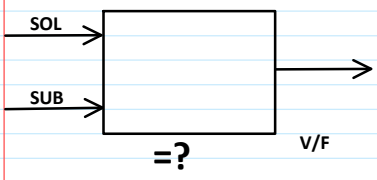
Solução Exame 2015/1-1

quarta-feira, 1 de julho de 2015 19:03

Questão 1



a) Por definição, uma função é dita *computável* se existe uma máquina de Turing que computa aquela função. Ou seja: f é computável se existe uma máquina de Turing que, ao receber como entrada um argumento *x* (expresso usando os símbolos da entrada da máquina), produz f(x) (expresso usando os símbolos de saída da máquina.
b) Assumindo que as entradas dos programas *sol* e *sub* são pertencentes a um conjunto infinito, PROG-CORRETO não é computável.

   Dem: note-se que a saída de PROG-CORRETO é verdadeira somente se, para todas as entradas para as quais *sol* produz um valor de saída, *sub* produz o mesmo valor. Existem infinitas entradas possíveis para o programa. Logo, testar sol(x) e sub(x) para todo x exigiria infinitos passos, e a computação nunca termina. Logo, a saída da função não pode ser *verdadeiro*, pois não foi testado sol(x) = sub(x) para todo x.

   Similarmente, a saída do programa é verdadeira somente se, para todas as entradas para as quais *sol* não produz um valor de saída, *sub* também não produz saída. Mas d eterminar se uma função produz um valor de saída é justamente o problema da parada, que é provado ser não computável.

   Logo, a função PROG-CORRETO não é computável pois é impossível verificar se a saída deve ser verdadeira oufalsa, ou seja, é impossível construir uma maquina de Turing que a compute.
c) PROG-CORRETO_mod é uma função que recebe duas funções totais *sol* e *sub* definidas sobre conjuntos finitos e retorna V, caso sol(x) = sub(x) para todo x e F caso contrário. Esta função é mais restrita que a anterior, pois sol(x) sempre está definido e é possível compará-lo com sub(x). Além disso, tal comparação para todo x pode ser concluida em tempo finito, já que existem apenas finitos x.

Questão 5

O problema apresenta uma subestrutura ótima, o que o torna ideal para ser resolvido através de programação dinâmica.

Considere o triângulo como formado por níveis, enumerados de [1..n]. Note que, no último nível, o valor da maior soma que utiliza o k-ésimo elemento daquele é o valor daquele elemento + a maior soma que utiliza algum dos pais daquele elemento. Em outras palavras,

Recursivamente, podemos definir a maior Soma que utiliza o k-ésimo elemento no n-ésimo nível como

```
Soma(n,k) = max{soma(n-1,k-1),soma(n-1,k)} + v(n,k) -- Se 1 < n e 1 < k < n
            soma(n-1, k) + v(n,k) -- Se 1 < n e 1 = k
            soma(n-1,k-1) + v(n,k) -- Se 1 < n e n = k
            v(1,1) -- Se 1 = n
```

O algoritmo recursivo para o problema é

```
MaiorSoma(n) {
    retorna max{Soma(n,k)} com 1 <= k <= n.
}
```

Utilizando programação dinâmica, o algoritmo fica

```
MaiorSoma(n) {
  S(1,1) = v(1,1);
  para i em 2...n faça
    para k em 1...i faça
      Se k = 1 então S(i,k) = S(i-1,k-1) + v(i,k)
```

Questão 2

São duas as principais classes de complexidade, P e NP.

A classe P refere-se a todos os problemas (funções) para os quais existe uma máquina de Turing determinística que o resolve em tempo polinomial. A classe NP refere-se aos problemas para os quais existe uma máquina de Turing não-deterministica que o resolve em tempo polinomial. Claramente, P está em NP, mas não se sabe se P é distinto de NP.

É importante se demonstrar a classe de complexidade de um problema para saber o quão tratável é o algoritmo. Problemas exclusivamente em NP não são solucionáveis para entradas muito grandes (a não ser em casos particulares), pois exigem um tempo que não é função polinomial da entrada.

