Editorial de projeto e análise de algoritmo 2017.1

Ufal - Universidade Federal de Alagoas
IC - Instituto de Computação

Prof. Rodrigo Paes
Alfredo Lima
Nelson Gomes

AB 1 **3**
Prova Teórica 3
Solução da prova teórica 4
Solução da prova prática 7

AB2 **8**
Prova Teórica 8
Solução da prova teórica 9
Solução da prova prática 12

Reavaliação AB1 **13**
Prova Teórica 13
Solução da prova teórica 14
Solução da prova prática 16

Reavaliação AB2 **17**
Prova Teórica 17
Solução da prova teórica 17
Solução da prova prática 19

Final **20**
Solução da prova 20

AB 1

Prova Teórica

1. Suponhamos que o pivot da partição do QuickSort é sempre escolhido como o elemento mais a direita do subvetor corrente. Neste caso, se um vetor de entrada de tamanho 2*n* é tal que $a_n < a_{n+1} < a_{n+2} < \ldots < a_{2n}$ e $a_k < a_n$ para k=1,2,…,n-1. Qual é a complexidade do QuickSort nesta instância? Apresente a análise da sua resposta.
2. Prove se as afirmativas abaixo são verdadeiras ou falsas por indução:
   a. 2^(2n) - 1 é divisível por 3?
   b. 1 + 3 + 5 + ... + (2n-1) = n²
3. Analise as recorrências abaixo e determine a complexidade. Utilize o Teorema Mestre quando possível e justifique a escolha do caso:
   a. T(n) = lg(n)T(n/2) , sendo n potência de 2, lg função logarítmica na base 2 e T(1) = 1
   b. T(n) = 25T(n/5) + n
   c. T(n) = 2T(n/4) + raiz(n)
   d. T(n) = 3T(n-1)
4. O código abaixo é um algoritmo para multiplicação de dois números (u e v com n dígitos), analise e determine a complexidade:

KARATSUBA $(u,\ v,\ n)$

se $n \leq 3$
    então devolva $u \times v$
    senão $m \leftarrow \lceil n/2 \rceil$
        $a \leftarrow \lfloor u/10^m \rfloor$
        $b \leftarrow u \bmod 10^m$
        $c \leftarrow \lfloor v/10^m \rfloor$
        $d \leftarrow v \bmod 10^m$
        $ac \leftarrow$ KARATSUBA $(a,\ c,\ m)$
        $bd \leftarrow$ KARATSUBA $(b,\ d,\ m)$
        $y \leftarrow$ KARATSUBA $(a+b,\ c+d,\ m+1)$
        $x \leftarrow ac \cdot 10^{2m} + (y - ac - bd) \cdot 10^m + bd$
        devolva x

5. Os paradigmas de programação estudados até agora foram o DC e método do guloso. Disserte sobre eles de acordo com a facilidade de implementação, a análise de complexidade e sobre o método em si.

Solução da prova teórica

1.

(1): tam $= 2n$, $a_n < a_{n+1} < \cdots < a_{2n}$ e $a_K < a_n \ \forall K \in [1, n-1]$

Neste caso: $T(2n) = T(1) + T(2n-1) + 2n$

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

$$T(2n) = 3T(1) + T(2n-2) + 3 \cdot 2n - 1 - 2$$

$$\vdots$$

$$T(2n) = KT(1) + T(2n-K) + K\,2n - \sum_{i=1}^{K-1} i$$

$$T(2n) = K + T(2n-K) + K\,2n - \frac{1}{2}K(K-1)$$

Vai até n.

$$T(2n) = n + T(2n-n) + 2n^2 - \frac{1}{2}n^2 + \frac{1}{2}n$$

$$T(2n) = n + T(n) + \frac{3}{2}n^2 + \frac{1}{2}n$$

Daqui para frente, teremos termos organizados randomicamente, e o QuickSort opera em $O(n \lg(n)) \sim O(n^2)$

$T(2n) = \frac{3}{2}(n^2 + n) + T(n) \rightarrow$ independente de ser $n \lg(n)$ ou n^2; a complexidade dominante ainda é n^2

$\boxed{O(n^2)}$ nesta instância do QuickSort.

2.

a.

12/4/2017 OneNote Online

quarta-feira, 29 de novembro de 2017 20:57

Queremos provar que: $P(K+1): 2^{2(K+1)} - 1 \bmod 3 = 0$.

$P(1): 2^{2\cdot 1} - 1 = 4 - 1 = 3$, e $3 \bmod 3 = 0$ ✓

$P(K): 2^{2\cdot K} - 1 \bmod 3 = 0$ assuma como verdade.

$P(K+1): 2^{2(K+1)} - 1 = 2^{2K+2} - 1 = 2^{2K} \cdot 2^{2} - 1 = 2^{2K} \cdot 4 - 1 =$

$= 2^{2K} \cdot (3+1) - 1 = \boxed{2^{2K} \cdot 3} + \boxed{2^{2K} - 1}$ Nossa hipótese □

Divisível por 3

https://onedrive.live.com/edit.aspx?resid=3D991B98A67AD9FE!536&cid=3d991b98a67ad9fe&app=OneNote 1/2

b.

