$$

$$\begin{matrix} a = 2 \\ b = 2 \\ k = 1 \end{matrix}$$

$$\begin{matrix} 2 > 2^1 & \times \\ 2 = 2^1 & \checkmark \end{matrix}$$

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

Q) Sejam dois vetores ordenados  $A$  e  $B$  contendo  $m$  e  $n$  elementos, respectivamente. Seja o seguinte algoritmo: (i) selecione o menor elemento de  $A$  e compare com o menor elemento de  $B$ ; (ii) copie o menor elemento encontrado para um vetor  $C$  e remova este elemento do vetor que ele está presente; (iii) repita os procedimentos (i) e (ii) enquanto houver elemento ainda não copiado para  $C$ . O custo computacional deste algoritmo é  $O(m+n)$ .

*processo 1*

```

função (A, B, C) {
    if (A != null OR B != null) { return C }
    a = A[0]
    b = B[0]
    if (a < b) {
        C += a ; A = A - a
    } else { C += b ; B = B - b }
    return (A, B, C)
}
  
```

$$7 \cdot m + 7 \cdot n + 1 \quad \text{caso predomina renomear}$$

$$\frac{11}{m+n}$$

*todos os elementos de A e B  
(para ordená-los em C), fazendo  
duas operações  $m+n$  vezes  
 $O(m+n)$*

*Um dia, V.*

Enquanto  $f'(n) = O(g'(n))$  e  $f''(n) = O(g''(n))$ , então  $f'(n) + f''(n) = O(\max\{g'(n), g''(n)\})$ . ou  
 (.)  $f(n) = O(g(n))$  se  $\exists c > 0, n_0 \geq 0$  tal que  $g(n) \leq cf(n), \forall n \geq n_0$ .

$$\exists c > 0, n_0 > 0 \mid f(n) \leq g(n) \quad \forall n \geq n_0$$

$$f'(n) + f''(n) \leq g'(n) + g''(n)$$

$$f(n) = 0$$

*analisando pelo crescimento  
assintótico, a ordem maior  
sempre predominará a determinação  
do crescimento da soma  
que da outra parcerá a ser  
insignificante perante a da*

Revise aqueles condicões de  $X \leq_p Y$ , etc

\*  $X \in \text{NP-Completo}$ :

i)  $\in \text{NP}$

ii)  $\nexists Y \in \text{NP-C} \mid Y \leq_p X$

existe

provando que podemos

reduzir  $Y$  para  $X$

porque aí, provamos  
que o  $X$  é tão difícil  
quanto  $Y$ . Portanto, se  
também é NP-Completo

**Clique**  $\rightarrow$  Para  $G(V, E)$  e um número  $K$ , queremos saber se existe um subconjunto de  $K$  vértices que formam um clique (todos os vértices do subconjunto estão conectados entre si).

**3-SAT**  $\rightarrow$  Fórmula Normal Cognitiva  $\rightarrow$  queremos saber se há uma atribuição de valores verdadeiros que satisfaz toda a fórmula

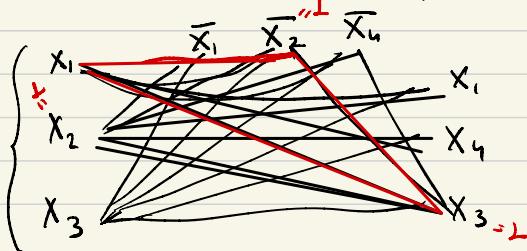
$\overbrace{(X_1 \vee X_2 \vee \bar{X}_3)}^{\text{clique}} \wedge \overbrace{(\bar{X}_1 \vee \bar{X}_2 \vee \bar{X}_4)}^{\text{clique}} \wedge \overbrace{(X_1 \vee X_4 \vee X_3)}^{\text{clique}} = L$  Clique é um subgrafo

se vai existir clique  
entre cláusulas diferentes

não forma conjuntos vazios em cada  
cláusula e conecta cada vértice aos  
outros se eles não formam um vértice

caixas de  $G$  com  
condensabilidade  $\geq K$

conjuntos que  
se conectam  
(tripartido)

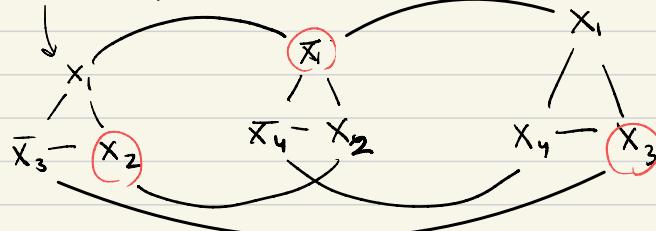


- vértice que cada vértice  
é conectado a todos  
os outros do subgrafo em  
tempo polinomial  
provando NP

$\text{3-SAT} \Leftrightarrow \text{CI}$

$$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_4) \wedge (x_1 \vee x_4 \vee x_3)$$