Questão 3

1. Verdadeira. Se um problema está em P, então existe um algoritmo com complexidade O(n^c) que o computa, e O(n^c) < O(2^n).
2. Falso. Decidibilidade e complexidade são noções diferentes. Não existe um algoritmo que decida se uma sentença em lógica de primeira ordem é válida ou não.
3. Falso. Se L é NP-difícil, então L é tão ou mais difícil que qualquer problema em NP. Se L é polinomialmente redutível a L', então em tempo O(n^c) transforma-se L em L' preservando as propriedade da linguagem. Se L' está em P, então existe uma solução em tempo O(n^c) para L', e portanto para L. Neste caso, P = NP.

Questão 4

1 panqueca: 2 minutos
2 panquecas: 2 minutos
3 panquecas: 3 minutos
4 panquecas: 4 minutos
5 panquecas: 5 minutos
6 panquecas: 6 minutos
....

Note que, pensando em termos de recursos, é possível processar um máximo de 2 lados por minuto. Então, a cada instante n pode-se ter no máximo n panquecas prontas. Qualquer algoritmo que consiga atingir um tempo n para n panquecas é um algoritmo ótimo. Considere o seguinte algoritmo

```
Panquecas := [1...n]
i := n;

Para i > 3 ou i = 2 faça
  Ponha um lado da panqueca(i) na frigideira 1;
  Ponha um lado da panqueca (i-1) na frigideira 2;
  Aguarde 1 minuto enquanto os lados assam;
  Vire as panquecas nas duas frigideiras;
  Aguarde 1 minuto enquanto os outros lados assam;
  Panquecas (i) e (i-1) prontas
  i := i - 2;
açaf

Para i =  3 faça
  Ponha um lado da panqueca (i) na frigideira 1;
  Ponha um lado da panqueca (i-1) na frigideira 2;
  Aguarde um minuto enquanto os lados assam;
  Remova a panqueca (i) da frigideira 1;
  Ponha um lado da panqueca (i - 2) na frigideira 1;
  Vire a panqueca (i - 1) na frigideira 2;
  Aguarde um minuto enquanto os lados assam;
  Panqueca(i-1) pronta;
  Vire a panqueca (i - 2) na frigideira 1;
  Ponha o lado não assado da panqueca (i) na frigideira 2;
  Aguarde um minuto enquanto os lados assam;
  Panquecas (i) e (i - 2) prontas;
  i := 0;
açaf

Para i =  1 faça
```

```
MaiorSoma(n) {
    S(1,1) = v(1,1);
    para i em 2...n faça
        para k em 1...i faça
            Se k = 1 então S(i,k) = S(i-1,k-1) + v(i,k)
            Se 1 < k < i então S(i,k) = max{S(i-1,k-1),S(i-1,k)} + v(i,k)
            Se k = i então S(i,k) = S(i-1,k) + v(i,k)
        açaf
    afaç
    Retorna max sobre 0 < k < n de {S(n, k)}
}
```

```
    i := 0;
açaf

Para i =  1 faça
    Ponha um lado da panqueca (i) na frigideira 1;
    Aguarde um minuto;
    Vire;
    Aguarde um minuto;
    Panqueca 1 pronta;
    i := 0;
afaç
```

Solução Exame 2014/2-1

quinta-feira, 2 de julho de 2015 17:35

Questão 1

a) **Árvore geradora mínima?**
b) Binary heap : inserção custa O(log n) e remoção O(log n)
Percorrer lista de adjacência: O(n)
Percorrer matriz de adjacência:

Questão 2

Para resolver o problema por divisão e conquista, vamos usar a seguinte observação: se o problema tem tamanho n, e M é o elemento maioral, então M deve ser o elemento maioral em um subconjunto S de tamanho n/2 e ocorrer mais n/4 vezes no conjunto original - S. O algoritmo é

```
Maioral(S[1..n]) is
 If n = 1 then return S[1];
 Else
  count_M1 = 0; count_M2 = 0;
  M1 = Maioral(S[1..piso(n/2)])
  M2 = Maioral(S[piso(n/2)+1..n])
  For i in piso(n/2)+1..n do
    If S[i] = M1 then count_M1 = count_M1 + 1;
  done;
  For i in 1..piso(n/2) do
    if S[i] = M2 then count_M2 = count_M2 + 1;
  done;

  if count_M1 > ceil(n/2) return M1;
  if count_M2 > ceil(n/2) return M2;
 si
```

Usamos o método de Akra-Bazzi para determinar a complexidade do algoritmo. Temos k = 2, a1 = a2 = 1; b1 = b2 = 1/2 e g(n) = O(n). A complexidade resultante é Theta(n log(n)).

