# MC458 - Projeto e Análise de Algoritmos I
# Prova Individual - 01/10/2018 (soluções ao final)

1. Comece pelas questões que você tem mais certeza de saber fazer.
2. Não perca tempo com detalhes menores. Foque no que é relevante para resolver a questão e passe logo para a próxima.
3. Se sobrar tempo, você pode acrescentar mais detalhes.

**Questão 1** (valor 2,5) Suponha que você receba como entrada a lista $L$ de arestas de uma árvore com $n$ nós, numerados de 1 a $n$. Cada aresta é um par de inteiros $(i, j)$ onde $i$ é o pai de $j$ na árvore. As arestas podem aparecer em qualquer ordem, mas todas as arestas aparecerão, e não haverá arestas na lista que não são da árvore. Seu objetivo é determinar todos os nós da árvore que não são raiz nem folha da árvore.

Faça um algoritmo eficiente para resolver este problema, e analise sua complexidade.

**Questão 2** (Valor 2,5) Seja $G$ um grafo com $n$ vértices onde todos os vértices têm grau maior ou igual a $n/2$. Considere a seguinte prova por indução de que exite um caminho simples (isto é, sem repetir vértices) em $G$ que usa $k$ vértices, para $k = 1, 2, 3, \ldots, n$.

**Prova:** Se $k = 1$, caminhos com apenas um vértice obviamente existem. Se $1 < k \leq n$, sabemos por hipótese de indução que existe um caminho $v_1, v_2, \ldots, v_{k-1}$. Se $v_1$ tiver algum vizinho fora deste caminho, ou $v_{k-1}$ tiver algum vizinho fora deste caminho, podemos acrescentar este vizinho antes de $v_1$ ou após $v_{k-1}$ e aumentar o caminho. Se todos os vizinhos de $v_1$ e de $v_{k-1}$ estiverem no caminho, então temos $k - 1 > n/2$ e, pelo princípio da casa do pombo, existe um $i$ tal que $v_1$ é vizinho de $v_{i+1}$ e $v_{k-1}$ é vizinho de $v_i$. Daí podemos transformar o caminho em ciclo:

$$v_1, v_{i+1}, v_{i+2}, \ldots, v_{k-1}, v_i, v_{i-1}, \ldots, v_2, v_1.$$

Como $k-1 > n/2$, pegue um vértice $v$ qualquer fora deste ciclo. Ele terá um vizinho $v_j$ no ciclo. Então podemos abrir o ciclo antes ou depois de $v_j$ e colocar a aresta $(v, v_j)$, criando um caminho de tamanho $k$. □

Faça um algoritmo eficiente que ache um caminho simples em $G$ que usa $n$ vértices, e analise seu algoritmo. Pode usar a prova acima neste algoritmo.

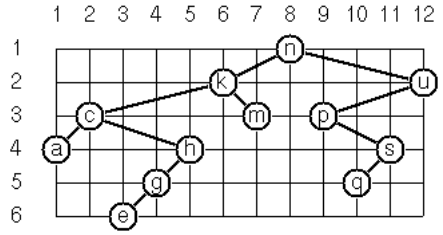
**Questão 3** (Valor 2,5) Um grupo de adolescentes e seu professor arrojado acabaram ficando presos numa caverna em um distante país asiático. Uma força-tarefa de resgate foi organizada. Juntaram-se $n$ experts em salvamento e a cada um deles foi feita a seguinte pergunta: *Quem é a pessoa dentro deste grupo de n em quem você mais confia?* Foram obtidas $n$ respostas, que serão a entrada do seu problema na forma de um vetor `confia[1..n]`, onde `confia[i]` é o número da pessoa em quem $i$ mais confia.

Os organizadores querem estabelecer a maior força-tarefa possível, mas com as duas condições seguintes:

- se $i$ estiver na força-tarefa, então a pessoa em quem $i$ mais confia também deve estar.
- para cada pessoa $j$ da força-tarefa, deve haver $i$ na força-tarefa tal que $j$ é a pessoa em que $i$ mais confia.

Faça um algoritmo eficiente para resolver este problema, e analise sua complexidade.

**Questão 4** (Valor 2,5) Como preparação para imprimir uma árvore binária de $n$ nós, é preciso calcular as coordenadas $x$ e $y$ de cada nó. Um possível esquema para isto está ilustrado na figura a seguir




Neste esquema, as coordenadas $x$ e $y$ dos nós são inteiros positivos, começando de 1. A coordenada $x$ de cada nó é igual à posição deste nó num percurso *in-order* da árvore. A coordenada $y$ de um nó é a sua profundidade mais 1, isto é, sua distância até a raiz mais 1.

