

Resumo - Estudos PAA,

Prova 3 //

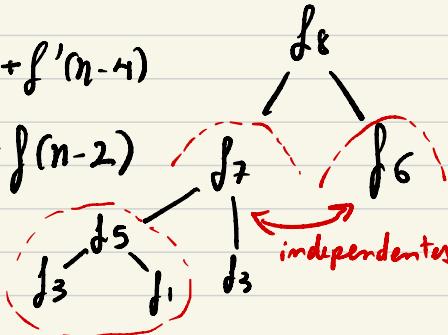
Sophia Carvazza

Divisão e Conquista:

- a instância do problema é dividida em duas ou + instâncias menores e independentes.
- cada instância menor é resolvida usando o próprio algoritmo definido
- as soluções das instâncias menores são combinadas p/ produzir uma solução da instância original.

$$f(n) = f'(n-2) + f'(n-4)$$

$$f(n) = f'(n-1) + f(n-2)$$



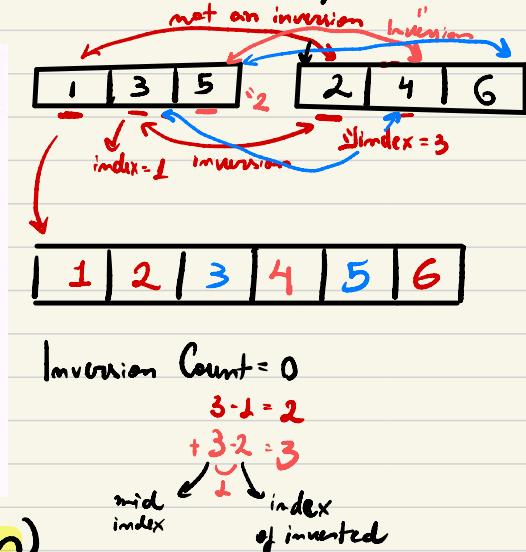
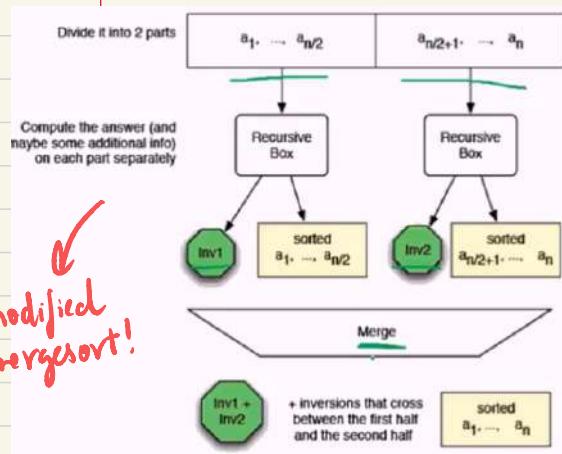
* exemplo → multiplicação de números grandes inteiros (Knuthuba), troumínó, closest pair of points (geometria computacional) e counting inversions

- ▶ Break up a problem into several parts.
- ▶ Solve each part **recursively**.
- ▶ Solve base cases by brute force.
- ▶ Efficiently **combine solutions** for sub-problems into final solution.
- ▶ Common use:
 - ▶ Partition problem into two equal sub-problems of size $n/2$.
 - ▶ Solve each part recursively.
 - ▶ Combine the two solutions in $O(n)$ time.
 - ▶ Resulting running time is $O(n \log n)$.

Counting Inversions:

→ indica o quanto pertence ao longe
o array está de estar ordenado

→ Dado um array, dois elementos $a[i]$ e $a[j]$ formam uma inversão se $a[i] > a[j]$ e $i < j$ (um número menor que outro está em uma posição maior). Mid-index = 3



★ Complexidade = $O(n \log n)$

Karatsuba

→ quanto tempo é necessário p/ multiplicar dois números naturais de m e n dígitos?

$$x = \underline{\underline{146}} \underline{\underline{123}}$$

$$x = 146123 = 146000 + 123 = \underbrace{146}_{a} \cdot \underbrace{10^3}_{b} + \underbrace{123}_{6/2}$$

$$x = a \cdot 10^{n/2} + b$$

$$\left. \begin{array}{l} x = \underline{\underline{146}} \underline{\underline{123}} \\ y = \underline{\underline{352}} \underline{\underline{120}} \end{array} \right\}$$

$$\rightarrow x \cdot y = (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d)$$

$$a \cdot c \cdot 10^{2(n/2)} + (ad + bc) \cdot 10^{n/2} + bd$$

$$\left. \begin{array}{l} ac = \text{Karatsuba}(a, c) \\ bd = \text{Karatsuba}(b, d) \\ ad + bc = \text{Karatsuba}(a+b, c+d) - ac - bd \end{array} \right\} \begin{array}{l} (a+b)(c+d) - ac - bd \\ (ac + ad + bc + bd) - ac - bd \\ = ad + bc \end{array}$$

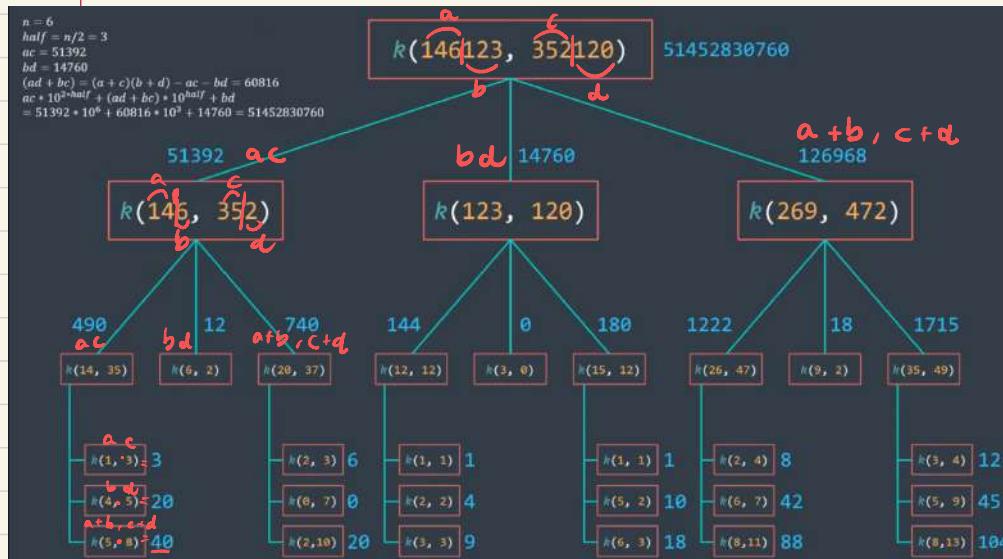
$k(146123, 352120)$

$$\begin{array}{c} K(146, 352) \\ \swarrow \quad \searrow \\ K(14, 35) \quad K(6, 2) \quad K(20, 37) \\ \swarrow \quad \searrow \quad \searrow \\ K(L, 3) \quad K(4, 5) \quad K(5, 8) \\ \text{base} = 20 \quad = 40 \quad = 90 \end{array}$$