↓ um componente convexo triangular



$$\text{CI} = \{x_2, \bar{x}_1, x_3\}$$

$$P = m + n \quad (\text{o todo})$$

$$T(p) = \begin{cases} T(p-1) + P, & p > 1 \\ 0, & p = 1 \rightarrow \text{a ultima} \end{cases}$$

vai chamar  
 fd de novo mod  
 copia resto  
 um elemento

$$\text{gámm: } \frac{n(a_1 + a_n)}{2}$$

$$\sum_{i=0}^{p-2} (p-i)$$

$$p \cdot \sum_{i=0}^{p-2} i - \sum_{i=0}^{p-2} i$$

$$p(p-1) = \frac{(p-2)(p-3)}{2}$$

$$p^2 - p - \frac{(p^2 - 2p - 3p + p)}{2}$$

Apresente a relação de recorrência das funções abaixo (com relação ao número de comparações), assim como complexidade das funções. Você deverá provar as ordens de complexidade apresentadas.

( ) O menor custo computacional da função misterio é  $O(n)$

```
int misterio(int n){  
    if (n <= 1) return 1;  
    else return misterio(n/2)+misterio(n);  
}
```

( ) O menor custo computacional da função misterio2 é  $O(\log n)$

```
int misterio2(int n){  
    if (n <= 1) return 1;  
    else return misterio2(n/3);  
}
```

visível

( ) O custo computacional da função misterio3 é  $O(\log n)$

```
int misterio3(int n){  
    if (n <= 1) return 1;  
    else return misterio3(n/2)+misterio3(n/2);  
}
```

busca binária  
até o final

( ) O custo computacional da função misterio4 é  $O(n)$

```
int misterio4(int n){  
    int a = 0;  
    for (int i = 0; i < 10; i++)  
        for (int j = 0; j < n; j++)  
            a++;  
    return a;  
}
```

A ordem correta, de cima para baixo, das respostas destas assertivas é:

- (a) F - F - V - F
- (b) V - V - F - V
- (c) V - F - V - V
- (d) V - V - V - F
- (e) V - V - V - V

C

MUSICA PRA VER

A RESOLUÇÃO "

Dostoyevsky

```

int misterio(int n){
    if (n <= 1) return 1;
    else return misterio(n/2)+misterio(n);
}

```

$$\begin{cases} T(n/2) + C, & n > 1 \\ C, & n = 1 \end{cases}$$

↓ expansão telescópica

duplicar de  $\theta(n)$ :

$$T(n) = T(n/2) + C$$

$$T(n/2) = T(n/4) + C$$

⋮

$$T\left(\frac{n}{2^i}\right) = T\left(\frac{n}{2^{i+1}}\right) + C$$

$$C = 1/n_0 = 2/R = c \log_2 n$$

$$L \cdot \log_2/2 + 1 \leq (1 \cdot \log_2 2 + 1) \cdot \log_2 n$$

$$L \cdot L + 1 \leq (L \cdot 0,3 + 1) \cdot 0,3$$

$$2 \leq 0,39 X$$

$$\frac{n}{2^i} = n \Rightarrow i=0$$

$$\frac{n}{2^i} = 2 \Rightarrow n = 2 \cdot 2^i$$

$$n/2 = 2^i$$

$$\log_2 n/2 = i$$

$$i = \log_2 n - \log_2 2$$

$$i = \log_2 n - 1$$

$$\sum_{i=0}^{\log_2 n} C$$

$\log_2 n - 0$  vezes

caso base

$$T(n) = C(\log_2 n - 0) + 1$$

$$T(n) = C(\log_2 n) + 1$$

$$C \cdot \log_2 n + 1 \leq K \cdot \log_2 n$$

$$\frac{C \cdot \log_2 n + 1}{\log_2 n} \leq K$$

$$\frac{C \cdot \log_2 n \cdot \log_2 n + 1}{\log_2 n} \leq K$$

$$\frac{C \cdot \log_2 n + 1}{\log_2 n} \leq K$$

constante finita  $\Rightarrow C \cdot \log_2 n + 1$

quando o valor final de  $i$  é considerado como 2, temos que somar o caso base no somatório!!!

de  $T(2)$   
e  $T(1)$

new case por ex,  
não tem necessidade de  
parar em  $T(2)$ , já que  
não caso as repetições  
sao sempre os mesmos  
(C). Então, temos que adicionar ao  
somatório inicialmente

$n = 1 \rightarrow n = 2^i$   
 $\log_2 n = i$

o caso base  $T(1)$ .

