Exame de Qualificação

06/07/2015

Nome: [redacted]

Dicas gerais: Leia todas as questões antes de começar; sempre justifique a sua resposta. A avaliação levará em consideração a abrangência e a profundidade demonstradas nas soluções apresentadas. Nas questões algorítmicas espera-se uma prova da corretude do algoritmo e uma análise do tempo em função do tamanho da entrada. As respostas podem ser breves e focadas na questão. Por favor apresente pseudo-código, caso seja necessário, nunca código explicito.

Questão 1 (2.5 pontos) Na área de Computação, um conceito fundamental é de função efetivamente calculável. Duas caracterizações importantes deste conceito são a *classe de funções Turing computáveis* e a *classe de funções recursivas parciais*.

a) Defina essas classes de funções

b) Faça um esquema da prova de que estas classes são equivalentes, explicando o que deve ser provado em cada passo.

c) Explique a importância desta equivalência.

Questão 2 (2.5 pontos) Em Computação muitas vezes resolvemos um problema usando uma solução de um outro problema. Para isso, usamos o conceito de *redução*. Disserte sobre redução de problemas, incluindo os seguintes tópicos, entre outros que você julgue importantes (identifique claramente no seu texto onde estes tópicos estão desenvolvidos):

a) Redução de problemas no contexto de computabilidade de problemas: Defina, dê um exemplo de problema não-computável, e explique como se poderia usar este problema para provar que outros problemas são também não computáveis.

b) Redução no contexto da definição de classes de complexidade: Defina, e explique intuitivamente o que diz o Teorema de Cook-Levin, bem como sua importância.

c) Por que a questão P = NP? é relevante para a Ciência da Computação?

Questão 3 (2 pontos) Um caixeiro viajante descobriu um bairro com potenciais clientes. O bairro é organizado em forma de uma grade regular de tamanho $n \times n$. Para cada interseção de ruas o caixeiro sabe, se tem um potencial cliente ou não. Ele quer entrar no canto superior esquerda do bairro, se movimentar somente ou para direita ou para baixo, e sair do bairro no canto inferior direita. Propõe um algoritmo que determina o caminho que visita o maior número de potenciais clientes em tempo polinomial.

Questão 4 (2 pontos) Um colega propõe um novo algoritmo para multiplicar duas matrizes $A \in \mathbb{R}^{n \times m}$ e $B \in \mathbb{R}^{m \times k}$. Seja $m' = \lfloor m/2 \rfloor$ ele calcula

$$A \times B = A[1:n, 1:m'] \times B[1:m', 1:k] + A[1:n, m'+1:m] \times B[m'+1:m, 1:k]$$

onde $A[1:k, 1:l]$ é a submatriz de A com entradas a_{ij}, $1 \leq i \leq k$, $1 \leq j \leq l$. Caso $m = 1$ o produto é calculado diretamente.
Qual a complexidade de tempo pessimista assintótica desse algoritmo?

Questão 5 (1 ponto) Uma vara de comprimento n tem que ser cortada em n peças de comprimento 1. Cada corte pode cortar múltiplas varas. Propõe um algoritmo que consegue isso com o número mínimo de cortes. Qual o número exato de cortes mínimos em função de n?

EXAME DE QUALIFICAÇÃO 2015/1-2

ALUNO: MARCELO BRANDALERO

C: 193400

1) a) Uma função é TURING-COMPUTÁVEL se existe uma máquina de Turing que a computa.

↳ (essa definição deve ser expandida, dependendo do modelo

0.8 de máquina de Turing utilizado. A máq. de Turing computa a função se, ao iniciar a leitura do ~~ponto~~ símbolo mais à esquerda de uma sequência ininterrupta de '1' numa fita contendo somente isso, ela pára ao ler o símbolo mais à esquerda de uma seq. ininterrupta de '1's. → e ~~se~~ a pára

Uma função é RECURSIVA PARCIAL se ~~pode ser~~ faz parte do conjunto de funções $\mathcal{F}$ definido indutivamente como:

- As funções sucessor, constante 0 k-ária e projeção n k-ária estão em $\mathcal{F}$.
- Todas as funções obtidas por composição com elementos de $\mathcal{F}$ estão em $\mathcal{F}$ ($\mathcal{F}$ é fechado sobre Comp).
- Todas as funções obtidas através da recursão primitiva de f e g, $f \in \mathcal{F}$ e $g \in \mathcal{F}$ estão em $\mathcal{F}$.
- Todas as funções (parciais) obtidas com o operador de minimização μn (menor 'n' tal que a função é zero e def $\forall i < n$) ~~sobre~~ sobre funções em $\mathcal{F}$ estão em $\mathcal{F}$.

b) Para provar equivalência de dois conjuntos A e B, devemos provar duas coisas:

i) $A \subseteq B$ ii) $B \subseteq A$.