$L \cdot 3 = 3$

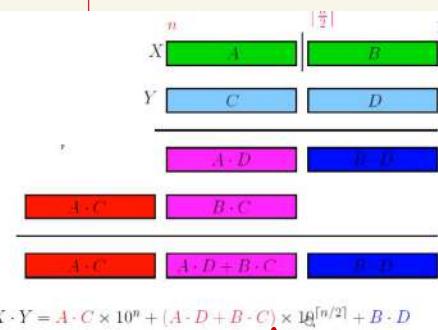
(um das daht
tun I no digits)

$$3 \cdot 10^2 + 17 \cdot 10^1 + 20 = 490$$



$$(ad + bc) = (a+c)(b+d) - ac - bd$$

$$ac \cdot 10^{2 \cdot n/2} + (ad + bc) \cdot 10^{n/2} + bd$$



$$\begin{array}{rcc} X & a & b \\ Y & c & d \end{array} \quad \begin{array}{c} ad \\ ac \\ a \cdot Y \end{array} \quad \begin{array}{c} bd \\ bc \\ ad + bc \\ X \cdot Y \end{array}$$

$$(a+b)(c+d) = ac + ad + bc + bd \Rightarrow ad + bc = (a+b)(c+d) - ac - bd$$

$$g = (a+b)(c+d) \quad e = ac \quad f = bd \quad h = g - e - f$$

$$(a+c)(b+d) - ac - bd$$

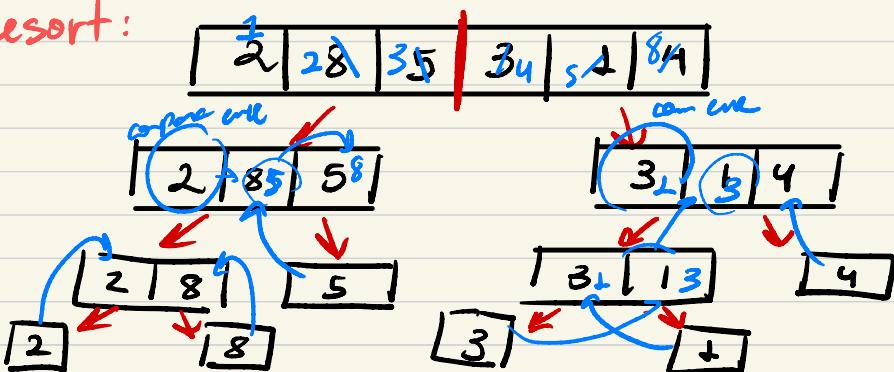
$$\frac{1}{x} \frac{x_1}{x_2} \cdot \frac{y_1}{y_2}$$

* Complexidade $\rightarrow O(n^{\log_2 3})$

$$T(n) = \begin{cases} 3T(n/2) + n & \Rightarrow O(n^{\log_2 3}) \\ 1 & \downarrow (3 \text{ chamados recursivos}) \end{cases}$$

$a = \text{mult. } (x_1, y_1)$
 $b = \text{mult. } (x_2, y_2)$
 $c = \text{mult. } (x_1 + x_2, y_1 + y_2)$
 return $a \cdot 2^n + (c - a - b) \cdot 2^{n/2} + b$

Mergesort:



```
mergesort (array a)
if (n == 1)
    return a

arrayOne = a[0] ... a[n/2]
arrayTwo = a[n/2+1] ... a[n]

arrayOne = mergesort (arrayOne)
arrayTwo = mergesort (arrayTwo)

return merge (arrayOne, arrayTwo)
```

```
merge (array a, array b)
array c

while (a and b have elements)
    if (a[0] > b[0])
        add b[0] to the end of c
        remove b[0] from b
    else
        add a[0] to the end of c
        remove a[0] from a

// At this point either a or b is empty

while (a has elements)
    add a[0] to the end of c
    remove a[0] from a

while (b has elements)
    add b[0] to the end of c
    remove b[0] from b

return c
```

> dividimos o array em 2 até permanecer com 1 elemento em cada array. (recursivamente)

> p/ cada recursão, ordenamos o subarray resultante separadamente (com 2 elertos, estaremos ordenado)

> comparar os elementos dos duos vetores ordenados um com um e mantemos um novo array (sempre adicionando o menor

work $\rightarrow O(n \log n) \rightarrow T(n) \left\{ 2T(n/2) + n \right\} \rightarrow$ intercalação (merge) dos metodos

$T(n/2) + T(n/2)$ - ordenação dos
2 metades (com recursividade)

elemento das duas arrays)

► repetimos esse processo de
divisão, ordenação e junção
até que tudo esteja ordenado.

Closest Pair of Points:

- encontrar os pontos + próximos (p/ prever colisão, por exemplo).
 - força bruta $\rightarrow O(n^2)$
 - técnica de Divisão e Conquista $\rightarrow O(n \log n)$
- caso base \rightarrow com 2 pontos, escolhemos os 2 como pares.

- 1- dividimos o espaço ao meio e consideramos os sub-espacos como novos arrays de pontos.
- 2- ordenamos esse array de pontos pelo seu valor de x (da escala X-Y).
- 3- repetimos esses passos recursivamente até encontrar os casos base. \rightarrow distância mínima
- 4- escolhemos o mínimo entre os pares em cada metade
- 5- guardamos a menor distância em cada lado e chamamos de δ (delta)
- 6- pegamos apenas os pontos que estão a uma distância máxima δ da linha vertical. Então, ordenamos esses pontos pelo eixo y.
- 7- p/ cada ponto, comparar com os + próximos na lista ordenada por y.
- 8- a menor distância estará dentro de uma das metades ou entre pontos de lados diferentes, próximos à linha divisória.

Example

```

closestPair(X, Y){
    n = X.length;
    // base cases:
    if (n==2): return dist(X[1], X[2]);
    if (n==3): return minAmongThree;

    // divide
    mid = X[n/2];
    dl = closestPair(X[1...mid], Y);
    dr = closestPair(X[mid+1...n], Y);
    d = min(dl, dr);

    // combine
    S = points in strip from Y
    for i=1 to S.length
        for j=1 to i+7
            d = min(d, dist(S[i], S[i+j]));
    return d;
}

```

cost barato dividir

distancia entre

distancia entre

anterior

* K-ésimo menor elemento de um array

- Quickselect (variação de quicksort)
- ordem de complexidade de $O(n)$ a $O(n^2)$

1º escolhemos um pivô (um elemento aleatório do array)

2º particionamos o array de forma que :

Pivô

1 → *elementos < que o pivô*

elementos > que o pivô

a pivô ocupa sua posição final \Rightarrow *(índice p)*

3º comparar com K :

- se $p+1 == K \rightarrow$ o pivô é o K -ésimo menor
- se $p+1 > K \rightarrow$ buscar recursivamente na parte esquerda
- se $p+1 < K \rightarrow$ buscar recursivamente na parte direita

$O(n \log n)$

$[0, 1, 2, 3]$

apontando K

segundo menor elemento $\rightarrow 2$

$v =$	0	1	2	3	4	5	6
	10	30	5	40	50	90	
	12	14	17	18	20	22	

$qS(v, 0, 5, 3)$

$qS(v, 0, 3, 3)$

→ quando chegarmos na posição $p=1$ achamos o segundo menor

$$p=1+1=2$$

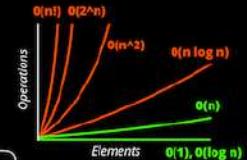
LEGEND

TIME Complexity VS. SPACE Complexity

- | TIME Complexity | SPACE Complexity |
|-----------------|------------------|
| Good | Good |
| Fair | Fair |
| Bad | Bad |

<BIG-O-CHEATSHEET>

www.bigochartsheet.com



DATA STRUCTURE

Operations

DATA Structure



DATA Structure	TIME Complexity				SPACE Complexity			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)
Circular Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)

ARRAY SORTING

Algorithms

ARRAY Algorithms



ARRAY Algorithms	TIME Complexity			SPACE Complexity		
	Best	Average	Worst	Best	Average	Worst
Quicksort				O(n log(n))	O(n log(n))	O(n^2)
Mergesort				O(n log(n))	O(n log(n))	O(n log(n))
TimSort				O(n)	O(n log(n))	O(n log(n))
HeapSort				O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort				O(n)	O(n^2)	O(n^2)
Insertion Sort				O(n)	O(n^2)	O(n^2)
Selection Sort				O(n^2)	O(n^2)	O(n^2)
Tree Sort				O(n log(n))	O(n log(n))	O(n^2)
Shell Sort				O(n log(n))	O((n log(n))^2)	O((n log(n))^2)
Bucket Sort				O(n+k)	O(n+k)	O(n^2)
Radix Sort				O(n/k)	O(n/k)	O(n/k)
Counting Sort				O(n+k)	O(n+k)	O(n+k)
Cursorsort				O(n)	O(n log(n))	O(n log(n))

Guloso vs Programação Dinâmica vs Divisão e Conquista

- Escolha localmente ótima incremental
- Pode abordagem míope (otimizações locais)
- Top-Down incremental
- Nunca reconsidera decisões.