2: B) $1 + 3 + 5 + \dots (2n - 1) = n^2$

I) Funciona para 1: $1 = 1^2$ ✓

II) Supõe que funciona para K: $\sum_{i=1}^{K} (2i - 1) = K^2$

III) Testa para K+1: $\ulcorner (K+1)^2 = K^2 + 2K + 1 \lrcorner$ *

$$\sum_{i=1}^{K} (2i - 1) + (2 \cdot (K+1) - 1) \stackrel{?}{=} (K+1)^2$$

$$K^2 + 2K + 1 \stackrel{?}{=} (K+1)^2 \ *$$

$$K^2 + 2K + 1 \stackrel{\checkmark}{=} K^2 + 2K + 1 \ \square$$

3.

a. Não podemos aplicar o Teorema Mestre, pois **a** não é uma constante

T(n) = lg(n) * lg(n/2) * lg(n/4) * … * lg(n/ 2^k) * T(1)

Aplicando a propriedade de lg(a/b) = lg(a) - lg(b), teremos:

T(n) = lg(n) * (lg(n) - lg2) * (lg(n) - lg4) * … * (lg(n) - lg(n/2)) * T(1)

T(n) = lg(n) * (lg(n) - 1) * (lg(n) - 2) * … * (1), Se lg(n) chamarmos de K

T(n) = K * (K-1) * (K-2) * … * 1 -> K! = (lg(n))!

b. Aplicando o Teorema Mestre: Dado lg(5,25) = log(25) na base 5

$n^{lg(5,25)} > n$, então: $O(T(n)) = n^2$, pois temos 1º caso.

c. Aplicando o Teorema Mestre: Dado lg(4,2) = log(2) na base 4

$n^{lg(4,2)} = \sqrt{n}$, então: $O(T(n)) = \sqrt{n}\, lg(n)$, pois temos 2º caso.

d. Não podemos aplicar o Teorema Mestre, pois **b** = 1.

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

$$T(n) = 3^k T(n-k),\ O(T(n)) = 3^n$$

4. O algoritmo divide o problema em três subproblemas (dois de tamanho n/2 e um de tamanho n/2 + 1) e sabemos que o custo de juntar é cn (Algoritmo de soma ou subtração de números). Dado: lg(2,3) = log3 na base 2.

$$T(n) = 2\,T(n/2) + T(\,n/2 + 1\,) + cn$$

$T(n) \approx 3\,T(n/2) + cn$, utilizando o Teorema Mestre:

$n^{lg(2,3)} > cn$, então: $O(T(n)) = n^{1,58}$.

5. O paradigma dividir e conquistar consiste em pegar um problema e dividi-lo em subproblemas menores, após a resolução dos subproblemas manipulamos eles para encontrar a solução do problema.
O paradigma guloso consiste em cada passo pegar a solução ótima local para no fim encontrar o ótimo global, porém nem todos os problemas têm solução ótima pegando os ótimos locais.
Os algoritmos gulosos têm sua análise de complexidade mais fácil, porém a análise de corretude mais difícil (Normalmente precisamos provar por indução ou contradição se realmente o algoritmo encontrar a solução ótima).
Os algoritmos de dividir e conquistar têm como análise de complexidade uma função de recorrência a qual pode ser difícil de ser calculado, porém com o Teorema Mestre temos algumas facilidades.
Normalmente temos intuições fáceis para aplicar uma técnica gulosa e até fácil implementação, porém como falamos a sua prova não é trivial. Já aplicar uma técnica de dividir e conquistar é mais difícil, porém a implementação é fácil.

Solução da prova prática

1. Questão 1263 - Enésima raiz.
   Utilizar o princípio de dividir e conquistar para fazer a rápida exponenciação e busca binária para chutes. Use busca binária em **b** , onde $\mathbf{b^n} = \mathbf{a}$.
   Por exemplo: entrada a = 25 e n = 2.

Begin	**End**	**Mid(Chute)**	**Mid²(Exp)**	**Comparação**
0	25	12,5	156,25	Maior
0	12,5	6,25	39,0625	Maior
0	6,25	3,125	9,765625	Menor
3,125	6,25	4,6875	21,97265625	Menor
4,6875	6,25	5,46875	29,907226562	Maior
4,6875	5,46875	5,078125	25,787353516	Maior
4,6875	5,078125	4,8828125	23,84185791	Menor
		. . .		
4,99	5,01	5	25	Igual

2. Questão 1264 - Quase o menor caminho.
   Rode o Dijkstra uma vez para encontrar o menor caminho salvando o caminho. Sendo que podemos ir para o mesmo vértice com caminhos diferentes, basta o outro caminho ter o mesmo custo. Assim, remova todas as arestas que passe por um menor caminho. Rode o Dijkstra uma segunda vez e terá a resposta.

AB2

Prova Teórica

1. Descreva sobre os paradigmas de projeto de algoritmos: Programação dinâmica e programação aproximada. Dê exemplos.

2. Você sabe que é possível implementar a solução de um problema de programação dinâmica de forma bottom-up ou top-down. Descreva sobre as características de cada uma dessas formas de implementar, destacando os aspectos de dificuldade para implementar e a complexidade do algoritmo resultante. Também descreva sobre quais cenários cada abordagem é mais rápida na prática.

3. Escreva o pseudocódigo para um algoritmo de 2-aproximação para o problema do caixeiro viajante. Se você utilizar outros algoritmos, como por exemplo, um dijkstra, não pode dizer simplesmente: “rode um dijkstra”. Você também deve descrever o pseudocódigo dos algoritmos que for usar.

4. Discorra sobre <u>Exp</u>, <u>P</u>, <u>R</u>, <u>NP</u>, <u>NP-Difícil</u> e <u>NP-Completo</u>. E onde eles estão localizados na reta de dificuldade computacional e faça uma relação de conjuntos deles.