Questão 4

(i) A tese de Church-Turing afirma que o limite da computabilidade é expresso pela máquina de Turing, i.e., se f é uma função efetivamente calculável então existe uma máquina de Turing que a computa. (ii) O significado da tese para a ciência da computação é que (1) tem-se uma boa caracterização para o conceito informal de função efetivamente calculável (f é efetivamente calculável sss f é computável em máquina de Turing) (2) as funções efetivamente calculáveis podem ser enumeradas (já que todas estão associadas a pelo menos uma máquina de Turing) e (3) existem funções que não podem ser computadas em nenhum formalismo (já que máquina de Turing representa tudo o que pode ser computado). (iii) A tese não pode ser provada formalmente justamente por não existir uma definição formal para o conceito de "efetivamente calculável". A tese prove, justamente, uma caracretização desse tipo de função. (iv) Ela é aceita como verdadeira pois vários outros modelos de computação (máquina de Post, Funções recursivas parciais, calculo lambda).

Questão 6

a) Demonstração (por diagonalização). Suponha a existência de uma enumeração das funções totais f_0 f1 f2 ... . Considere uma matriz onde (i,j) é $f_i(j)$. Agora, considere uma função g(n) definida como

$$g(n) = \begin{cases} 7, se\ \ f_k(k) = 3. \\ 3, caso\ contrário \end{cases}$$

Note que g(n) é uma função bem definida para todos os argumentos. Logo, é uma função total e, por hipótese, consta na enumeração. Seja k a posição de g na enumeração, i.e. $f_k = g$. Agora, consideremos a avaliação de g para o argumento k. Se $f_k(k)$ = 3, então g(k) = 7. Mas com $f_k = g$, então 3 = 7, o que é uma contradição. Similarmente, se $f_k(k) \neq 3$ então g(k) = 3, e 7 = 3. A contradição surge ao colocar-se g(n) na enumeração. Mas se g(n) não estiver na enumeração, então também tem-se uma contradição, pois g(n) está bem definida e portanto é total. Logo, não pode existir uma enumeração de todas as funções totais e N

Questão 3

Note-se que o problema apresenta a propriedade de subestrutura ótima. Denote-se por N(i, c) o número de paradas necessárias a partir do i-ésimo posto, em uma situação com c kilometros de gasolina restantes (isto é, c = km/litro * litros de gasolina no tanque). Então

N(i,c) = N(i+1, c - d(i)), caso não se pare no posto i
N(i+1, c_max - d(i)) + 1, caso se pare no posto i.

Não. Dá para resolver por programação dinâmica, mas um algoritmo guloso é suficiente.

Considere o algoritmo a seguir.

```
cap = 0

for i in 0..n-1 do
 if d(i) > cap then
   stop(i) = T
   cap = 500 - d(i);
 else
   stop(i) = F
   cap = cap - d(i);
 fi
done
```

Vamos demonstrar a corretude do algoritmo usando uma estratégia "the greedy algorithm stays ahead". Seja O(i) uma solução ótima para o problema, considerando os primeiros i postos de gasolina. Seja Og(i) a solução obtida pelo algoritmo guloso. Assuma que o carro inicie a viagem com o tanque vazio, seja d(i) a distância do i-ésimo posto até o proximo e assuma que max d(i) < 500 . Vamos mostrar que

Og(i) <= O(i)