* rápidos, mas podem não ser ótimos

- Baseada na sobreposição de subproblemas
- Pode subestrutura ótima *
- Bottom-Up ou Top-Down
- Reutiliza soluções já calculadas (sobreposição).

* garante optimidade quando correta, mas + complexidade e uso de memória

qualquer problema resolvido por Guloso pode ser resolvido com P.D.

- Divisão do problema em subproblemas independentes e combina as soluções
- Resolução recursiva
- Top-Down recursivo
- Não há sobreposição de problemas.

* paralelização natural dos subproblemas

* Infos da Programação Dinâmica:

* Um problema tem subestrutura ótima quando uma solução ótima p/ o problema contém em seu interior, soluções ótimas p/ os subproblemas

* A sobreposição de subproblemas ocorre quando um algoritmo recursivo reexamina o mesmo problema muitas vezes

→duce até os casos base

* Há duas formas de abordagem:

> Top-Down: (Revolução + memoização) libera o problema de forma recursiva e armazena os valores em uma tabela.

> Bottom-Up: (Iteração + tabulação) Começamos pelos problemas menores e aumentando a complexidade com o passar da execução.

→ (casos base)

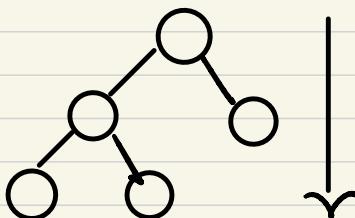
→ (loops)

(Top Down)
↑

memoização vs tabulação

- Armazena os resultados na tabela conforme necessário;
- Verifica primeiro se o resultado já foi computado antes de computar (preenche a tabela só demanda, só quando precisa)

top-down



(Bottom Up)
↑

tabulação

- Preenche uma tabela sistematicamente;
- Calcula todos os subproblemas de forma itativa, de menor pra maior (preenchendo toda a tabela, sem verificação prévia)

bottom-up



```
1 def fib memoized(n):
2     def fib(n):
3         if n <= 1:
4             return n
5         else:
6             return fib(n-1) + fib(n-2)
7
8     cache = {}
9     if n not in cache:
10        cache[n] = fib(n)
11
12    return cache[n]
```

```
1 def fib(n):
2     t = []
3     t.append(0)
4     t.append(1)
5
6     for i in range(2, n):
7         t.append(t[i-1] + t[i-2])
8
9     return t[n-1]
```

Recorrência

→ representação matemática da recursividade

recursividade
função que chama
ela mesma

direta

indireta

uma função chama
a si mesma diretamente
dentro de sua própria
definição

→ ex: Fibonacci ou
Fatorial

quando uma função
chama outra função,
que por sua vez chama
a função original no-
varmente.

→ ex: $f(n) = f'(n-2) + f'(n-1)$
 $f(n) = f'(n-2) + f(n-2)$

Recorrências famosas e suas complexidades:

busca binária → $T(n) = \begin{cases} T(n/2) + 1 & \rightarrow O(\log_2 n) \\ 1 \end{cases}$

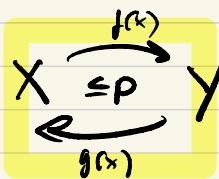
$$T(n) = \begin{cases} T(n/3) + 2 & \rightarrow O(\log_3 n) \\ 1 \end{cases}$$

Merge sort → $T(n) = \begin{cases} 2T(n/2) + n & \rightarrow O(n \log n) \\ 1 \end{cases}$

Karatsuba → $T(n) = \begin{cases} 3T(n/2) + n & \rightarrow O(n^{\log_2 3}) \\ 1 \end{cases} \quad O(n^{\log_3 2 - \epsilon})$

- Casos Gerais:
- $T(n) = aT(n/b) + f(n) \rightarrow O(n^{\log_b a})$
 - (Teorema Master)
 - $T(n) = 2T(n/2) + n \rightarrow O(n \log n)$
 - $T(n) = T(n-1) + C \rightarrow O(n)$

Reduções polinomiais:



- $X \leq_p Y$
- Y é tão difícil quanto X
- X é tão fácil quanto Y

mas pode ser muito mais difícil que X (então se X for polinomial, por ex., não implica que Y também seja).

Reduções Conhecidas:

3-SAT \leq_p Clique

3-SAT \leq_p Vertex Cover

3-SAT \leq_p Caminho Hamiltoniano

3-SAT \leq_p Coloração de Grafos

3-SAT \leq_p Subset Sum

Caminho Hamil. \leq_p Traveling Salesman (TSP)

Vertex Cover \leq_p Independent Set

Independent Set \leq_p Clique

} redução direta

} redução por complemento

★ Todo problema P se reduz p/ qualquer outro problema (inclusive NP-Completo), mas o contrário é desconhecido.

★ Todo problema NP-Completo se reduz p/ outro NP-Completo

Como Funciona a Redução Trivial

Passo a passo:

1. Recebemos uma instância x do problema A
2. Resolvemos x diretamente usando o algoritmo polinomial de A (isso é possível porque A $\in P$)
3. Baseado na resposta:
 - Se a resposta para x é "SIM" \rightarrow criamos uma instância de B que sabemos que tem resposta "SIM"
 - Se a resposta para x é "NÃO" \rightarrow criamos uma instância de B que sabemos que tem resposta "NÃO"

Exemplo Prático

Suponha que queremos reduzir Ordenação (que está em P) para Satisfatibilidade (SAT):

text

Função de Redução(lista L):

1. Ordena L usando algoritmo $O(n \log n)$ // Resolve diretamente
2. Se L foi ordenada com sucesso:
 - Retorna fórmula SAT sempre verdadeira: $(x \vee \neg x)$
3. Senão:
 - Retorna fórmula SAT sempre falsa: $(x \wedge \neg x)$

Revisão de Complexidade

→ Big O → limite superior

$f(n) = O(g(n))$ SSE: $\exists c > 0, n_0 \geq 0 \mid f(n) \leq c \cdot g(n), \forall n > n_0$

→ $\hat{\Omega}$ mega (Ω) → limite inferior

$$f(n) = \Omega(g(n)) \text{ SSE: } \exists c > 0, n_0 > 0 \mid f(n) \geq c \cdot g(n), \forall n > n_0$$

→ Theta (Θ) → comportamento médio exato

$$f(n) = \Theta(g(n)) \text{ SSE: } \exists c_1 > 0, c_2 > 0, n_0 > 0 \mid c_1 \cdot g(n) \geq f(n) \geq c_2 \cdot g(n), \forall n > n_0$$

$$\text{Se } \left. \begin{array}{l} f(n) = n \\ g(n) = n^2 \end{array} \right\} \quad \begin{array}{l} f(n) = O(g(n)) \\ f(n) = \Omega(g(n))? \end{array} \quad X$$

$$\star \text{Se } \left. \begin{array}{l} f(n) = n^2 \\ g(n) = n^2 \end{array} \right\} \quad \left. \begin{array}{l} f(n) = O(g(n)) \\ f(n) = \Omega(g(n))! \end{array} \right. \quad \checkmark$$

$f(n) = g(n)$

$$\star f(n) = O(g(n)) \quad \text{SSE: } g(n) = \lceil 2(f(n)) \rceil$$

Se $f(n) = \Omega(g(n))$ → então $g(n) = O(f(n))$

* Se $\begin{cases} f(n) = O(g(n)) \\ g(n) = O(h(n)) \end{cases}$ então $f(n) = O(h(n))$ (a notação)

$$f(n) \leq g(n) \leq c \cdot h(n)$$

*** Se** $\left. \begin{array}{l} f(n) = \Theta(g(n)) \\ g(n) = \Theta(h(n)) \end{array} \right\}$ **então** $f(n) = \Theta(h(n))$