5. Assuma que P != NP, assinale a alternativa verdadeira e justifique as falsas.
   a. NP-Completo = NP
   b. NP-Completo ∩ P = ∅
   c. NP-Difícil = NP
   d. P = Exp - NP-Completo
   e. P = Exp - NP-Difícil

6. Você pediu para Ambrósio mostrar que um certo problema X é NP-Completo. Ambrósio fez uma redução em tempo polinomial de X para 3-SAT. Assinale a alternativa verdadeira e justifique as falsas.
   a. X é NP-difícil, mas não é NP-Completo
   b. X está em NP, mas não é NP-Completo
   c. X é NP-Completo
   d. X não é NP-difícil e não está em NP
   e. Nenhuma alternativa está correta

7. Assinale a alternativa verdadeira e justifique sua resposta.
   a. Se nós quisermos provar que um problema X é NP-Completo basta utilizar um problema NP-Completo conhecido Y e reduzir Y para X em tempo exponencial
   b. O primeiro problema NP-Completo foi o problema da satisfazibilidade de circuitos
   c. NP-Completo é subconjunto de NP-Difícil
   d. Todas as anteriores estão corretas
   e. Apenas b e c estão corretas

8. Faça um diagrama das classes de problemas considerando P == NP.

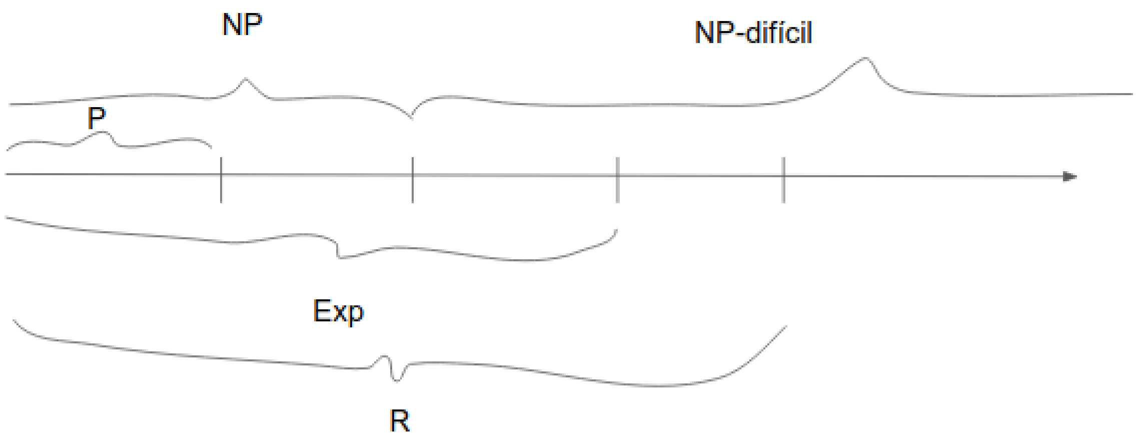
Solução da prova teórica

1. Programação dinâmica consiste em uma técnica de resolução de problemas que possam ser transformados em subproblemas e manipulando eles encontramos a solução para o problema (Semelhante ou dividir e conquistar), porém essa técnica tem mais dois princípios a otimalidade e a sobreposição de subproblemas. A otimalidade consiste em afirmar que quando um subproblema é resolvido, ele contém a melhor solução possível, então não precisamos recalcular os subproblemas (armazenando a resposta do subproblema). O princípio de sobreposição afirma que se "A resposta do ponto **A** para o **D** passa por **B** e **C** (nessa ordem), então se quisermos a resposta a resposta de **B** para **D** podemos afirmar que a resposta passa por **C**. Outro fato da programação dinâmica consiste em gastar memória para diminuir a complexidade do problema.
   Exemplo o problema de fibonacci, sem programação dinâmica tem complexidade 2^N e com programação dinâmica tem N, gastando N * (Bytes de um inteiro). de memória.
   Programação aproximada consiste em ter uma solução próxima da ótima. Aplicamos esta técnica quando para encontrar a solução ótima é muito custosa, assim com um outro algoritmo menos custoso encontrando a solução aproximada. Podemos classificar o quão bom é o algoritmo aproximado, por exemplo, se a resposta ótima de problema for X e o algoritmo aproximado encontra a resposta até 2*X, chamamos ele de 2-aproximação. Assim podemos afirmar de generica que um algoritmo aproximado pode ser classificado com k-aproximação.
2. Bottom-up consiste de solucionar os menores problemas e juntá-los para encontrar a solução de um problema maior.
   Top-down se assemelha a técnica de dividir e conquistar, que transformamos o problema em subproblemas.
   A dificuldade em aplicar a técnica bottom-up consiste em encontrar os menores problemas e manipulá-los (identificar as folhas), já top-down é descrever a função de recorrência (casos base e chamadas recursivas).
   A complexidade das duas técnicas são iguais, porém a bottom-up precisa encontrar a solução de todos os estados, enquanto a técnica top-down só necessita do "[caminho de solução]". Porém, como a técnica bottom-up é normalmente implementada de maneira interativa e a técnica top-down é implementada recursiva, na prática se precisamos calcular todos os estados a bottom-up é mais rápida do que a top-down, mas quando uma taxa "pequena de (quantidade de problemas para ter a resposta)/(quantidade de problemas totais) a técnica top-down é mais rápida.
3. O algoritmo 2-aproximação do caixeiro viajante possui algumas restrições: Ser um grafo completo, respeitar desigualdade triangular e o custo de a para b é o mesmo que b para a. Primeira rodamos algum algoritmo de MST, após geramos a árvore duplicamos suas arestas e escolhemos um vértice qualquer e rodamos um DFS adicionando o vértice no path no início do algoritmo e no fim dele. Após retiramos os ciclos do path temos a resposta.

