2ª Lista de Algoritmos e Grafos – 2022/1 - **GABARITO**
Prof. Vinícius Gusmão

1) Diga se a afirmativa é verdadeira ou falsa, apresentando justificativa.

a) O algoritmo de Dijkstra para caminhos mínimos a partir de origem fixa funciona para todos os digrafos que não possuem ciclos de custo negativo.

Falso. A existência de uma única aresta de custo negativo já é suficiente para levar o algoritmo de Dijkstra a uma resposta incorreta.

b) É possível resolver o problema da árvore geradora *máxima* para um certo grafo G rodando o algoritmo de Kruskal para árvore geradora mínima num grafo G' que tenha os mesmos vértices e arestas de G, mas onde o custo de cada aresta seja igual ao da aresta correspondente em G multiplicado por -1.

Verdadeiro. O algoritmo funciona normalmente com arestas de custo negativo. Sua prova de corretude não demanda que sejam não-negativas.

c) A técnica da programação dinâmica por memoização permite resolver problemas numa abordagem *top-down*, reduzindo sempre, em relação à abordagem tradicional de programação dinâmica *bottom-up*, a quantidade de soluções de subproblemas que precisam ser efetivamente computadas.

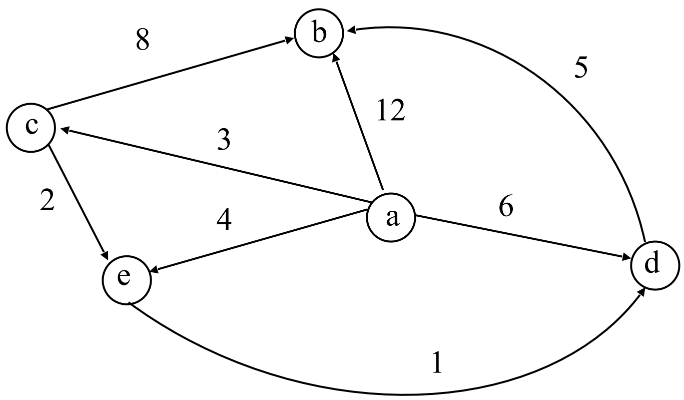
Falso. Nem sempre.

d) O laço do algoritmo seguinte, que descobre se um inteiro $x > 1$ é primo ou composto, é executado no máximo x-2 vezes, cada uma das quais demandando tempo constante. Portanto, trata-se de um algoritmo que roda em tempo polinomial no tamanho da entrada.

```
para todo j de 2 a x-1
    se x for divisível por j retorne ‘composto’
retorne ‘primo’
```

Falso. Trata-se de um algoritmo de tempo O(x), onde x é um número codificado por n = log_2 x bits. Sendo assim), O(x) é considerado pseudo-polinomial, já que é polinomial em um valor numérico, e não no tamanho da entrada. Como função do tamanho da entrada, ele é O(2^n), e portanto exponencial.

2) Seja o digrafo *D* abaixo. Deseja-se determinar os caminhos de custo mínimo entre o vértice *a* (origem fixa) e os demais vértices de *D*.



Suponha que o algoritmo dado a seguir será executado.

```
Entrada: custo[1..n,1..n] // matriz de adjacências
para cada vértice v de D: d[v] = ∞; ant[v] = null;
d[a] = 0; ant[a] = null;
seja L uma ordenação topológica para os vértices de D;
para cada vértice v, na ordem em que aparecem em L:
     para cada vértice w que é vizinho de saída de v:
          relaxar(v,w);
para cada vértice v de D: imprima v, d[v], ant[v];
```

a) Escreva em pseudocódigo a função *relaxar*, cuja entrada compreende os vértices incidentes à aresta que deve ser relaxada, bem como os vetores *d* e *ant* a serem (possivelmente) atualizados.

```
relaxar(x,y):
    if estimativa[x] + custo(x,y) < estimativa[y]:
            estimativa[y] = estimativa[x] + custo(x,y)
            antecessor[y] = x
```

b) Exatamente quantas chamadas a *relaxar* são feitas ao longo de sua execução quando a entrada é o digrafo *D* dado?

8 chamadas; uma por aresta.

c) O algoritmo acima funciona para todo digrafo acíclico e sem arestas negativas?

Sim. Sem problemas.

d) Suponha que, ao invés do algoritmo acima, o algoritmo de Bellman-Ford seja executado para esta entrada *D*. Quantas chamadas a *relaxar* são feitas ao longo de sua execução (desconsiderando a etapa final do algoritmo, que verifica a existência de ciclos de custo negativo)?

24 chamadas (3 iterações completas de 8 relaxamentos cada), se as arestas são relaxadas, a cada iteração, em ordem lexicográfica.