* Se $\left. \begin{array}{l} f(n) = \Omega(g(n)) \\ g(n) = \Sigma(h(n)) \end{array} \right\}$ então $f(n) = \Omega(h(n))$

*todos
só
tradicionais*

Se encontrarmos UM PAIR que satisfizer a equação, ela é verdadeira

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

- (V) $2^{n-1} = O(2^n)$ ✓ $f(n) = 2^{n-1}$ $g(n) = 2^n$ $f(n) \leq c \cdot g(n)$
- (V) $2^{n+1} = O(2^n)$ ✓
- (F) $2^{3n} = O(4^n)$ - F
- (V) $2^{3n} = O(8^n)$ ✓
- (V) $10000n^2 + 10000n + n\log n = O(n^3)$ ✓
- (V) $\log_3 n = O(\log n)$ ✓

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

- (a) V - F - F - V - V - V ✗
 ✗ 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 ✗

* P/ provar que é Falsa, quando não existe nenhum par que satisfizer a equação, usamos argumentos de limite:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

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

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

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

$$\text{condições } c=1 \text{ e } n_0=1$$

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

$$2^{\cancel{n}} \cdot \frac{1}{2} \leq 1 \cdot 2^{\cancel{n}}$$

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

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

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

$$2^n \cdot 2^1 \leq c \cdot 2^n$$

$$2^1 \cdot 2 \leq 2 \cdot 2^1$$

$$\frac{2^n \cdot 2}{2^n} \leq c \Rightarrow c \geq 2$$

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

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

$$2^{3n} \leq c \cdot 2 \cdot 2^n$$

$$\frac{2^{3n}}{2^n} \leq c \cdot 2$$

$$2^{2n} \leq c \cdot 2$$

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

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

$$2^{3n} \leq c \cdot 2^{3n}$$

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

$$1 \leq c$$

$$8 \leq 2 \cdot 8$$

$$8 \leq 16$$

Não conseguimos definir um valor fixo de c que atenda a equação.

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

$$1 \leq c$$

✓

$$e) 1000n^2 + 1000n + n \log n \leq c \cdot n^3$$

$$p/c = 10.000 \text{ e } n=10$$

$$\frac{1000n^2 + 1000n + n \log n}{n^3} \leq c$$

$$\frac{1000 \cdot 10 + 1000 + \log 10}{10^2} \leq c$$

$$\frac{100 + 10 + \frac{10 \cdot 10}{100}}{100} \leq 10.000$$

$$100 + 10 + 0,01 \leq 10.000$$

$$100,01 \leq 10.000$$

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

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

$$\frac{1000 \cdot \infty + 1000 + \log \infty}{\infty^2} \leq c$$

tende a 0.

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

$$\log_3 n \leq c \cdot \log_{10} n$$

$$\frac{\log_{10} n}{\log_{10} 3} \leq c \cdot \log_{10} n$$

$$\frac{\frac{\log_{10} n}{\log_{10} 3}}{\frac{\log_{10} n}{\log_{10} 3}} \leq c$$

$$\frac{1}{\log 3} \leq c \checkmark$$

$$p/c = \frac{1}{\log 3} \quad n=L \rightarrow \frac{\log L}{\log 3} \leq \frac{1}{\log 3} \cdot \log L \checkmark$$

$$\left(\frac{\log L}{\log 3} = \frac{\log L}{\log 3} \right)$$

i) Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$
 então $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

ii) Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$
 então $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$

i) $\begin{cases} f_1(n) = O(g_1(n)) \\ f_2(n) = O(g_2(n)) \end{cases} \quad \left\{ \begin{array}{l} f_1(n) + f_2(n) = O(g_1(n) + g_2(n)) \end{array} \right.$

$f_1(n) = O(g_1(n))$ sse: $\exists c_1 > 0, n_0 \geq 0 \mid f_1(n) \leq c_1 \cdot g_1(n), \forall n > n_0$

$f_2(n) = O(g_2(n))$ sse: $\exists c_2 > 0, n_0 \geq 0 \mid f_2(n) \leq c_2 \cdot g_2(n), \forall n > n_0$

$$f_1(n) + f_2(n) \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n)$$

$\begin{cases} \text{Se } c_1 > c_2 \rightarrow \leq c_1 \cdot g_1 + c_1 \cdot g_2 \\ \text{Se } c_2 > c_1 \rightarrow \leq c_2 \cdot g_1 + c_2 \cdot g_2 \end{cases} \quad \left\{ \begin{array}{l} C = \max(c_1, c_2) \rightarrow \leq C \cdot g_1 + C \cdot g_2 \end{array} \right.$

pois o C máximo
 vai definir o comportamento
 asymptótico predominantemente
 dos dois f_1 e f_2

$$f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$$

se maximizarmos C ,
 temos que:



$$\text{ii) } \left. \begin{array}{l} f_1(n) = O(g_1(n)) \\ f_2(n) = O(g_2(n)) \end{array} \right\} \quad \checkmark$$

Como $\max(g_1(n), g_2(n)) \geq g_1(n)$ e $\max(g_1(n), g_2(n)) \geq g_2(n)$ o máximo entre os dois sempre vai definir o comportamento.

$$f_1(n) + f_2(n) \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n)$$

$$c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \leq c_1 \cdot \max(g_1(n), g_2(n)) + c_2 \cdot \max(g_1(n), g_2(n))$$

$$c_1 \cdot \max(g_1(n), g_2(n)) + c_2 \cdot \max(g_1(n), g_2(n)) = (c_1 + c_2) \cdot \max(g_1(n), g_2(n))$$

$$\text{portanto } \Rightarrow f_1(n) + f_2(n) \leq (c_1 + c_2) \cdot \max(g_1(n), g_2(n))$$

2- Se $f(m) = O(g(m))$ e $h(m) = O(g(m))$ então $f(m) = h(m)$?

pt limite superior

↳ Falso \times

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

$$f(n) \leq c_1 \cdot g(n)$$

$$h(n) \leq c_2 \cdot g(n)$$

não há garantia de que c_1 e c_2 sejam iguais ou que $f(n) = h(n)$
sejam iguais \rightarrow não é que os 2 tenham como limite assimétrico a $g(n)$

Prove que:

- a) (12 %) Seja $h_1(n) = O(f(n))$ e $h_2(n) = O(g(n))$. Mostre que $h_1(n) + h_2(n) = O(\max\{f(n), g(n)\})$
- b) (13 %) Seja $h_1(n) = O(f(n))$ e $h_2(n) = O(g(n))$. Mostre que $h_1(n) \cdot h_2(n) = O(f(n) \cdot g(n))$

$$\text{a)} \quad h_1(n) = O(f(n)) \quad h_1(n) + h_2(n) = O(\max\{f(n), g(n)\})$$