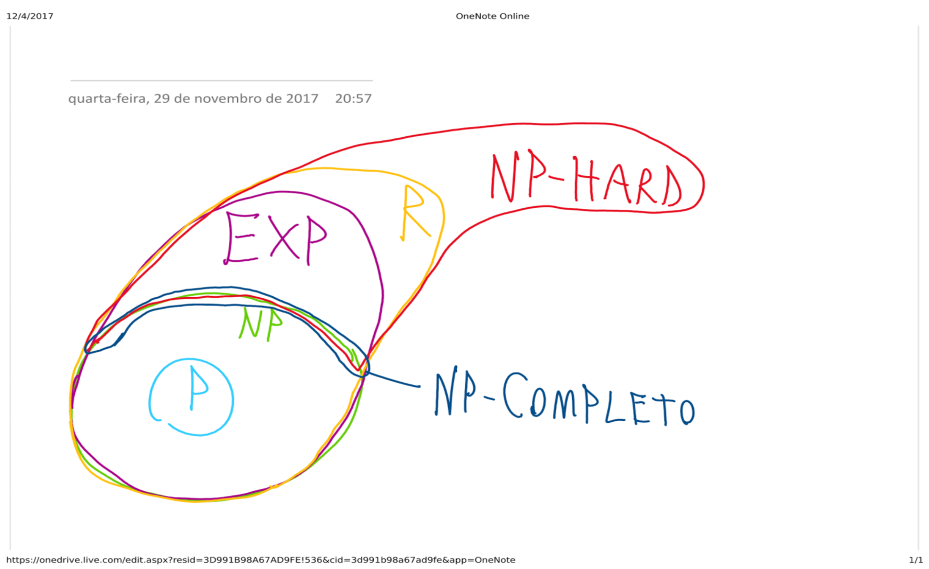
```
DFS(G,u) # G é grafo e u vértice
    P ← u
    para cada v em G[u]
       se v visitado
          marque v como visitado
           DFS(G,v)
    P ← u
```

```
PRIM(G) # G é grafo -> MST
  s ← seleciona-um-elemento(vertices(G))
  para todo v ∈ vertices(G)
    π[v] ← nulo
  Q ← {(0, s)}
  S ← ø
  enquanto Q ≠ ø
    v ← extrair-mín(Q)
    S ← S ∪ {v}
    para cada u adjacente a v
      se u ∉ S e pesoDaAresta(π[u]→u) > pesoDaAresta(v→u)
        Q ← Q \ {(pesoDaAresta(π[u]→u), u)}
        Q ← Q ∪ {(pesoDaAresta(v→u), u)}
        π[u] ← v
  retorna π( árvore geradora )
```

4. R - Conjunto de problemas resolvidos em tempo finito.
   P - Conjunto de problemas resolvidos em tempo polinomial.
   NP - Conjunto de problemas resolvidos em tempo polinomial com um algoritmo não determinístico.
   Exp - Conjunto de problemas resolvidos em tempo exponencial
   NP-Difícil - Conjunto de problemas que não podem ser resolvidos em tempo finito ou conjunto de problemas resolvidos com algoritmos não determinísticos.
   NP-Completo - Intersecção do NP-difícil com NP.



   NP-Completo está exatamente no segundo risco da esquerda para direita, NPC = NPH ∩ NP.



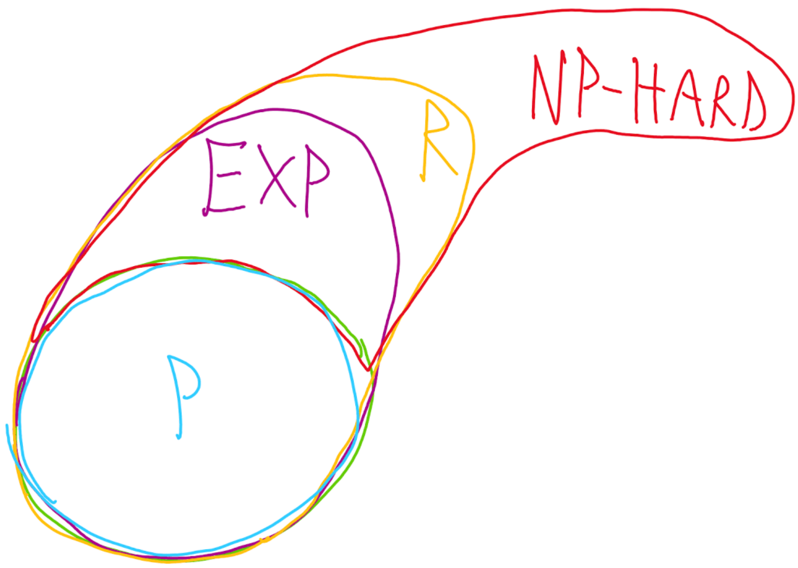
5.
   a. NPC = NPH ∩ NP e não somente NPC = NP
   b. Verdadeiro
   c. NP-Difícil ∩ NP = NPC.
   d. P ∈ Exp, porém como vemos na linha de complexidade da questão anterior P != Exp - Completo.
   e. NP - NPC = Exp - NP-Difícil.