3) Simule a execução das 2 (duas) primeiras iterações do laço principal do algoritmo de programação dinâmica de Floyd-Warshall para caminho mínimo entre todos os pares de vértices, quando a entrada é o digrafo *D* da questão 2. Suponha os vértices considerados em ordem alfabética no laço principal do algoritmo.

   a) Ao fim da segunda iteração, quais os antecessores *ant*(*i,j*) e estimativas *d*[*i,j*] para o custo de um caminho mínimo entre cada par de vértices *i,j* de *D*?

	a	b	c	d	e
a	0/-	12/a	3/a	6/a	4/a
b		0/-			
c		8/c	0/-		2/c
d		5/d		0/-	
e				1/e	0/-

	a	b	c	d	e
a	0/-	12/a	3/a	6/a	4/a
b		0/-			
c		8/c	0/-		2/c
d		5/d		0/-	
e				1/e	0/-

Nada muda na primeira iteração, pois o vértice a é fonte (e portanto não "ajuda" nenhum caminho).

	a	b	c	d	e
a	0/-	12/a	3/a	6/a	4/a
b		0/-			
c		8/c	0/-		2/c
d		5/d		0/-	
e				1/e	0/-

Nada muda na segunda iteração, pois o vértice b é sumidouro (e portanto não "ajuda" nenhum caminho).

b) Escreva, em pseudocódigo, como é feito o teste e a eventual atualização de uma célula *d*[*i*,*j*] e *ant*[*i*,*j*], a cada iteração, em função dos valores anteriores guardados nas matrizes de estimativas e de antecessores.

Na k-ésima iteração, a célula (i, j) será atualizada da seguinte forma:

```
if d[i,k] + d[k,j] < d[i,j]:
    d[i,j] = d[i,k] + d[k,j]
    ant[i,j] = ant[k,j]
```

4) Seja *G* o grafo subjacente ao mesmo digrafo *D* da questão 2 (isto é, *G* é idêntico a *D*, exceto pelo fato de que suas arestas não são orientadas).

a) Simule a execução das 3 (três) primeiras iterações do algoritmo guloso de Kruskal para determinação de uma árvore geradora de custo mínimo com entrada *G*, apresentando a estrutura obtida ao fim de sua terceira iteração.

1a iteração: aresta e,d é adicionada
2a iteração: aresta c,e é adicionada
3a iteração: aresta acc é adicionada

A estrutura é uma floresta com as seguintes árvores:
(a—c—e—d)
(b)

b) Idem, trocando Kruskal por Prim, e considerando *c* como o vértice inicial.

Estado inicial: vértice c é adicionado à árvore que está sendo construída
1a iteração: vértice e é adicionado via aresta c,e
2a iteração: vértice d é adicionado via aresta e,d
3a iteração: vértice a é adicionado via aresta a,c

A estrutura é uma árvore com os vértices a, c, e, d.

5) Seja mais uma vez o digrafo *D* da questão 2, onde os pesos das arestas representam suas capacidades, no contexto do problema do fluxo máximo. ACRESCENTE A ARESTA b—>c COM CAPACIDADE 8, E MODIFIQUE A CAPACIDADE DA ARESTA a—>c PARA 8. Seja *a* a fonte e *c* o sumidouro da rede. Suponha que o método de Ford-Fulkerson esteja sendo aplicado e que, num dado momento, o valor

do fluxo na rede é igual a 7 unidades de fluxo, das quais 5 unidades estão indo da fonte ao sumidouro pelo caminho *a-c*, e 2 unidades estão indo pelo caminho *a-b-c*.

a) Desenhe o que seria, nesse momento, a rede residual.

A rede residual seria essa abaixo:

	a	b	c	d	e
a		10	3	6	4
b	2		6		
c	5	2			2
d		5			
e				1	

b) Aponte um caminho aumentante (qualquer) e indique qual seria seu efeito na atualização do fluxo atual na rede pelo algoritmo de Ford-Fulkerson.

Caminhos aumentantes:

a-c
a-b-c
a-d-b-c
a-e-d-b-c

Se o caminho aumentante escolhido for, por exemplo, a-d-b-c, seu gargalo seria a aresta d-b com capacidade 5. Aplicar este caminho aumentante, segundo Ford-Fulkerson, faria o fluxo total aumentar de 7 para 12 unidades.

6) Preciso ir de bicicleta da cidade A à cidade B, cidades estas ligadas por uma única estrada. Carrego na bicicleta um *bidon* (garrafinha) de 600 mililitros, e preciso ingerir 100 mililitros de água a cada 5 Km pedalados, do contrário desidrato e morro. Carrego comigo um mapa que indica a localização exata dos postos de combustível ao longo daquela estrada, locais esses onde posso beber água e encher minha garrafinha.

a) Dê um algoritmo guloso que garanta uma viagem com número mínimo de paradas para abastecimento.

Algoritmo: a partir da posição inicial na cidade A, a próxima parada será no posto mais longe possível da parada anterior que não exceda 35 Km de distância da parada anterior.

Em pseudo-código.