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

$$h_1(n) + h_2(n) \leq c_1 \cdot f(n) + c_2 \cdot g(n)$$

$$c_1 \cdot f(n) + c_2 \cdot g(n) \leq c_1 \cdot \max(f(n), g(n)) + c_2 \cdot \max(f(n), g(n))$$

$$c_1 \cdot f(n) + c_2 \cdot g(n) \leq (c_1 + c_2) \cdot \max(f(n), g(n))$$

$$h_1(n) + h_2(n) \leq (c_1 + c_2) \cdot \max(f(n), g(n))$$

$$O(\max(f(n), g(n)))$$

$$\text{b)} \quad h_1(n) = O(f(n))$$

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

$$h_1(n) \cdot h_2(n) = O(f(n) \cdot g(n))$$

de acordo com a
definição

$$h_1(n) \cdot h_2(n) \leq c_1 \cdot f(n) \cdot c_2 \cdot g(n)$$

$$h_1(n) \cdot h_2(n) \leq (c_1 \cdot c_2) (f(n) \cdot g(n))$$

$$\text{e nesse caso } n_0 = \max(n_1, n_2)$$

$$O(f(n) \cdot g(n))$$

Dada uma coleção **S de intervalos**, em que cada intervalo é representado pela tripla (s, f, v) sendo s o tempo de início, f o tempo de fim do intervalo e v é o ganho associado a cada intervalo. Se temos vários intervalos, numerados de 1 a n , o início de um intervalo i será denotado por s_i e o término por f_i . Um intervalo i é anterior a um intervalo j se $f_i < s_j$. Analogamente, i é posterior a j se $s_i > f_j$. Dois intervalos i e j são disjuntos se e somente se i é posterior a j ou anterior a j . Uma coleção de intervalos é disjunta se os intervalos da coleção são disjuntos dois a dois. O problema do **escalonamento ponderado de intervalos** consiste em encontrar uma **subcoleção disjunta** de S com ganho máximo M , isto é, não existe outra subcoleção com ganho maior que M . Projete (i) uma solução gulosa que apresente o melhor resultado possível quando todos os ganhos forem maiores ou iguais a 1 (10 %); (ii) uma solução usando programação dinâmica que apresente o melhor resultado possível quando todos os ganhos forem maiores ou iguais a 1 (15 %). Qual(is) solução(es) será(ão) ótima(s)? Por quê? Dê um exemplo de sua solução. Justifique todas as suas opções e escolhas, assim como, deixe claro tanto a modelagem quanto os custos computacionais.

$S \rightarrow$ tempo de inicio

$f \rightarrow$ tempo de fim

$v \rightarrow$ ganho de cada intervalo

PD:

i) 1- ordenarmos os intervalos por tempo de término em ordem não - decrescente.

2- p/ cada intervalo i (s, f, u) , calcularmos o maior índice que caeca antes de i e $f_j \leq s_i$ (\circ intervalo termina antes ou quando i começa).

3- recorrência $\rightarrow M(i) = \max(M_i + M[f_p(i)], M[i-1])$

\hookrightarrow caso base $= M[0] = 0$

p/ $i=1$ atén

solução ótima dos intervalos que terminam antes de começar.

4- o resultado será a última posição da tabela.

solução ótima ignorando e.

Gulosa:

i) 1- ordenarmos os intervalos por tempo de término em ordem não - decrescente.

2- p/ cada intervalo i (s, f, u) , calcularmos o maior índice que caeca antes de i e $f_j \leq s_i$ (\circ intervalo termina antes ou quando i começa).

3- Ordenarmos os intervalos por peso não - crescente e subtraímos a maior

X

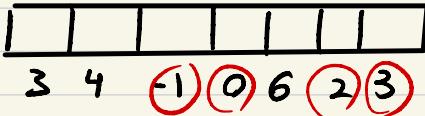
QUESTÃO 3

(25 %)

Considere que todos os alunos de PAA são colocados de forma aleatória em uma única fila. Projete um algoritmo que retire o menor número possível de alunos da fila, e que os restantes fiquem em ordem crescente. Justifique todas as suas opções e escolhas, assim como, deixe claro tanto a modelagem quanto os custos computacionais.

Problema de
Largest Increasing Subsequence

→ (Programação Dinâmica)



- mantemos uma tabela com $x = \text{velhos}$ ou ordem da fila
- $y = \text{valores ordenados crescentemente}$
- preenchemos cada posição com: $\max(dp[i][j-1], dp[i-1][j])$
- se o os valores não batem, a $dp[i-1][j-1] + 1$ se os valores batem.
- caso base $dp[0] = 0$

ou seja:

	0	3	4	-1	0	6	2	3
0	0	0	0	0	0	0	0	0
-1	0	0	1	1	1	1	1	1
0	0	0	0	2	2	2	2	2
2	0	0	0	2	2	3	3	3
3	0	1	1	1	2	2	3	4
3	0	1	1	1	2	2	3	4
4	0	1	2	2	2	3	4	
6	0	2	2	2	3	3	4	

(Também é possível
fazer com só 1
array!)

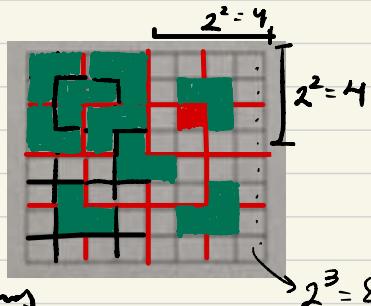
resultado

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.



Divisão e Conquista

* O tabuleiro deve ser $2^n \times 2^n$!



Se o tabuleiro for 2×2 , preenche com 1 tromino, deixando o quadrado vazio.

divide o tabuleiro em 4 sub-tabuleiros de $2^{n-1} \times 2^{n-1}$

colocar o tromino no meio de tabuleiro de forma que ocupe 2 posições de cada sub-tabuleiro (exceto o que tem o quadrado vazio) → o que formará um novo quadrado vazio em cada um dos 3 sub-tabuleiros

replicarmos esses passos de divisão e preenchimento até preencher o tabuleiro

ele irá na verdade preenchido, só com que ele comece num quadrado certo

* Complexidade:

$$T(n) = \{ 4T\left(\frac{n}{2}\right) + 1 \}$$

$$= O(n^2) \leftarrow$$

* logo = $O(n^2)$ ou $O(4^K)$

considerando que $n = 2^K \Rightarrow n^2 = 4^K = O(4^K)$

- 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 .

$$\text{a)} \quad h_1(n) = O(f(n)) \quad h_1(n) + h_2(n) = O(\max\{f(n), g(n)\})$$

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

$$h_1(n) + h_2(n) \leq c_1 \cdot f(n) + c_2 \cdot g(n)$$

$$= (c_1 + c_2) \cdot (f(n) \cdot g(n))$$

$$\text{e, se } c_1, c_2 \in \mathbb{R} \text{ s.t. } f(n) + g(n) \leq (c_1 + c_2) \cdot \max\{f(n), g(n)\}$$

$$h_1(n) \cdot h_2(n) \leq (c_1 + c_2) \cdot \max\{f(n), g(n)\}$$

\hookrightarrow (quando $n_0 = \max\{n_1, n_2\}$)

$$\text{b)} \quad h(n) = O(h(n)) ? \rightarrow h(n) \leq \overbrace{c \cdot h(n)}$$

toda função é assintoticamente limitada por si mesma

$$\frac{h(n)}{h(n)} \leq c \quad \begin{array}{l} \text{se } c = 1 \text{ e } n = 1, \\ \text{por exemplo, é verdadeira} \\ 1 \leq c \end{array}$$

Questões sobre tratabilidade:

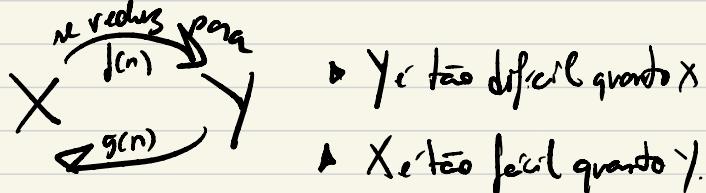
- a) (10 %) Como provar que um problema X pertence a classe de problemas NP-Completo?
- b) (10 %) Discorra sobre reduções polinomiais entre dois problemas mostrando o grau de dificuldade relativo estes problemas.

a) I-X deve pertencer a NP (um problema é polinomial nos de verificação que podemos verificar uma instância em tempo polinomial)

2-Todo outro problema B contido em NP deve ser reduzir polinomialmente a X.

b) Um problema X é reduzível em tempo polinomial a Y quando qualquer instância de X pode ser transformada em uma instância de problema Y .