Dem. Por indução.
Base: O(0) = 1, pois o carro necessita estar de tanque cheio para fazer a primeira viagem. Similarmente, o algoritmo vai retornar stop(0) = T pois d(i) > 0.
P.I.: Assuma que Og(i) <= O(i). Vamos mostrar que a propriedade continua válida para i+1. Seja C(i) a quantidade remanescente de gasolina no tanque ao chegar no i-ésimo posto. Observe que, se d(i) < C(i), então não há a necessidade de parar, e O(i+1) = O(i). Este é precisamente o comportamento do algoritmo, de forma que Og(i+1) = Og(i) <= O(i) = O(i+1). Caso haja a necessidade de parar, emtão O(i+1) = O(i) + 1, e similarmente Og(i+1) = Og(i) + 1 <= O(i) + 1 = O(i+1).

Na verdade, mostrar que dist(i-ésima parada do ótimo) <= dist(i-ésima parada do guloso)?

Questão 5

a) São exemplos de propriedades não-triviais de uma função f computada por um programa M:
   a. M pára para uma entrada arbitrária w? Ou: o valor de f(w) está definido?
   b. M produz algum valor maior que 1000 para alguma entrada? Ou: existe w tal que f(w) > 1000?
   c. M é equivalente a outra máquina M' que computa f'? Ou: f(w) = f'(w), quando M e M' páram para a entrada w, e f(w) e f'(w) são ambas indefinidas quando as máquinas não páram?
b) O teorema de Rice é um resultado bastante surpreendente. Em geral, deseja-se uma garantia que um programa escrito seja correto, e que satisfaça certas propriedades. O teorema de Rice afirma que qualquer propriedade interessante (não trivial) sobre esse programa é não decidível - ou seja, não existe uma forma algoritmica de se verficar essa propriedade. Isso implica que, muitas vezes, deve-se analisar particularidades de um problema (programa) para demonstrar algumas propriedades.

para N e esse cojunto é incontável.

b) Note-se que o conjunto de todas as funções totais é subconjunto próprio do conjunto das funções parciais. Logo, o conjunto de todas as funções parciais também não é contável.
c) Demonstração. Pela Tese de Church Turing, o conjunto de todas as funções computáveis é exatamente o conjunto de todas as funções computáveis em máquina de Turing. Cada máquina de Turing pode ser codificada por um número de Gödel. Uma enumeração para as máquinas de Turing pode ser dada por m(n) : retorna o n-ésimo número de Gödel que codifica uma máquina sem repetições. Assim, tem-se uma enumeração das funções Turing-Computáveis e, portanto, das funções computáveis.
d) Totais é parte de parciais. Logo, se o maior é contável então o menor também é.
e) Se (a) e (d), então o conjunto de todas as funções não computáveis totais é incontável. Logo, existe um elemento.
f) Mesmo argumento acima, considerando (b) e (d)

Solução Exame 2014/1-1

sábado, 4 de julho de 2015 15:49

Questão 1

a) Funções recursivas parciais: o conjunto de funções definido indutivamente como:
   a. As funções iniciais sucessor, projeção e constante 0 k-aria estão nesse conjunto;
   b. Todas as funções que podem ser construídas utilizando as operações de composição, recursão primitiva e minimização estão no conjunto.
   c. Nada além do obtido pelos passos a e b está no conjunto.

   Já o conjunto de funções Turing-computáveis é o conjunto de todas as funções que podem ser computáveis em uma máquina de Turing.

   Para provar esse teorema, deve-se provar duas proposições ("lados")
   1. Toda a função que é recursiva parcial é computável em uma máquina de Turing

      Para mostrar isso, construímos a máquina de Turing que computa as funções iniciais e a que computa as operações de composição, recursão e minimização. Como toda FRP é construida usando esses blocos, compondo os blocos pode-se construir qualquer MT que compute aquela função.