6. Ambrósio só provou que X <= 3-SAT em complexidade, então X pode ser P, NP ou NP-Completo
   a. Não podemos afirmar
   b. Não podemos afirmar
   c. Não podemos afirmar
   d. Não podemos afirmar
   e. Verdadeira

7.
   a. Falso
   b. Verdadeiro, pois SAT foi o primeiro problema provado NPC.
   c. Verdadeiro, NP-Difícil ∩ NP = NPC
   d. Todas as anteriores estão corretas
   e. Verdadeiro

8. Note que: NP-Completo também é igual a P.

quarta-feira, 29 de novembro de 2017 20:57



Solução da prova prática

1. Questão 1848 Subconjunto complementar:
   Solução 1:
   Sabemos que temos dois subconjuntos **S’** e **S”**, chamando **X** = SUM(**S’**) e **Y** = SUM(**S”**). Podemos afirmar que **X** + **Y** = SUM(**S**) e **X** - **Y** = **D**, somando as duas equações temos 2**X** = SUM(**S**) + **D**, então basta procurar se em S existe alguma combinação que dê (SUM(**S**) + **D**)/2 ou (SUM(**S**) - **D**)/2 com o algoritmo da mochila 0-1.
   Solução 2:
   Suponha que **i** $\in$ [0,SUM(**S**)], basta encontrar alguma combinação que satisfaça a equação D = (SUM(**S**) - mochila(**i**)) - mochila(**i**), se encontrar então podemos dividir em dois subconjuntos que satisfaz a questão.

Reavaliação AB1

Prova Teórica

1. Prove que qualquer método de ordenação utilizando comparação tem complexidade Ω(n lg n). (3 Pontos)

2. Analise as recorrências abaixo e determine a complexidade. Utilizando o método proposto no item. (2 pontos)
    a. $T(n) = \sqrt{n} \times T(\, n/2 \,)$, Método da árvore
    b. $T(n) = 7 \times T(n/2) + n^2$, Método de substituição

Solução da prova teórica

1.

quarta-feira, 29 de novembro de 2017 20:57

Árvores de decisão são árvores binárias:

$a[i] < a[j]$

não — sim

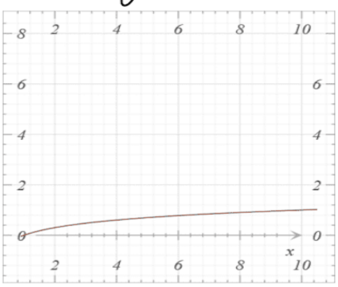
$a[7] \leq a[4] \leq \cdots$ $\quad\quad$ $a[3] \leq a[7] \leq \cdots$

Existem $n!$ respostas (Todas as permutações possíveis).

Daí: A altura é $lg(n!)$

$$lg(n!) = lg(n \cdot (n-1) \cdot \ldots \cdot 1) = lg(n) + lg(n-1) + \cdots + lg(1) = \sum_{i=1}^{n} lg(i)$$

$lg(x)$



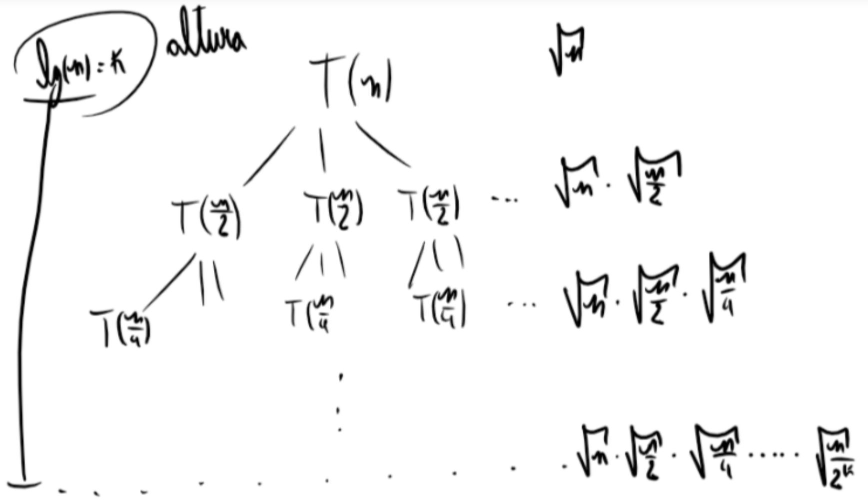
$$\sum_{i=1}^{n} lg(i) \geq \sum_{i=\frac{n}{2}}^{n} lg(i)$$

$$\sum_{i=\frac{n}{2}}^{n} lg(i) \geq \sum_{i=\frac{n}{2}}^{n} lg\left(\frac{n}{2}\right) = \sum_{i=\frac{n}{2}}^{n} lg(n) - \overbrace{lg(2)}^{1} = \frac{n}{2} \cdot lg(n) - \frac{n}{2}$$

Que é: $\Omega(n\,lg(n))$ $\square$

2.

a.



$$T(n) = \sqrt{n} \cdot \sqrt{\frac{n}{2}} \cdot \sqrt{\frac{n}{4}} \cdot \ldots \cdot \sqrt{\frac{n}{2^k}} = \sqrt{\frac{n^k}{1 \cdot 2 \cdot 2^2 \cdot 2^3 \cdot \ldots \cdot 2^k}} = \sqrt{\frac{n^k}{2^{0+1+2+\ldots+k}}} \qquad \left[\sum_{i=0}^{k} i = \frac{k \cdot (k+1)}{2} \right]$$

$$T(n) = \sqrt{\frac{n^k}{2^{k(k+1)/2}}} = \left(\frac{n^k}{\left(2^{\frac{(k+1)}{2}}\right)^k} \right)^{1/2} = \left(\frac{n}{2^{\frac{(k+1)}{2}}} \right)^{k/2} = \left(\frac{n}{\sqrt{n} \cdot \sqrt{2}} \right)^{k/2} = \left(\frac{\sqrt{n}}{\sqrt{2}} \right)^{k/2}$$

$$\boxed{T(n) = \left(\frac{n}{2} \right)^{\frac{lg(n)}{4}} \quad \square}$$

b.