Faça um algoritmo eficiente para calcular estas coordenadas para todos os nós de uma árvore dada, e analise sua complexidade.

Boa sorte!

# Soluções

**Questão 1** (valor 2,5)

**Algorithm 1** Retorna os nós que não são raiz nem folha.

```
1:  function INTERNOS(L)
2:      for i ← 1 to n do
3:          folha[i] ← true
4:          raiz[i] ← true
5:      for (i, j) in L do
6:          folha[i] ← false
7:          raiz[j] ← false
8:      resp ← lista vazia
9:      for i ← 1 to n do
10:         if not folha[i] and not raiz[i] then
11:             taca i em resp
12:     return resp
```

No começo todo mundo é raiz e folha (vai que a árvore não tenha arestas). Pai não é folha, e filho não é raiz: esta é a explicação para o loop das linhas 5 a 7. Depois, é só colher os nós certos.

Análise: O primeiro loop tem $n$ iterações. O segundo loop tem $|L|$ (tamanho de $L$) iterações. O terceiro loop tem $n$ iterações. Dentro de cada loop só há comandos simples, e fora dos loops também. Complexidade total: $O(n + |L|)$. Mas $|L| = O(n)$ por ser árvore. Então dá $O(n)$.

**Questão 2** (valor 2,5)

Veja um possível pseudocódigo no Algoritmo 2.

Para obter o caminho desejado, basta chamar Caminho$(G, n(G))$. O caminho é representado pela variável $p$. A complexidade é $(n^2)$ se observarmos os seguintes detalhes:

- O grafo terá uma matrix de adjacências com a qual se pode perguntar se dois vértices são ligados em $O(1)$. Se esta matrix não for dada, pode ser facilmente contruída no início gastando $O(n^2)$.
- Haverá um vetor para marcar quais vértices pertencem ao caminho, permitindo responder questões do tipo $v$ está em $p$ em $O(1)$.

**Algorithm 2** Retorna caminho Hamiltoniano (passa por todos os vértices).

```
1:  function CAMINHO(G, k)
2:      if k > n(G) then
3:          return "não há caminho tão longo"
4:      if k = 1 then
5:          return caminho de 1 vértice
6:      else
7:          p ← CAMINHO(G, k − 1)
8:          if existe vértice c fora de p ligado a extremo de p then
9:              adicione v a p na extremidade certa
10:         else
11:             encontre i com p[i] ligado a p[k − 1] e p[i + 1] ligado a p[1]
12:             transforme p em ciclo p[1] + p[i + 1..k − 1] + p[i..2]
13:             tome vértice v fora de p
14:             encontre p[j] ligado a v
15:             rompa o ciclo em p[j] e acrescente v a p
16:         return p
```

- O caminho será representado por uma coleção de arestas e mais duas variáveis para indicar os extremos, permitindo adicionar vértices uma extremidade em $O(1)$, bem como transformar em ciclo e cortar o ciclo em $O(n)$.

Com isto, podemos ver que a complexidade total será $O(n^2)$, pois a cada chamada recursiva gasta-se $O(n)$.

**Questão 3** (valor 2,5)

Esta questão é uma aplicação direta do algoritmo para achar subconjunto máximo onde uma função é 1-1 (ou bijetora). Pode ser feito em $O(n)$. A solução é idêntica à que está no artigo do Manber.

Houve gente que usou um método diferente, de achar ciclos no grafo, que também dá certo e com a mesma complexidade.

**Questão 4** (valor 2,5)

A chamada principal é COORDS($raiz, 1, 1$), onde COORDS é a rotina mostrada a seguir. A ideia é usar recursão. Para a coordenada `x`, a rotina recebe a primeira posição livre a usar, e retorna a próxima posição livre a usar. Para

a coordenada `y`, basta chamar para os filhos com 1 a mais do que o pai tem. Não é preciso retornar nada relativo a `y`.

Este algoritmo roda em $O(n)$, onde $n$ é o número de nós na árvore, pois cada nó é visitado apenas uma vez.

**Algorithm 3** Popula uma árvore binária com coordenadas `x` e `y` para a impressão de seus vértices.

```
1:  function COORDS(v, x, y)
2:      if v.left ≠ NULL then
3:          nx ← COORDS(v.left, x, y + 1)
4:      else
5:          nx ← x
6:      v.x ← nx
7:      v.y ← y
8:      if v.right ≠ NULL then
9:          return COORDS(v.right, nx + 1, y + 1)
10:     else
11:         return nx + 1
```