Agora, X é tão fácil quanto Y e Y é tão difícil quanto X (pois já tínhamos conhecimento da dificuldade de X , e demonstrando que ele é reduzível p/ Y , agora sabemos que Y é tão difícil quanto ele).

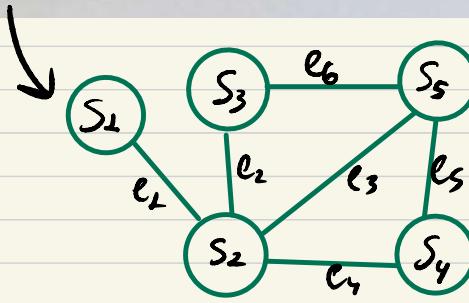


QUESTÃO 3 (25 %)

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.



- formemos cada conjunto dos vértices como e_i ; e vértice como S_i
- formemos cada conjunto de vértices que saem de um vértice como um subconjunto da coleção S .

$$S_1 = \{e_1\}$$

$$S_2 = \{e_1, e_2, e_3, e_4\} \leftarrow$$

$$S_3 = \{e_2, e_6\}$$

$$S_4 = \{e_3, e_4, e_5\}$$

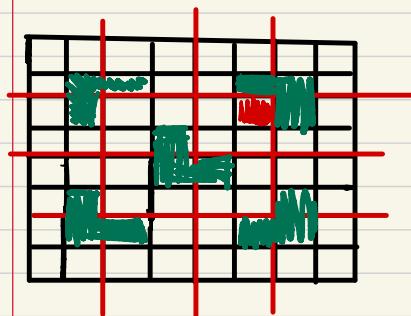
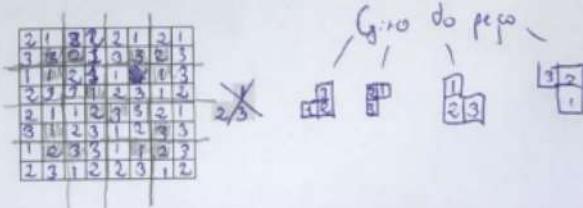
$$S_5 = \{e_3, e_5, e_6\} \leftarrow$$

para $K = 3$:
 $S_2 \cup S_5 = U$

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^m 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.

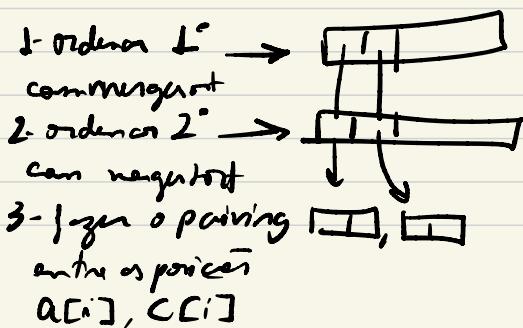


QUESTÃO 3

(30 %)

Seja uma competição de ciclismo de velocidade por equipes, em que todos os membros estão divididos em m equipes. Um dos desafios das equipes é fazer um percurso o mais rapidamente possível. Você 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 de cada ciclista sabe-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 do par ciclista/bicicleta. Considere que a sua equipe possui n membros, com pesos a_1, a_2, \dots, a_n , e que existam n bicicletas com pesos c_1, c_2, \dots, c_n .

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 deixe claro como chegou ao algoritmo com o custo desejado.



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

- X Algoritmos gulosos têm como principal característica a construção incremental da solução, por meio da otimização局部 de algum critério local, e por causa disto não é possível obter soluções ótimas. → pode ser sim
 - X Algoritmos gulosos têm como principal característica a construção incremental da solução, por meio da otimização局部 de algum critério global, sempre obtendo soluções ótimas. → sempre ótimas
 - ✓ Algoritmos com ordem de complexidade $O(n)$ também possuem ordem de complexidade $O(n^2)$. → $c \cdot n \leq c \cdot n^2$
 - X 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. → depende da estrutura, etc
 - ✓ Encontrar elementos em um vetor, de forma exata, pode ser feita em $O(n)$. → busca linear normal
 - ✓ É possível fazer uma busca binária, desenvolvida por meio de recursão, uma busca binária, desenvolvida por meio de uma estrutura de repetição, terem a mesma ordem de complexidade. → os dois têm a mesma ordem
- 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 ✗
- $T(n) = T(n/2) + 1$
 cada iteração divide a metade dos elementos.
 $O(n \log n)$

Problema dos Divisíveis:

$$\left\{ \begin{array}{l} T[i, j] = \max_{i \leq k \leq j} \frac{T[i, k]}{T[k+1, j]} \rightarrow \text{maximizar} \\ +[i, j] = \min_{i \leq k \leq j} \frac{+[i, k]}{T[k+1, j]} \rightarrow \text{minimizar} \end{array} \right.$$

QUESTÃO 1

A distância de edição (ED) é uma métrica para strings, ou seja, uma forma de quantificar o quanto diferentes dois strings são entre si. Esta medida seria o menor número de operações necessárias para transformar um string em outro. Todas as operações (inserção, remoção e substituição de caractere) têm o mesmo peso, que é igual a 1. Assim, deseja-se encontrar o número de operações, denotado por $ED(m, n)$, par transformar X_m em Y_n (ou vice-versa), onde uma definição recursiva incompleta para a função $ED(i, j)$ para calcular ED entre X_m e Y_n é fornecida abaixo:

$$ED(i, j) = \begin{cases} 0 & \text{if either } i = 0 \text{ or } j = 0 \\ expr1 & \text{if } i, j \geq 0 \text{ and } X_{i-1} = Y_{j-1} \\ expr2 & \text{if } i, j \geq 0 \text{ and } X_{i-1} \neq Y_{j-1} \end{cases}$$

[A] $expr1 = ED(i, j - 1) + 1$

[B] $expr1 = ED(i - 1, j - 1) + 1$

[C] $expr2 = \min(ED(i - 1, j) + 1, ED(i, j - 1) + 1, ED(i - 1, j - 1) + 1)$ ✓

[D] $expr2 = \max(ED(i, j - 1), ED(i - 1, j))$

$X = a b c d$

$Y = a b e d$

mantém

remoção (ED)(i-1, j)

inserção (ED)(i, j-1)

substituição (ED)(i-1, j-1)

$ED(i, j) = \text{distância entre os 2 caracteres}$

QUESTÃO 2

Qual das alternativas a seguir é uma desvantagem de um algoritmo guloso?

Nem sempre pode encontrar a solução globalmente ótima.

Pode ser lento para tamanhos de entrada grandes.

Garante sempre a solução ótima. → depende do algoritmo

Requer muita memória. → depende

QUESTÃO 3

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

Algoritmos gulosos têm como principal característica a construção incremental da solução, por meio da otimização mfope de algum critério local, e por causa disto, não é possível obter soluções ótimas. → paralelo sim

Algoritmos gulosos têm como principal característica a construção incremental da solução, por meio da otimização mfope de algum critério global, sempre obtendo soluções ótimas. → sempre também não

Algoritmos de programação dinâmica são baseados na sobreposição de sub-problemas. A solução completa é baseada na análise dos problemas menores para os problemas maiores. → Bottom-up

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

[A] V - V - F

[B] F - F - F

[C] V - V - V

F - F - V

QUESTÃO 4

Suponha que você tenha moedas de valores 1, 3 e 4. Você usa um algoritmo guloso, no qual escolhe a moeda de maior valor que não é maior que a soma restante. Para qual das seguintes somas o algoritmo produzirá uma resposta ótima?