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + n^2$$

Testando com $O(n^2)$: I) Hipótese: $T(k) \leq c k^2$, para $k < n$

II) Provar para n: $T(n) \leq 7 \cdot \left(c\left(\frac{n}{2}\right)^2\right) + n^2$

$$T(n) \leq 7c\frac{n^2}{4} + n^2 = n^2 \cdot \left(\frac{7c}{4} + 1\right)$$

Queremos provar que: $T(n) \leq c n^2$, daí: $\cancel{n^2}\left(\frac{7c}{4} + 1\right) \leq c \cancel{n^2}$

$$\frac{7c}{4} + 1 \leq c \therefore \frac{3c}{4} \leq -1 \therefore \boxed{c \leq -\frac{4}{3}}$$ Que não é válido!

Testando com $O(n^3)$: I) Hipótese: $T(k) \leq c k^3$, para $k < n$

II) Provar para n: $T(n) \leq 7 \cdot \left(c\left(\frac{n}{2}\right)^3\right) + n^2$

$$T(n) \leq 7 \cdot c\frac{n^3}{8} + n^2 = n^2 \cdot \left(\frac{7cn}{8} + 1\right)$$

Queremos provar que: $T(n) \leq c n^3$, daí: $\cancel{n^2}\left(\frac{7cn}{8} + 1\right) \leq c n^{\cancel{3}}$

$$\frac{7cn}{8} + 1 \leq cn \therefore \frac{1}{8}cn \geq 1 \therefore c \geq \frac{1}{8n} \therefore \boxed{c > 0 \; \square}$$ Que é válido!

Solução da prova prática

Problema 1860: Imposto Real

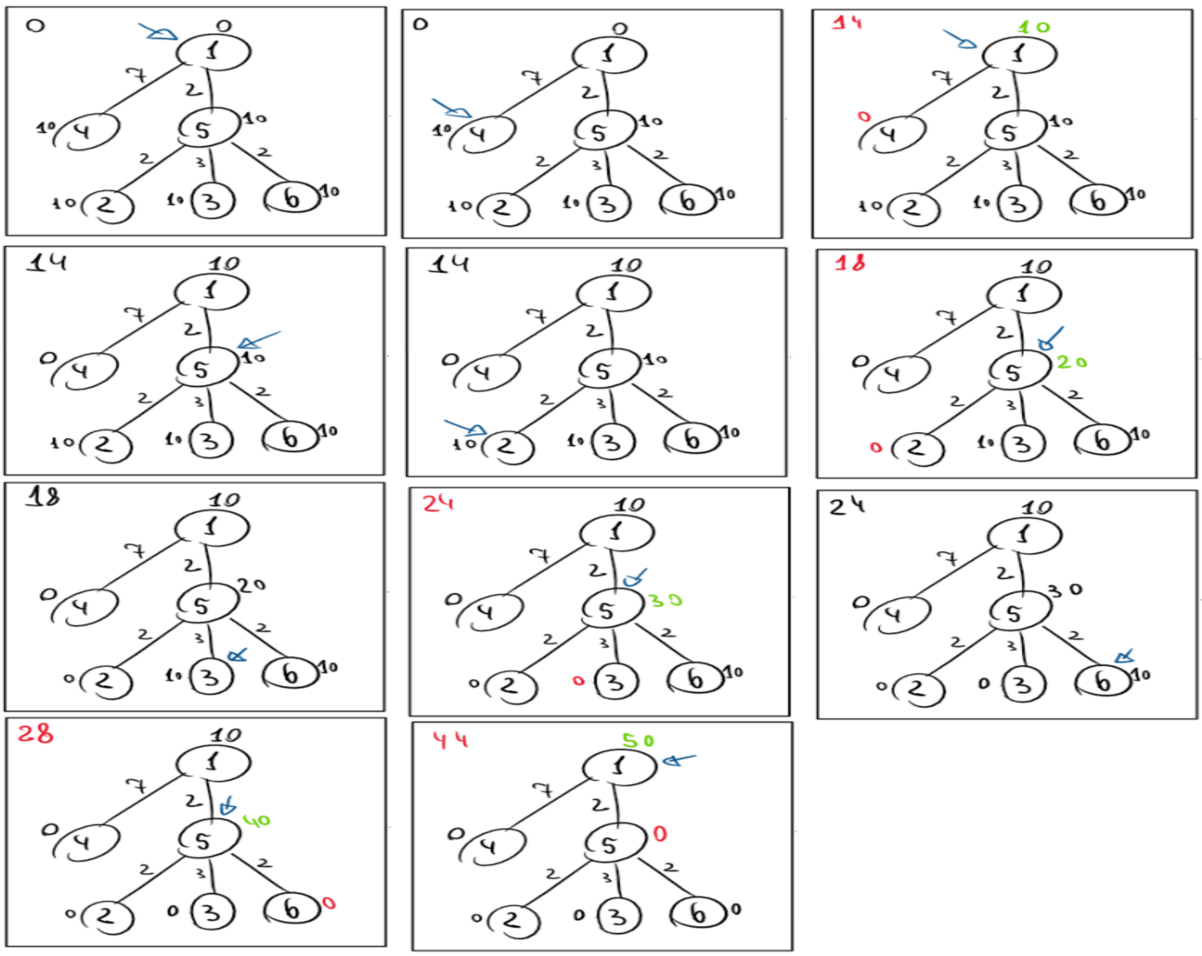
Só de ler a questão, a primeira intuição é de que devemos aplicar o Dijkstra de alguma forma. Mas, assim como dito na questão: Só existe um caminho possível para chegar em cada cidade (é uma árvore), logo: Não é necessário aplicar o Dijkstra (O(|V + E|*log(V))). Atente que toda cidade pode armazenar **todo** o ouro, e que a carruagem tem uma capacidade. Então, restam apenas duas opções: Pegar o ouro da cidade mais próxima e trazer para capital, ou pegar o ouro da cidade mais distante e trazer para cidade anterior (até que chegue na capital).