Entrada: um array com as posições de n postos ao longo da estrada, e a posição da cidade B. (A cidade A é assumida como estando na posição zero.)

Saída: os índices dos postos em que devo parar para abastecer.

```
stop_indexes = [ ]

current_pos = 0

next_stop_index = 0

pos[n+1] = pos[B]

i = 1

while i <= n+1:

    if pos[i] - current_pos > 35:

        if next_stop_index == 0:

            fail! // não há forma possível de completar o trajeto sem morrer

        stop_indexes += [next_stop_index]

        current_pos = pos[next_stop_index]

    else:

        next_stop_index = i

        i++

return stop_indexes
```

Complexidade: O(n).

b) Prove a corretude de seu algoritmo usando argumento do tipo *cut and paste*.

Seja A a saída fornecida pelo algoritmo guloso do item (a) para alguma instância de entrada que tenha conjunto viável não-vazio (se o conjunto viável for vazio, o algoritmo certamente indicará esse caso corretamente). Suponha, para efeito de contradição, que a solução A = [p1, p2, …, pk], k>=0, dada pelo algoritmo, embora claramente viável (por construção), *não seja uma solução ótima* para aquela instância do problema. Vamos agora inferir uma contradição.

Seja S = {T_i}, 0 <= i <= t, o conjunto com todas as t > 0 soluções ótimas para aquela instância, e seja T* dentre essas aquela que maximiza j, o índice da primeira divergência com a solução A. Isto é, T* contém p1, p2, …, p(j-1), mas não contém pj, e nenhuma outra solução em S contém prefixo de A com tamanho maior do que j-1. A figura abaixo ilustra esse prefixo comum entre A e T*, e o X vermelho indica que pj não está presente na solução T*.

```
          p1        p2      p3                  p(j-1)           pj          p(j+1)    …           pk

A:  _____|_________|______|______…_________|____________|___________|______ …_______|______

T*:_____|_________|______|______…_________|_______|___X_______|_________|__…___|________

                                                      q1             q2         q3       qm
```

Note que T* tem outros pontos de parada, à direita de pj, possivelmente coincidentes, mas não necessariamente coincidentes com pontos de parada da solução A. Note, também, que é necessário que exista um ponto de parada, na solução T*, *entre* a posição de p(j-1) e a posição de pj, uma vez que, pela construção gulosa do algoritmo, pj é o ponto *mais distante* à direta de p(j-1) que pode ser atingido, a partir de p(j-1), sem que o ciclista desidrate. Não seria possível, portanto, que na solução T* o ciclista reabastecesse em p(j-1) e depois apenas em algum posto à direita de pj. Chamaremos esse ponto de parada entre p(j-1) e pj, existente em T*, de q1.

Agora vamos construir T' da seguinte maneira: primeiramente, copiamos todos os pontos de parada de T* para T', e *forçamos* a inclusão do ponto de parada pj nesta solução T'que estamos construindo, como mostra a figura abaixo.

```
         p1        p2      p3                  p(j-1)           pj

   _____|_________|______|______…_________|_______|_____|_______|_________|__…___|________

                                                      q1             q2         q3       qm
```

Neste ponto, T' está claramente com um ponto de parada *a mais* em relação a T*, e, portanto, não é uma solução ótima (embora certamente viável). O que faremos agora será *removermos* de T' justamente o ponto de parada q1. Para isso, precisamos garantir que é possível remover esse ponto de parada sem afetar a viabilidade da solução. Ora, remover esse ponto de parada tem dois efeitos: (1) após reabastecer em p(j-1), o ciclista seguindo a solução T’ vai reabastecer apenas em pj, e não em q1, como teria feito se estivesse seguindo a solução T*; e (2) o ciclista chegará em q2 tendo reabastecido em pj, e não em q1. O efeito 1 é claramente ok, pois sabemos que a distância de p(j-1) a pj é *percorrível* pelo ciclista, uma vez que o ciclista precisa percorrê-la quando segue a solução (viável) A. O efeito 2 também não fere a viabilidade de T’, uma vez que a distância de pj a q2 será menor do que a distância de q1 a q2, que, por sua vez, é percorrível. Chegamos assim à solução viável T’ mostrada abaixo:

```
         p1        p2      p3                 p(j-1)            pj

T’:____|_______|_____|_____…_______|__________|______|_______|__…___|_______

                                                               q2          q3        qm
```

Finalmente, a solução T’ acima é viável, como já argumentado, *e é também ótima*, uma vez que tem exatamente a mesma quantidade de paradas de T* (pois apenas entrou pj e saiu q1). Ocorre que o tamanho do prefixo comum de T’ e de A, sendo maior ou igual a j, é pelo menos uma unidade maior que o do prefixo comum de T* e de A, que é j-1. E isso é uma contradição, pois T* tinha sido escolhida como sendo a solução ótima que maximizava o tamanho do prefixo comum com A.

Sendo assim, a solução A precisa ser ótima.