100

10

14

6

$100/4 = 25$

$10/4 = 2$

$14/4 = 3$

$6/4 = 1$

$100/4 = 25$ moedas de 4 → não resta mais → 6 → 4, 1, 1 (ótimo nlc 3, 3)

$10/4 = 2 \cdot 4 =$ sobra 2 + 2 moedas de 1 → 4, 4, 1, 1 (mas o ótimo nlc 4, 3, 1)

$14/4 = 3 \cdot 4 =$ sobra 2 → 1, 1, 4, 4, 4 (mas o ótimo nlc 4, 4, 3, 3)

$$\begin{array}{l} A = \underline{1}, \underline{3}, \underline{2}, \underline{5} \\ B = \underline{2}, \underline{5}, \underline{2}, \underline{4}, \underline{3} \end{array} \left. \right\} \text{major subgroups column} = 1, 3 \text{ on } 1, 5$$

$$\begin{array}{c} \text{LCS}(A, \text{inv}(A)) : \text{LCS}(A, \text{ord}(A)) \\ \begin{array}{cccc} 1 & 3 & 2 & 5 \\ 1 & 3 & 2 & 5 \\ | & & & \\ 2 & & & \\ 3 & & & \\ \hline 11 & & & \checkmark \end{array} \end{array}$$

major req.
increcente

QUESTÃO 5

Sejam duas sequências de inteiros $A = a_1, a_2, \dots, a_n$ e $B = b_1, b_2, \dots, b_m$. Encontrar o tamanho da maior subsequência comum entre A e B , denotado por $LCS(A, B)$ com custo computacional, em termos de tempo, $O(mn)$ significa encontrar subsequências $a_{i_1} a_{i_2} \dots a_{i_k} = b_{j_1} b_{j_2} \dots b_{j_k}$, em que $i_1 < i_2 < \dots < i_k$ e $j_1 < j_2 < \dots < j_k$, sendo k maior possível. Analise as assertivas a seguir, assimilando V se a assertiva for verdadeira, ou F se a assertiva for falsa.

- Se $A = 1, 3, 2, 5$ e $B = 1, 5, 2, 4, 3$, o $LCS(A, B) = 3$. *problema em ordem crescente e depois dizer não-decrescente.*

Para encontrar a maior subseqüência crescente de A , basta aplicar $LCS(A, \text{ord}(A))$ em que $\text{ord}(A)$ é uma seqüência ordenada não-decrescente. *1001. v*

Sejam duas seqüências A e B , de tamanhos m e n , respectivamente, uma seqüência C é uma superseqüência comum de A e B , se A e B são subseqüências (não necessariamente consecutivas) de C . O tamanho da menor superseqüência comum entre A e B é dado por $MCS(A, B) = n + m - LCS(A, B)$. *→ MCS = Shortest Common Supersequence*

Sejam duas seqüências A e B , de tamanhos m e n , respectivamente, uma seqüência C é uma superseqüência comum de A e B , se A e B são subseqüências de C (não necessariamente consecutivas) de C . O tamanho da menor superseqüência comum entre A e B é dado por $MCS(A, B) = n + m - 2 \times LCS(A, B)$.

Para encontrar a maior subseqüência que seja um palíndromo de A , basta aplicar $LCS(A, \text{inv}(A))$ em que $\text{inv}(A)$ é uma seqüência invertida de A . *1002. v* *é a maior subseqüência de A que é palíndromo.*

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

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

- Com intervalos de tamanhos iguais, o problema de encontrar o maior conjunto de intervalo disjuntos pode ser resolvido de forma ótima usando a estratégia de menor tempo de início. Interval scheduling

Para um grafo simples, conexo e não direcionado com n vértices e exatamente n arestas, é possível afirmar que é possível encontrar a minimum spanning tree em tempo $O(n)$ *.

Coloração de vértices não pode ser resolvida em tempo polinomial mesmo se $P = NP$. ~~NP completo, entao se P=NP ele poderia ser resolvida em tempo polinomial~~

Em problemas de programação dinâmica, os problemas são divididos em subproblemas dependentes, que são resolvidos em qualquer ordem para encontrar a solução final. ↓ ↓ ↓ ↓ ↓

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

- A V-V-F-F B F-F-F-V C V-F-V-F D F-V-F-F

um ciclo pode ser encontrado em $O(n)$ usando DFS ou BFS

Se precisa de pelo menos $n-1$ bordas (arrows)
Se tem n bordas, então tem uma borda "extra"
garantindo a existência de círculos.

* MCS → Shortest Common Supersequence

↳ Dado duas sequências A e B , uma supersequência comum é uma sequência em que podemos obter tanto A quanto B removendo alguns elementos

↳ ex: $A = abc$ } supersequência comum = abcc
 $B = aec$ } (4 elementos)

* Algoritmo:

- calcular a LCS entre duas sequências
- inserir os elementos mais-LCS na sequência LCS, preservando a ordem original

$$\text{ou seja: } \text{tam}(A) + \text{tam}(B) - \text{LCS}(A, B)$$

$$3 + 3 - 2 = 4 \checkmark$$

X é fácil quanto Y, e Y é polinomial
(então mesmo se for muito + fácil, continua sendo polinomial)

pelo menos
X → Y polinomial

QUESTÃO 7

Sejam dois problemas X e Y . Analise as assertivas a seguir, assinalando V se a assertiva for verdadeira e F se a assertiva for falsa.

- X Se o problema X for resolvido em tempo polinomial, e há uma redução em tempo polinomial de X para Y , então o problema Y também poderá ser resolvido em tempo polinomial. *X se reduz a Y com tempo polinomial, mas não podemos afirmar nada sobre Y.*
- F Se o problema Y puder ser resolvido em tempo polinomial, e há uma redução em tempo polinomial de X para Y , então o problema X também poderá ser resolvido em tempo polinomial. *Deve-se ter certeza de que X, mas pode haver um erro. X nem podem dizer que X é polinomial de fato.*
- V Considere que há uma redução em tempo polinomial de X para Y , então pode-se afirmar que o problema X é, no mínimo, tão fácil quanto o problema Y . *Logo, X é pelo menos tão fácil quanto Y.*
- F É possível afirmar que não há uma redução em tempo polinomial de um problema NP-Completo para um problema da classe P. *Isso é uma dúvida na computação, então não podemos afirmar.*
- F Um problema X é dito ser NP-Completo se X pertencer a classe NP e se houver uma redução de X para um problema NP-Completo Y . *Contrário de Y para X.*

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

- A F - F - V - F - B B F - F - V - V - F C V - V - V - F - V D V - V - F - V - V

QUESTÃO 8

Sejam duas funções positivas $f(n)$ e $g(n)$. O limite assintótico inferior de uma função $f(n)$ é dito ser $\Omega(g(n))$, representado por $f(n) = \Omega(g(n))$, se existirem duas constantes $c > 0$ e $n_0 \geq 0$ tal que $f(n) \geq cg(n) \forall n \geq n_0$. Ainda, $f(n)$ é dito ser $\Theta(g(n))$ se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$. Analise cada uma das assertativas a seguir.

1. Se $f(n) = O(g(n))$ e $h(n) = O(g(n))$ então podemos afirmar que $f(n)$ é igual a $h(n)$. *O e Ω se encontram*
2. Se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$, então $f(n) = \Theta(h(n))$. *Sim, pois todos f(n) e h(n) podem crescer na mesma ordem de grandeza*
3. Se $f(n) = \Omega(g(n))$ então $g(n) = O(f(n))$. *f(n) e h(n) podem crescer e c1 < c2 + b*
4. Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$ então podemos afirmar que $f(n) = O(h(n))$. *f(n) = O(h(n))*

A resposta correta para estas assertivas é:

- A Todas as assertivas são verdadeiras.
 B Há somente três assertativas verdadeiras.
 C Há somente duas assertivas falsas.
 D Nenhuma das respostas anteriores.

$$f(n) = \Omega(g(n)) \rightarrow$$

$$g(n) = \Theta(h(n))$$

$$\underbrace{c_1 \cdot g(n)}_{\Omega} \leq f(n) \leq \underbrace{c_2 \cdot g(n)}_{\Theta}$$

$$\underbrace{c_1 \cdot h(n)}_{O} \leq g(n) \leq \underbrace{c_2 \cdot h(n)}_{\Theta}$$

$$\underbrace{f(n)}_{\Theta} = \Theta(h(n))$$

QUESTÃO 9

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

- F X O problema da "Subsequência Crescente Mais Longa (LIS)" envolve encontrar o comprimento da subsequência mais longa de um vetor no qual os elementos estão em ordem crescente. A complexidade de tempo da abordagem de programação dinâmica para resolver o problema LIS para uma matriz de comprimentos é $O(n \log n)$. \times_{n^2}
- V A abordagem de programação dinâmica é usada quando a solução tem subestrutura ótima.
- V A programação dinâmica tem como característica a memoization.
- V A ideia da programação dinâmica é dividir um problema em subproblemas menores que tenham sobreposição.