No primeiro caso: Ele teria que ir e voltar para cada cidade, sendo que quando fosse para as cidades mais distantes, ele teria que passar novamente por cidades vazias.

No segundo caso: Ele iria passar direto pelas cidades próximas, e quando voltasse; haveria a garantia de que todo o ouro já foi coletado, fazendo assim com ele passasse por cada cidade a menor quantidade de vezes possível.

Uma forma de fazer o segundo caso: É com um DFS (O(V + E), sendo que “E” é exatamente “V - 1”, nesse caso, ficando: O(V)).

Uma outra forma é observar isso como uma ordem topológica (Montando uma lista para cada nível), onde você primeiro encontra e remove as folhas do último nível e vai voltando para o primeiro nível.



Reavaliação AB2

Prova Teórica

1. Considere que, durante a análise de um problema de programação, tenha sido obtido a seguinte fórmula recursiva que descreve a solução do problema.

$$C(i,j) = \begin{cases} 0 & se \quad i = 0 \text{ ou } j = 0 \\ 1 + C(i-1, j-1) & se \quad 0 < i \leq M, 0 < j \leq N \; e \; i = j \\ \max\{C(i, j-1), C(i-1, j)\} & se \quad 0 < i \leq M, 0 < j \leq N \; e \; i \neq j \end{cases}$$

   Qual a complexidade da solução de busca completa e com programação dinâmica, qual é o gasto de memória em ambas soluções. Se a solução com programação dinâmica diminuir a complexidade, explique o motivo. (2 pontos)

2. O **problema da partição** é a tarefa de decidir se um determinado multiconjunto *S* de números inteiros pode ser particionado em dois subconjuntos de S_1 e S_2 , tais que a soma dos números em S_1 é igual à soma dos números em S_2. Prove que este problema é NP-Completo. (3 Pontos)

Solução da prova teórica

1. Essa modelagem assemelha-se a fibonacci, assim: é fácil notar que a complexidade é da ordem exponencial, assim como o gasto de memória. Para utilizar programação dinâmica: basta verificar que existiram N*M combinações, e podemos facilmente representar cada uma delas com uma matriz N por M. Assim, uma vez que foi executado para um x, y: Basta salvar o resultado na matriz e depois reutilizar. Fazendo com que a complexidade fique O(N*M), e o gasto de memória também.
2. Para provar que um problema é NP-Completo: Precisamos mostrar que ele é NP e NP-Hard, fazendo com que ele esteja na intersecção.
   a. NP: Para ser NP, basta podermos verificar se a solução é válida em tempo polinomial, porém sua solução é feita com algoritmo não determinístico. E isso é fácil para este problema, pois, tendo S_1 e S_2: basta somar cada elemento do subconjunto e verificar se a soma é igual, e isso pode ser feito em O(N) e como devemos construir dos os possíveis cenários, então cada elemento do conjunto S pode seguir dois caminhos de S -> S’ ou S -> S’’.

b. NP-Hard: Basta reduzirmos um problema NP-Hard ou NP-Completo para o problema da partição, e podemos fazer isso com o Subset Sum:

PAA

domingo, 30 de julho de 2017 22:20

Subset Sum para Partition

$S=\{a_1, a_2, \dots, a_n\}$ e target $= t$

Agora $\sigma = \sum_{i=1}^{n} a_i$, $a_{n+1} = \sigma + t$ e $a_{n+2} = 2\sigma - t$

Com $S' = S \cup \{a_{n+1}, a_{n+2}\}$, se tentássemos separar de tal forma que ficasse: $S \mid \{a_{n+1}, a_{n+2}\}$, teríamos: $\sigma \mid 3\sigma$.

Que não é uma partição válida, então, temos que separar a_{n+1} e a_{n+2}

O Partition vai fazer algo do tipo:

$S'' \cup \{a_{n+1}\} \mid (S' - S'') \cup \{a_{n+2}\}$, daí:

$\sum S'' + \sigma + t \mid \sum (S' - S'') + 2\sigma - t$, e:

$x + \sigma + t \mid (\sigma - x) + 2\sigma - t$, e:

$$x + \sigma + t = 3\sigma - x - t$$
$$\cancel{2}\sigma = \cancel{2}x + \cancel{2}t$$
$$\sigma = x + t$$
$$\boxed{x = \sigma - t}$$, daí:

$$\sigma - \cancel{t} + \sigma + \cancel{t} \mid (\cancel{\sigma} - (\cancel{\sigma} - \cancel{t})) + 2\sigma - \cancel{t}$$
$$\boxed{2\sigma \quad 2\sigma}$$