   2. Toda função que é computável em máquina de Turing é recursiva parcial.

      Para mostrar isso, devemos apresentar uma forma de construir uma FRP a partir da MT. Isso é feito definindo uma FRP que simula a MT e executando essa função com o mesmo argumento de entrada da MT. Para isso, devemos definir uma codificação da entrada, do estado da máquina, funções para o proximo estado da máquina e da fita, e compor essas funções.

b) Esse teorema é importante pois fornece forte evidência para validade da tese de Church-Turing - os dois formalismos foram desenvolvidos de forma independente e chegaram no mesmo resultado. Além disso, fornece uma noção mais intuitiva de o que é função Turing-Computável (são todas as funções que são RP) e permite que se utilize para essas funções todos os resultados já provados para FRP - e vice-versa.

Questão 2

Redução é a transformação de um problema em outro com a prreservação das propriedades. Dizemos que um problema A é redutível a um problema B se uma solução para B pode ser transformada (eficientemente) em uma solução para A. No contexto da decidibilidade de problemas, podemos mostrar que um problema B é indecidível indiretamente: se A não é decidível, e A reduz para B, então B não é decidível. A é como se fosse um caso particular de B. Um exemplo é o problema da auto-parada, que não é decidível e reduz para o problema da parada. Então o problema da parada não é solucionável.
No contexto de classes de complexidade, fala-se de redução efetiva se um problema A para um problema B se A pode ser transformado em tempo polinomial para B. Usa-se a relação de redução para indicar que um problema pode ser resolvido com mais ou menos tempo que o outro.

Questão 6

Assuma que as tarefas estejam ordenadas por t_i. Considere o seguinte algoritmo.

0. $b \leftarrow \emptyset$
1. para $i = 1 \ldots n$ faça
2. se não colisão$(s_i \ldots t_i, b)$
3. retval += 1
4. $b \leftarrow b + s_i \ldots t_i$
5. O_g = retval;

Vamos demonstrar a corretude do algoritmo utilizando uma estratégia "the greedy

Questão 3

Teorema de Rice: Informalmente, o teorema afirma que qualquer propriedade interessante (não-trivial) sobre a função computada por um programa não pode ser determinada algoritmicamente.
Formalmente, se F é um conjunto de funções e Mf o conjunto de máquinas de Turing que computam funções em F, então Mf é um conjunto não recursivo; i.e., pertinência de uma máquina M no conjunto Mf é não-decidível. A grande implicação desse teorema é que, em geral, não é um procedimento simples determinar se um determinado programa satizfaz uma propriedade. O Teorema diz que não existe uma forma algoritmica que funciona para todos os casos; mas casos particulares ainda podem ser analisados.

Teorema de Cook-Levin: O teorema afirma que qualquer problema em NP pode ser reduzido para SAT. A importância desse teorema é que foi o primeiro a provar a existência de um problema NP-Completo (\in NP e \in NP-Hard). Uma consequencia desse teorema é que, havendo uma solução de tempo polinomial para SAT, então haverá uma solução de tempo polinomial para qualqiuer problema em NP (implicando, também, P = NP). Outra consequencia interessante é que, existindo um problema NP-Completo, fica mais facil mostrar a existencia de outros: basta apresentar uma redução do problema NP-Completo para o outro problema (de fato, isso é feito por Karp o apresentar um paper em 1972 com 21 problemas NP-Completos).

Questão 4

a) Dijkstra - menor caminho entre s e os outros nós.
b) i)

4. $b \leftarrow b + s_i \ldots t_i$

5. O_g = retval;

Vamos demonstrar a corretude do algoritmo utilizando uma estratégia "the greedy algorithm stays ahead".

i: tarefas $1 \ldots i$

Prop: Se $O(i)$ é o número máximo (ótimo) de tarefas concomitantes, então $O(i) \leq O_g(i)$.

Dem: por indução.

Base: $O(0)$. Claramente $O(0) = 1$.

Também $O_g(i) = 1$. Logo, $O(i) \leq O_g(i)$.

P.I: Suponha que $O(k) \leq O_g(k)$ para $k = 1 \ldots i$. Vamos mostrar que $O(i+1) \leq O_g(i+1)$.

Note que

$$O(i+1) = \begin{cases} O(i), & \text{se tarefa } i \text{ colide com a } i+1 \\ O(i)+1, & \text{caso contrário.} \end{cases}$$

$\Rightarrow$ #soluções ótimas

Este é exatamente o comportamento do alg. guloso

Sol: mostrar que $t_g(i) \leq t_o(i)$.