OU: se f é FRP ⟹ f é T.C. se f é T.C. ⟹ f é FRP.

A prova de (i) é mais simples, dada a construção indutiva do conj. de FRP. Basta mostrar que existe uma máquina de Turing que computa as funções iniciais, e então mostrar como construir uma MT que, dados f e g FRP, compute Comp[f,g], Pr[f,g] e Mn[f].

1

A prova de (ii) é mais complicada, visto que o conj dos máq. de Turing não é construído indutivamente. Para provar a proposição, deve-se construir uma ~~F~~RP que SIMULA o comportamento da máq. de Turing. Essa prova, conforme apresentada por Taylor no livro, requer 13 passos para a construção da ~~máq~~ função. Essencialmente, define-se ~~funções que~~ codificações para a fita, para o estado da máquina; define-se funções para determinar o próximo estado, qual o símbolo a ser escrito, qual a quantid de passos necessários à computação. Definido isso, constrói-se, utilizando composição, uma função RP que simula o comportamento de qualquer MT, e, portanto computa qualquer função Turing-Computável.

c) Esta equivalência permite afirmar propriedades ~~coisas~~ sobre as funções Turing-Computáveis que são ~~muito~~ mais simp de serem provadas utilizando-se FRP. Ou seja: todos os resultados já sabidos sobre F.T.C valem para F e vice-versa. Além disso, esse resultado (bem como outros envolvendo modelos de computação equivalentes máquina de Turing - Post, Markov, λ) fornece forte evidência para a validade da Tese de Church-Turing. A tese afirma, intuitivamente, que o modelo da máquina de Turing representa todas as funções computáv Se modelos diferentes, desenvolvidos de forma independ são equivalentes a Turing, então a tese se sustenta empiricamente.

0.5

2) Intuitivamente, o PRINCÍPIO DA REDUÇÃO afirma q se uma propriedade é válida para um conjunto maior, ela tem que valer para todo sub-conjunto. Dependendo do contexto, o princípio é aplicado de formas levemente distintas.

a) No contexto de COMPUTABILIDADE, usa-se o princípio da redução para provar que alguns problemas são insolúveis. Isso é enunciado formalmente como

Se $\begin{pmatrix} P_1 \subseteq P_2 \\ P_1 \text{ é não-computável} \end{pmatrix}$ então (P_2 é não computável

Isso é bastante evidente pois, caso P2 fosse computável então poder-se-ia usar a solução de P_2 em P_1. Mais formalmente, dizemos que um problema P_2 é redutível para P_1 se existe uma função f tal que ~~para~~ ~~...~~ tem-se $w \in P_1 \iff f(w) \in P_2$.

(note que problemas podem ser codificados como linguagens.

1

O princípio é usado, por exemplo, para mostrar que o PROBLEMA DA PARADA é não computável, sabendo-se que o PROBLEMA DA AUTO-PARADA é não computável. (pois, se o primeiro fosse computável, poderia-se utilizar uma solução dele para resolver a auto-parada, que é não-computável. Contradição). Similarmente, muitos outros problemas práticos (como o de desenvolver um programa que testa se o outro programa computa uma função) são redutíveis ao problema da parada, e usa-se isso para demonstrar que esses também são não-computáveis.

b) O teorema de COOK-LEVIN Afirma que ~~o pro~~ todo problema que está em NP pode-ser redutível em tempo polinomial para o problema da satisfatibilidade booleana (SAT), que é NP-Hard. O teorema foi o primeiro a demonstrar uma redução assim, e o primeiro a apresentar um problema NP-Completo ($\in$NP, $\in$NP-Ha

0.7

O teorema diz, então que SAT é tão (ou mais) difícil que qualquer problema em NP. Isso forneceu uma base para a busca de novos problemas NP-Compl agora, não é mais necessário mostrar que todo NP é redutível ao problema de interesse; basta demonstrar que SAT é redutível ao problema de interesse (claramente, a segunda proposição é menos genérica e mais simples de ser provada). Um dos trabalhos mais interessantes a usar o princípio de redução e COOK-LEVIN é o de Karp, que e 1972 apresentou 21 problemas NP-Completos, todos a partir de reduções ~~para~~ de SAT ou dos problemas previa apresentados. Hoje, graças a esses trabalhos, sabemos que muitos problemas interessantes (caixeiro viajante, ~~[illegible]~~ partições de conjuntos, etc) são NP-Completos.