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



F - V - V - V

B F - V - F - F

C F - F - F - V

D V - F - V - F

QUESTÃO 10

Uma subsequência de uma determinada sequência é apenas a sequência dada com alguns elementos (possivelmente nenhum ou todos) deixados de fora. Dadas duas sequências $X_m = x_0x_1 \dots x_{m-1}$ e $Y_n = y_0y_1 \dots y_{n-1}$ de comprimentos m e n respectivamente, com índices de X e Y a partir de 0. Seja $Z = \{Z_k\}$ o conjunto de todas as subsequências comuns de X_m e Y_n . O peso de uma subsequência é dado pela soma dos pesos dos elementos que a compõe. Considerando que $X_4 = acdb$ e $Y_6 = cabcda$, e peso(a)=1, peso(b)=3, peso(c)=2, peso(d)=1.

- A A maior subsequência comum de X_m e Y_n , em termos de tamanho, e a subsequência que possui maior peso são iguais. \times
- B A maior subsequência comum de X_m e Y_n , em termos de tamanho, possui 2 elementos. $F \quad \overset{a(c)d(b)}{ac} \overset{\text{de peso}}{\underset{\text{fazendo}}{cd}} \{ac, cd = 3\}$
- C Há 9 subsequências comuns de X_m e Y_n . \times
- D O peso da subsequência comum de X_m e Y_n com maior peso é igual à 5. $cab = 5$

$$\begin{aligned} a &= L \\ b &= 3 \\ c &= 2 \\ d &= 1 \end{aligned}$$

$$\begin{aligned} ab &= L+3=4 \\ ac &= L+2=3 \\ cd &= 2+1=3 \\ ad &= L+1=2 \\ cb &= 2+3=5 \end{aligned}$$

QUESTÃO 11

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

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

Qual a ordem correta, de cima para baixo, das respostas destas assertivas?



V - V - F - V - V - V



V - F - F - V - V - V



C F - F - V - F - F - F



D F - V - V - F - F - F

QUESTÃO 12

$F_1 F_2 F_3 F_4 F_5 F_6$

Seis arquivos F_1, F_2, F_3, F_4, F_5 e F_6 possuem 100, 200, 50, 80, 120, 150 registros respectivamente. Em que ordem devem ser armazenados para otimizar o desempenho. Suponha que os arquivos sejam acessados com a mesma frequência.

F3, F4, F1, F5, F6, F2

A ordem é irrelevante, pois todos os arquivos são acessados com a mesma frequência.

F1, F2, F3, F4, F5, F6

F2, F6, F5, F1, F4, F3

ocorre requencial \Rightarrow quanto + cedo um arquivo aparece, menor será o tempo médio p/ acende-la, já que os arquivos menores "atramoram" menos o acesso dos próx arquivos

QUESTÃO 13

mais implicação que um problema NP-completo só tem solução p/ que ele esteja em P, q (P=NP)

Seja a versão do problema de decisão *Interval Scheduling Problem* definido a seguir: Dado uma coleção de intervalos baseados no tempo, e um valor k , a coleção contém um subconjunto de intervalos não sobrepostos de tamanho no mínimo k ?

Analice cada uma das questões a seguir.

1. Escalonamento de intervalos \leq_p Conjunto independente? $\text{NP-Completo} \Rightarrow$ cada intervalo diferente é uma coleção independente
2. Escalonamento de intervalos \leq_p Cobertura de vértice? $\text{NP-Completo} \Rightarrow$ cada intervalo...
3. Conjunto independente \leq_p Escalonamento de intervalos? $\text{NP-Completo} \Rightarrow$ X não conhecido

Problemas em P podem ser reduzidos p/ qualquer outro problema pg P é tão fácil quanto NP

Para cada uma das questões acima, decida se a resposta é (i) verdadeiro, (ii) falso, ou (iii) não conhecido.

Os itens (1) e (2) são falsos.
 O item (3) é verdadeiro.

Nenhuma item é verdadeiro.
 Os itens (1) e (2) são verdadeiros.

QUESTÃO 14

Qual é a ideia principal por trás do problema de escalonamento de intervalos?

- Ordenar os intervalos por tempo de inicio e selecionar aquele que começa primeiro. X
- Gerar todos os subconjuntos possíveis de intervalos e selecionar aquele com mais intervalos. X
- Ordenar os intervalos por tempo de término e selecionar aquele que termina primeiro.
- Selecione um subconjunto de intervalos que não se sobreponham. O

Interval Scheduling
↓
tempo de janela
mas não podem se sobrepor

QUESTÃO 15

Deseja-se encontrar o comprimento da subsequência comum mais longa (LCS) de X_m e Y_n como $l(m, n)$, onde uma definição recursiva incompleta para a função $l(i, j)$ para calcular o comprimento do LCS de X_m e Y_n é fornecida abaixo:

$$l(i, j) = \begin{cases} 0 & \text{if either } i = 0 \text{ or } j = 0 \\ \text{expr1} & \text{if } i, j > 0 \text{ and } X_{i-1} = Y_{j-1} \\ \text{expr2} & \text{if } i, j > 0 \text{ and } X_{i-1} \neq Y_{j-1} \end{cases}$$

→ se forem iguais pega ($i-1, j-1$) + 1
→ se forem diferentes, é o maior entre ($i-1, j$) e ($i, j-1$)

Analise as assertivas a seguir para identificar qual delas está correta de acordo com a definição de LCS:

expr2 = $\max(l(i-1, j-1), l(i, j))$
 expr2 = $\max(l(i-1, j), l(i, j-1))$

expr1 = $l(i-1, j) + 1$
 expr1 = $l(i, j-1)$

► Redução polinomial
► Recursões

► K-Complexo

► Verificar divisão - congruência e
faz + questões diviso.

Exercício 2 (2 pontos) – entregar ao final da aula pelo menos o resultado

Considerar os seguintes somatórios:

$$\sum_{i=0}^{n-1} i = c \times (x_1 - x_0 + 1) \quad (1)$$

$$\sum_{i=0}^n i = \frac{n(n+1)}{2} \quad (2)$$

$$\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad (3)$$

$$\sum_{i=0}^n i^3 = \frac{n^2(n+1)^2}{4} \quad (4)$$

1. Encontre a função de complexidade, assim como sua ordem de complexidade, das seguintes funções, com relação ao número de operações realizadas.

(a) // Operação a ser contabilizada: *

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++){
        C[i][j] = 0;
        for (int k = 1; k <= n; k++)
            C[i][j] = C[i][j] + A[i][k]*B[k][j];
    }
```

(b) // Operação a ser contabilizada: ++

```
for (int i = 1; i < n; i++)
    for (int j = i+1; j <= n; j++)
        for (int k = 1; k <= j; k++)
            cont++;
```

(c) // Operação a ser contabilizada: ++

```
for (int i = 1; i < n; i++)
    for (int j = i+1; j <= n; j++){
        for (int k = 1; k <= j; k++)
            cont++;
        for (int k = 1; k <= j; k++)
            cont++;
    }
```

(d) // Operação a ser contabilizada: cout

```
void misterio1(int n){
    cout << "Oii!!";
}
void misterio2(int n){
    if (n%2 == 0)
        misterio1(n);
```