1. (•) ***I understand the course policies.***

   Describe and analyze an algorithm that either constructs a matrix that agrees with R and C, or correctly reports that no such matrix exists.

Solution: We first set all the elements in a matrix $M[1..n][1..n]$ to be 0. Then, we fill 1s into the matrix row by row. For row i, we iteratively choose some j such that j is the index of $\max_j C[j]$ where $M[i][j] = 0$, and set $M[i][j]$ to be 1 until $R[i]$ 1s are filled. Whenever an '1' is filled to $M[i][j]$, let $C[j]$ decrease by one. If $C[j] = 0$ for all j while $R[i] > 0$, we report no such matrix exists. After filling in all the rows, if there is some $j \in [1..n]$ such that $C[j] > 0$, we report no such matrix exists. Otherwise, $M[1..n][1..n]$ is the matrix that agrees with R and C.

Define an auxiliary function DECREASEINDEX($C[1...n]$) that returns an array $D[1...n]$ such that $C[D[1]] \geq C[D[2]] \geq \ldots \geq C[D[n]]$. We can use some $O(nlogn)$ sorting algorithm to do this, thus we don't show the pseudo code here. The overall pseudo code is given as follows:

```
CONSTRUCTMATRIX(C[1...n], R[1...n]):
  for i ← 1 to n
      for j ← 1 to n
          M[i][j] ← 0
  for i ← 1 to n
      D[1...n] ← DECREASEINDEX(C[1...n])
      j ← 1
      while R[i] > 0
          id ← D[j]
          if C[id] = 0
              report "no such matrix"
          M[i][id] ← 1
          R[i] ← R[i] − 1
          C[id] ← C[id] − 1
          j ← j + 1
  for j ← 1 to n
      if C[j] > 0
          report "no such matrix"
  return M[1..n][1..n]
```

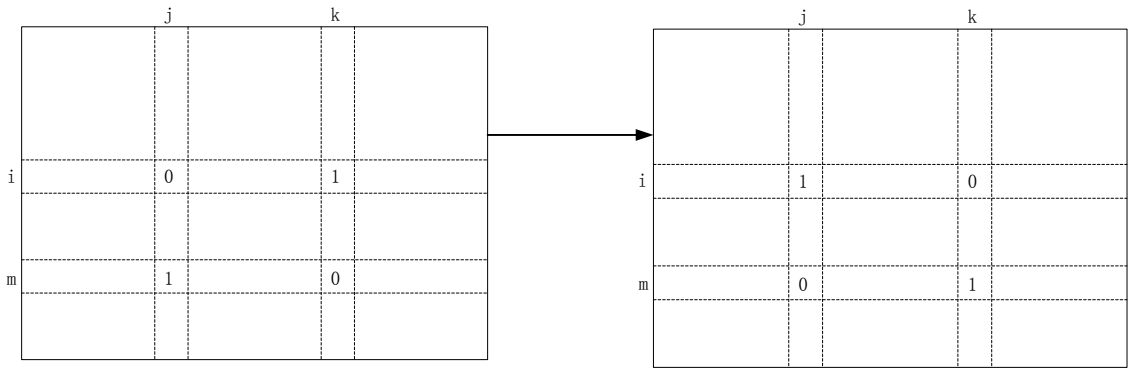

Figure 1. Switch both the pairs of elements in row i and row m at the same time.

Proof: Assume that there is a matrix M that agrees with R and C but different from the matrix M' that is generated by our algorithm. Let row i be the first row in M that is different from M'. Our algorithm sets $M'[i][j]$ to 1, where $C[j]$ has the largest number, while $M[i][j]$ is not filled. Instead, $M[i][k]$ is filled with '1', where $k \neq j$ and $C[j] > C[k]$. In this case, in the sub-matrix $M[i+1..n][1..n]$, the jth column has $C[j]$ 1s and the kth column has $(C[k]-1)$ 1s. Because $C[j] > C[k] > C[k]-1$, there must be some row m in the sub-matrix such that $M[m][j] = 1$ and $M[m][k] = 0$. We can just switch $M[m][j]$ with $M[i][j]$ and $M[m][k]$ with $M[i][k]$. The resulting matrix still agrees with R and C. This is shown in Figure 1. By repeatedly doing this, $M[1..n][1..n]$ can be modified to $M'[1..n][1..n]$ without hurting the argument. □

Running time: There are n rows in the for loop, and the sorting takes $O(nlogn)$ time. Thus, our algorithm runs in $O(n^2 logn)$.