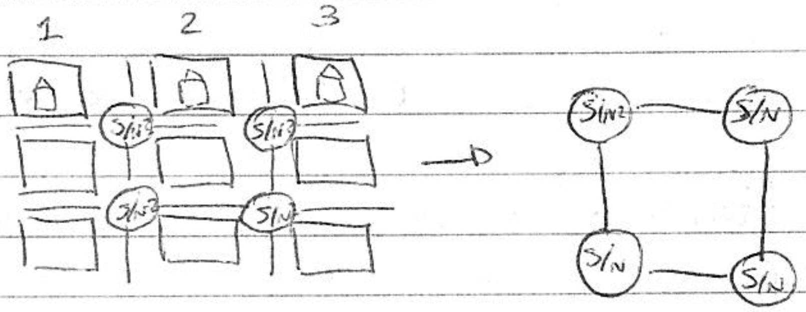
c) ~~Essa~~ A questão P=NP? é importantíssima para a ciência da computação. Apesar de todas as evidê apontarem para P≠NP, ainda não foi possível provar esse resultado. Caso P=NP, então haveria uma soluçã

0,5 em tempo polinomial para problemas que, há décadas, pessoas buscam uma solução. ~~Contudo~~, ~~caso se prova uma prova de P≠NP também~~ Uma questão interessante é a de como provar P≠NP. Resultados já sabidos afirmam que basta mostrar uma redução de um problema NP-Completo para um P, ou apresentar um algoritmo de tempo polinomial (em uma MT determinística) para um problema NP-Completo. Contudo, o fato de ninguém ter resolvido isso (ainda) sugere que necessitamos de uma teo mais desenvolvida para provar resultados como esse. Além disso, se P≠NP, devemos nos contentar que alguns problemas são, simplesmente, intratáveis para entradas muito grandes.

EXAME DE QUALIFICAÇÃO 2015/1-2

ALUNO: MARCELO BRANDALERO

C: 193400

3) 

Para resolver o problema, vou assumir que cada célula no grid representa uma intersecção de ruas. Assim, cada célula está associada a uma ~~decisão~~ isso é dado, ñ decisão (tem ou não tem cliente). O objetivo do problema, então, é encontrar um caminho de "↖" até "↘" que passe pelo maior número de células "sim", usando sempre a direção → ou ↓.

Note-se que o problema apresenta subestrutura ótima: seja $C(x,y) \in \{0,1\}$ o valor sim (1) ou não (0) de cada célula. ~~Então o~~ Seja $OPT(x,y)$ o maior número possível de clientes visitados ao chegar na célula (x,y). Observe que

$$OPT(x,y) = \begin{cases} C(1,1), & \text{se } x=y=1 \\ \cancel{\max} OPT(x,y-1)+C(x,y), & \text{se } x=1 \text{ e } y>1 \\ OPT(x-1,y)+C(x,y), & \text{se } y=1 \text{ e } x>1 \\ \max\{OPT(x,y-1), OPT(x-1,y)\}+C(x,y), & \text{se } x>1, y>1. \end{cases}$$

Isso sugere uma abordagem via programação dinâmica. Usando botton-up temos o seguinte algoritmo.

```
OPT (C) is
  for i in 1...n do
    for j in 1...n do
      if i=j=1 then OPT(i,j)=C(i,j)
      elseif i=1, j>1 then OPT(i,j) = OPT(i,j-1)+C(i,j)
      elseif j=1, i>1 then OPT(i,j) = OPT(i-1,j)+C(i,j)
      else then OPT(i,j) = max{OPT(i-1,j),OPT(i,j-1)}+C(i,j)
    done
  done
  return OPT(n,n)
```

O algoritmo usa uma tabela $n \times n$, então utiliza espaço $O(n^2)$. Similarmente, o uso de dois for aninhados, cada um com n iterações e operações de tempo $O(1)$ levam a uma complexidade $O(n^2)$, conforme solicitado.