Ou seja: num escalonamento ótimo, o recurso sempre é liberado antes.

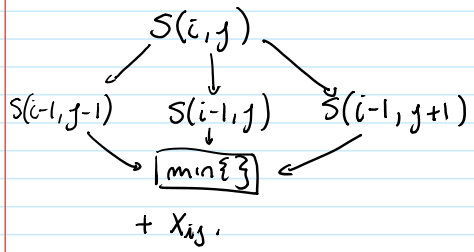
Se houvesse uma tarefa na solução ótima que não é contada pelo algoritmo guloso, então teria-se uma contradição

Solução Exame 2014/1-2

domingo, 5 de julho de 2015 11:24

Questão 3

Vamos usar uma matriz para guardar o valor da maior soma possível até aquela célula. Seja $S(i,j)$ esse valor. S pode ser definido recursivamente como (aqui: x_{ij}: valor da célula (i,j) na entrada, m: dimensões)



Ou seja:

$$S(i,j) = \begin{cases} x_{1j}, & \text{se } i = 1 \\ \min\{S(i-1,j), S(i-1,j+1)\} + x_{ij}, & \text{se } i > 1 \text{ e } j = 1 \\ \min\{S(i-1,j-1), S(i-1,j), S(i-1,j+1)\} + x_{ij}, & \text{se } i > 1 \text{ e } 1 < j < m \\ \min\{S(i-1,j-1), S(i-1,j)\} + x_{ij}, & \text{se } i > 1 \text{ e } j = m \end{cases}$$

A complexidade de espaço do algoritmo é $O(n \times m)$, pois é o tamanho da tabela necessária p/ guardar os $S(i,j)$.

↳ Na verdade, como cada nível necessita apenas do dado do nível anterior, não é necessário armazenar toda a tabela. A complexidade pode ser reduzida para $O(m)$

O algoritmo é

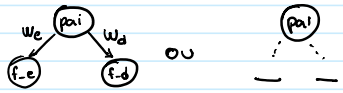
```
O(n)  for i in 1...n
        for j in 1...m
          if i=1 then S(i,j) = x_ij
          else
O(1)        if j=1 then S(i,j) = min{S(i-1,1), S(i-1,2)} + x_ij
O(1)        elseif j=m then S(i,j) = min{S(i-1,m-1), S(i-1,m)} + x_ij
O(1)        else then S(i,j) = min{S(i-1,j-1), S(i-1,j), S(i-1,j+1)} + x_ij
          end for
        end for
```

O(m)

A compl. tempo é $O(n \cdot m)$.

Questão 4

Considere um nó da árvore



Observe que, como a árvore é completa, somente um dos casos é verdadeiro. O menor caminho pode ser calculado recursivamente:

$$\mu C(pai) = \begin{cases} \min\{\mu C(f\_e) + w_e, \mu C(f\_d) + w_d\}, & \text{se pai não é folha} \\ 0, & \text{caso contrário} \end{cases}$$

Observe também que, se o tamanho do pai é n e a árvore é completamente balanceada, então o tamanho do filho é $n/2$. A eq. de recorrência é

$$T(n) = 2 \cdot T(n/2) + O(1).$$

Usando o método de Akra-Bazzi tem-se

$k=1$, $a_1 = 2$, $b_1 = 1/2$.

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

$$T(n) \in \Theta\left(n \cdot \left(1 + \int_1^n \frac{1}{u^2}\, du\right)\right)$$

$$T(x) \in \Theta\left(x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\, du\right)\right)$$

$$T(n) \in \Theta\left(n \cdot \left(1 + \frac{u^{-1}}{-1}\Big|_1^n\right)\right)$$

$$= \Theta\left(n \cdot \left(1 - n^{-1} + 1\right)\right) = \Theta(n).$$

Questão 1

Usando MT: Se P é válida para todas as funções computáveis, e assumindo-se a CTT, então toda a função computada por uma máquina de Turing satisfaz P.