O menor custo computacional da função misterio2 é:

```
int misterio2(int n){
    if (n <= 1) return 1;
    else return misterio2(n/3);
}
```

$$\begin{cases} T(n/3) + L, n > L \\ L, n = L \end{cases}$$

$$\log_b x = a \Leftrightarrow b^a = x$$



$$T(n) = T(n/3) + L$$

$$T(n/3) = T(n/9) + L$$

$$\frac{n}{3^i} = n$$

$$T(n/3^i) = T(n/3^{i+1}) + L$$

$$S_n = \frac{a_1(r^2 - 1)}{r - 1}$$

$$3^i = L$$

$$\frac{n}{3^i} = L$$

$$n = L \cdot 3^i$$

$$\log_3 n = i$$

$$\left[ \sum_{i=0}^{\log_3 n} L \right] + 2^{\log_3 r}$$

$$T(m) = L \log_3 (n) - O + L$$

$$T(m) = \log_3 (n) + L$$

$$O(\log n)$$

$$\log n \leq C \cdot (\log_3 (n) + 1)$$

$$\log n \leq C \cdot \frac{\log n}{\log 3} + 1$$

que é uma constante finita

$$\frac{\log_3 \cdot \log n}{\log 3} \leq C + 1$$

que é um valor de C existente entre 1 e 3 a depender

$$\log_3 - 1 \leq C$$

que é entre 1 e 3 a depender

$$c) T(n) = \begin{cases} 2T(n/2) + C, & n > 1 \\ C, & n = 1 \end{cases}$$

$$T(n) = 2T(n/2) + C$$

$$2T(n/2) = 2 \cdot 2T(n/4) + C$$

$$2^i T(n/2^i) = 2^{i+1} 2T(n/2^{i+1}) + C$$

que cada r se pone (?)

$$PG = \frac{a_1(r^n - 1)}{r - 1}$$

$$T(2) = 2T(1) + C$$

$$T(1) = 2^1 C$$

$$\frac{n}{2^i} = 1 \Rightarrow i = \log_2(n)$$

$$\sum_{i=0}^{\log_2 n} 2^i \cdot C$$

$$T(n) = C \cdot \left( \frac{2^{\log_2 n + 1} - 1}{2 - 1} \right)$$

$\downarrow$

$$\frac{2n - 1}{1}$$

$$T(n) = C(2n - 1)$$

$$C(2n - 1) \leq c \cdot n$$

Prove que:

- Seja  $h_1(n) = \mathbf{O}(f(n))$  e  $h_2(n) = \mathbf{O}(g(n))$ . Mostre que  $h_1(n) + h_2(n) = \mathbf{O}(\max\{f(n), g(n)\})$
- Seja  $h_1(n) = \mathbf{O}(f(n))$  e  $h_2(n) = \mathbf{O}(g(n))$ . Mostre que  $h_1(n) \times h_2(n) = \mathbf{O}(f(n) \times g(n))$
- Seja  $h_1(n) = \mathbf{O}(f(n))$  e  $c$  uma constante positiva. Mostre que  $h_1(n) \times c = \mathbf{O}(f(n))$

a)  $h_1(n) = \mathbf{O}(f(n)), \forall n \geq n_1$   
 $h_2(n) = \mathbf{O}(g(n)), \forall n \geq n_2$

$$h_1(n) + h_2(n) \leq C_1 \cdot f(n) + C_2 \cdot g(n)$$

$$h_1(n) + h_2(n) \leq C_1 C_2 (f(n) + g(n)), \forall n \geq \max(n_1, n_2)$$

$$h_1(n) + h_2(n) \leq C_1 C_2 (f(n) + g(n))$$

$$f(n) \leq \max\{f(n), g(n)\}$$

$$g(n) \leq \max\{f(n), g(n)\}$$

considerando o maior das  $M_0$ , temos uma notação de ambos os definidores, basta determinar a complexidade da função com maior crescimento assimétrico, pois ela dominará o comportamento da soma das funções.

faz que relacionemos um  $M_0$  que englobe ambos os definidores, basta determinar a complexidade da função com maior crescimento assimétrico, pois ela dominará o comportamento da soma das funções.

b)  $h_1(n) = \mathbf{O}(f(n))$   
 $h_2(n) = \mathbf{O}(g(n))$

$$h_1(n) \cdot h_2(n) \leq C_1 \cdot f(n) \cdot C_2 \cdot g(n)$$

$$h_1(n) \cdot h_2(n) \leq C_1 \cdot C_2 \cdot f(n) \cdot g(n)$$

$C = C_1 \cdot C_2$  é uma constante positiva  
 $N_0 = \max\{N_1, N_2\}$  para englobar ambos as funções

$$\underbrace{h_1(n) \cdot h_2(n)}_{\text{ou seja, o produto } h_1 \cdot h_2 \text{ é limitado}} \leq C_1 \cdot f(n) \cdot g(n)$$

ou seja, o produto  $h_1 \cdot h_2$  é limitado por uma constante multiplicada por  $f(n) \cdot g(n)$   
 ou seja:  
 $\mathcal{O}(f(n) \cdot g(n))$

c)  $h_L(n) = \mathcal{O}(f(n))$

$$h_L(n) \cdot c = K \cdot f(n), \forall n \geq n_0$$



$$h_L(n) \cdot c \leq K \cdot f(n) \cdot c, \forall n > n_0$$

$\rightarrow h_L(n) \cdot c \leq K \cdot c \cdot f(n)$

temos as  
definições de  
 $\text{big-O}$ .

análogicamente, multiplicam a função por uma constante não alterando "classe",  
apenas redimensiona seu valor.