2pt

4) ~~Seja $T(n,m,k)$~~ ~~o~~ Seja $T(m)$ o tempo necessário para multiplicar duas matrizes $A_{n \times m}$ e $B_{m \times k}$. Vamos assumir que n e k são constantes; assim, podemos (tentar) utilizar o método de Akra-Bazzi. A eq. de recorrência satisfaz

Tempo p/ somar duas matrizes $n \times k$.

$$T(m) = \underbrace{2 \cdot T(\lfloor m/2 \rfloor)}_{T(\lfloor m/2 \rfloor) + T(\lceil m/2 \rceil)} + \cancel{\Theta}(n \cdot k)$$

Usando o método de Akra-Bazzi, temos $k=1$, $g(u)=nk$, $a_1 = 2$, $b_1 = 1/2$ e $|h_1(m)| \leq 1$ (ou seja, podemos ignorar a função piso). Assim, temos p dado por

$$2 \cdot (1/2)^p = 1 \Rightarrow p = 1.$$

Aplicando no método $T(x) \in \Theta\left(x^p\left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)\right)$, temos

$$T(m) \in \Theta\left(m \cdot \left(1 + \underbrace{\int_1^m \frac{nk}{u^2} du}_{nk \cdot \frac{u^{-1}}{-1}\Big|_1^m}\right)\right) = \boxed{\Theta(m + m \cdot n \cdot k)}$$

$$nk \cdot \left(\frac{-1}{m} + 1\right)$$

$$\boxed{m + mnk - nk} = m \cdot \left(1 + nk - \frac{nk}{m}\right)$$

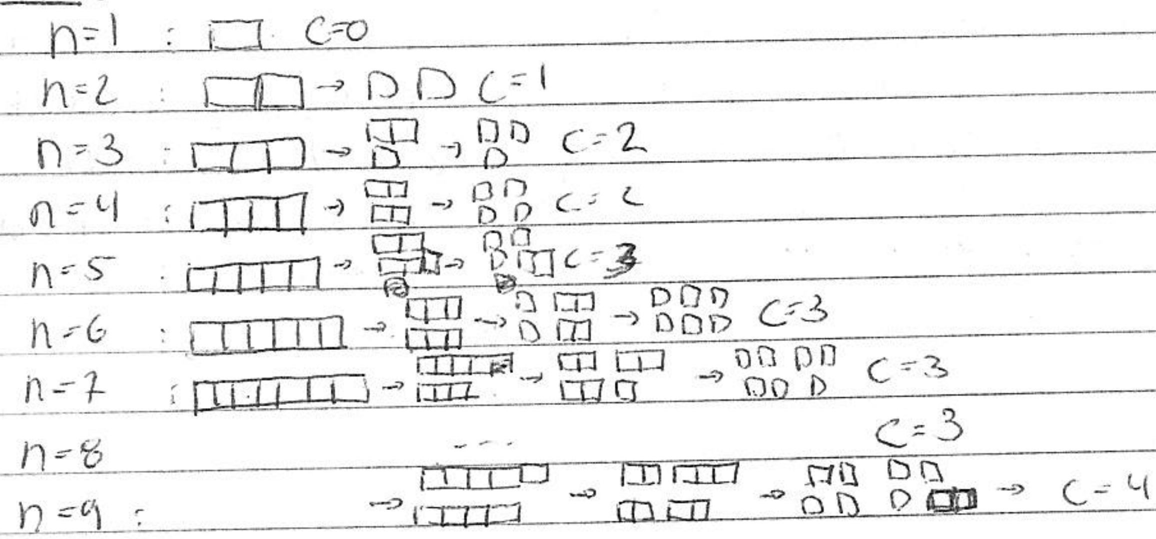
$$\boxed{\Theta(m \cdot n \cdot k)}$$

ESS. Correto c/ um erro

~~Note que, embora n, k afetem o tempo de ex, apenas m aparece na eq. pois consideramos n, k como constante~~

Teste:

Q5)



Uma análise por tentativas sugere que o número mínimo de cortes é $\lceil \log_2 n \rceil$.

Observe que, no primeiro corte, dividimos o problema em dois: um de tamanho $\lfloor n/2 \rfloor$ e o outro $\lfloor n/2 \rfloor + 1$. Essa divisão pode ser feita sucessivamente.

O algoritmo é

```
Corta(n) is
    Alinhe todas as varas à esquerda.
    Corte todas as varas na posição ⌈n/2⌉
    Corta(⌈n/2⌉).
```

Ou uma versão não recursiva

```
1. Corta(n) is
2.     ~~Enquanto~~
3.     max = n;    -- Tamanho da maior vara
4.     while (max > 1) {
5.         Alinhe todas as varas à esquerda
6.         Corte todas as varas na pos. ⌈max/2⌉
7.         max := ⌈max/2⌉
8.     }.
```

enar

2pt

Observe que o ~~algoritmo~~ número de cortes é o número de vezes que a linha 6 é executada, que é o número de vezes que o laço é executado. Mas esse número é precisamente $\lceil \log_2 n \rceil$, pois a cada iteração max se reduz a $\lceil max/2 \rceil$.

Alg. correto e análise quase correta. 0,75pt