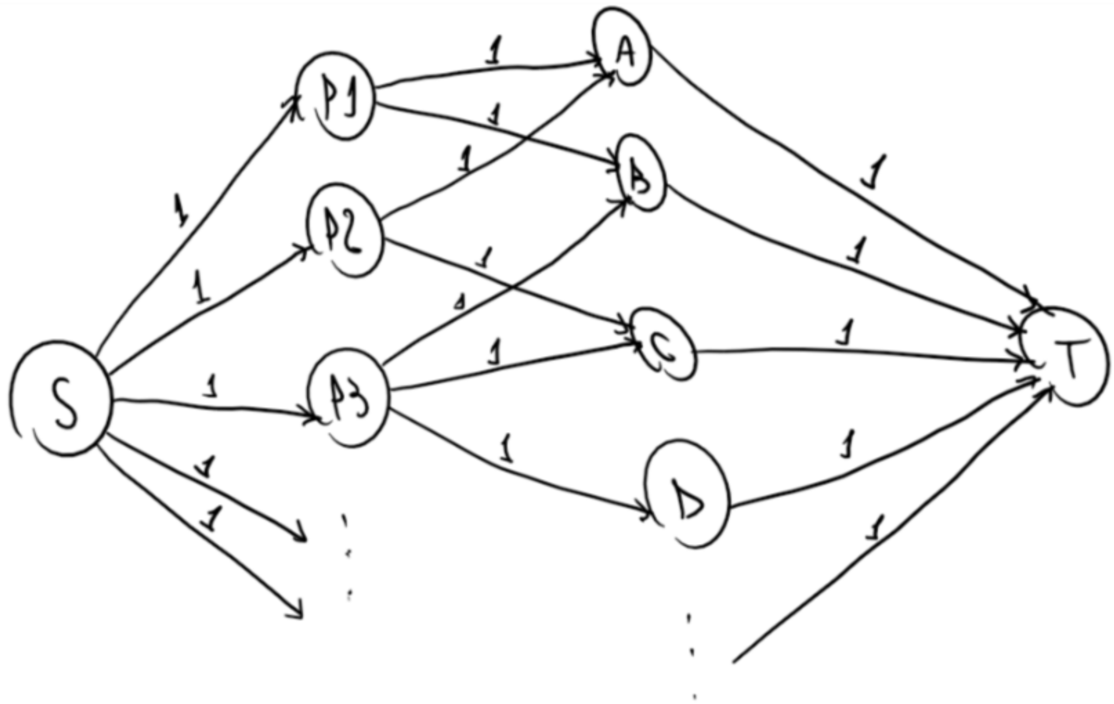
Veja que o conjunto: $S' - S''$ tem soma igual a t, resolvendo assim o Subset Sum, utilizando o Partition

E com isso, termino a redução do Subset Sum.

Solução da prova prática

Problema 1861: Nomeando Problemas

Uma forma de ver esse problema é como um problema de fluxo em um grafo bipartido.



Onde teríamos uma fonte (S), vários problemas (Pi), cada letra do alfabeto e um alvo (T). Para modelar o grafo:

1. Partimos da fonte para cada problema com uma capacidade 1, já que cada problema pode aparecer apenas uma vez.
2. Partimos de cada problema para as possíveis letras que ele pode ter, também com capacidade 1, pois ele vai poder ter apenas um nome.
3. Partimos de cada letra para o alvo, também com capacidade 1, já que cada letra vai poder aparecer apenas uma vez.

Assim: Basta aplicar algum algoritmo de fluxo máximo(Ford-Fulkerson, Dinic), e o algoritmo vai se encarregar de achar a melhor combinação. Depois, basta verificar as arestas de retorno das letras para com os problemas, e essa é a combinação nome - problema.

Final

Solução da prova

1. Resolva o problema abaixo (1981: Conjuntos Complementares 2), após resolvê-lo. Explique o motivo que utilizou técnica aplica na questão e analise detalhadamente a complexidade do seu algoritmo.

A menor diferença entre dois subconjuntos será justamente quando um deles tiver o somatório o mais próximo possível do somatório de S dividido por 2 (sum(S') mais próximo possível de sum(S)/2). Dessa forma, problema pode ser encarado como um caso particular do problema da mochila 0-1, neste outro problema, o objetivo é conseguir o maior valor possível sem que passe da capacidade da mochila. Se pensarmos similarmente, o problema dos conjuntos complementares poderia ter como capacidade o somatório de S dividido por 2, e os valores como cada *ai*, e então, o maior valor que pode ser "carregado" na mochila será o mais próximo possível da metade.
Nesta questão: também precisamos "criar" os itens da mochila. Para isso ordenamos o multiconjunto e encontramos os índices que não respeita as condições de v = u ou v + 1 = u. Utilizando programação dinâmica, a complexidade para executar essa "mochila" será: **O(N*sum(S))**, já que a complexidade da "mochila" é: O(N*Capacidade).
Complexidade:

Ordenação -> N*lg(N)
Encontrar os índices -> N
Gerar novos itens -> N
M = Quantidade de itens novos
Mochila -> M*sum(Mi)
Assim, O(N*lgN + N + N + M*sum(Mi)), aplicando as definições, temos:
O(N*lgN + M*sum(Mi))

2. Resolva o problema abaixo (1982: Aeroportos), após resolvê-lo. Explique o motivo que utilizou técnica aplica na questão e analise detalhadamente a complexidade do seu algoritmo.

"O menor custo de conectar todas as cidades", isso será uma árvore geradora mínima, a única coisa que precisaremos tratar é justamente o fato de que devemos preferir aeroportos à estradas. Se tivermos duas cidades, se o custo para fazer uma estrada entre elas for maior ou igual ao custo de construir um aeroporto: Então devemos construir um aeroporto, e nem precisaremos mais considerar essa estrada. Então, depois de tratar isso, basta rodar executar algum algoritmo de árvore geradora mínima enquanto houverem cidades desconexas. A complexidade para isso é: O(V*log(E)), no caso de utilizar Prim.