Space: $O(n^2)$

■

2. (•) ***I understand the course policies.***

   Describe an efficient algorithm to assign a ski to each skier, so that the average difference between the height of a skier and her assigned ski is as small as possible.

   Solution: We can use a greedy algorithm to solve it. We sort the two arrays in increasing order. Denote the sorted arrays as $P'[1..n]$ and $S'[1..n]$. Then, we just match each pair of $P'[i]$ and $S'[i]$ for all i.

   Proof: We prove that the greedy choice is correct. Suppose there is an optimal solution that make the same choice as the greedy solution for the first $i-1$ pairs, but differs from the greedy solution at some index i. In other words, $P'[i]$ is not matched to $S'[i]$. Instead, $P'[i]$ is matched to some other skis, say, $S'[j]$, and $S'[i]$ is matched to some other skier $P'[k]$. Since the previous $(i-1)$ pairs are already matched, we have $i < j$ and $i < k$. If we switch to a greedy choice here, i.e., match $P'[i]$ with $S'[i]$ and $P'[k]$ with $S'[j]$. To simplify, let $a = P'[i]$, $b = S'[i]$, $c = P'[k]$ and $d = S'[j]$. The swap changes the expression by:

$$\Delta = \frac{1}{n}(|a-b| + |c-d| - |a-d| - |c-b|)$$

   Note that $a < c$ and $b < d$. We can analyse the cost case by case:

   (a) $a < c < b < d$: Δ evaluates to 0

   (b) $a < b < c < d$: Δ evaluates to $2 \cdot (b-c)/n < 0$

   (c) $a < b < d < c$: Δ evaluates to $2 \cdot (b-d)/n < 0$

   (d) $b < a < c < d$: Δ evaluates to $2 \cdot (a-c)/n < 0$

   (e) $b < a < d < c$: Δ evaluates to $2 \cdot (a-d)/n < 0$

   (f) $b < d < a < c$: Δ evaluates to 0

   But $\Delta < 0$ is actually an improvement over the cost. In conclusion, the greedy choice is at least as good as the choice in optimal solution. □

   Thus, we conclude that the greedy algorithm is correct.

   Running time: Sorting the arrays takes $O(nlogn)$ time. Traverse the arrays takes linear time. Hence the overall running time is $O(nlogn)$.

   Space: We don't need extra space to store things, thus the space is $O(1)$. ■

3. (•) ***I understand the course policies.***

Describe and analyze an algorithm that computes the largest possible number of guests Alice can invite, given a list of n people and the list of pairs who know each other.

Solution: Let the set of all guests be S. Define LARGESTGUESTS(S) as the largest possible number of guests that Alice can invite, we can have a recursive algorithm as follows:

```
LargestGuests(S):
  x ← a guest that does not satify the constraints.
  if no such x
      return |S|
  else
      return LargestGuests(S \ {x})
```

We prove by induction. Let S^i be the remaining set at i-th iteration. In the base case, nobody is removed from the guest list and the algorithm correctly returns size of S. Suppose at the i-th iteration of the algorithm, the set S^i contains the set of optimal solution. If there is some x that cannot satisfy the contraints, it must not be contained in the optimal solution. Hence, we can safely remove it and the resulting set S^{i+1} still contains the optimal solution. Finally, the algorithm must stop because S is finite. When the algorithm stops at k-th iteration, by induction S^k still contains the optimal solution. Hence, the algorithm is correct.

The know and unknown relationship can be obtained from input pairs and stored in a 2D array $K[1 .. n][1 .. n]$. $K[i][j]$ is *true* if i knows j, otherwise *false*. Also, let $Y[i]$ and $N[i]$ denote the number of guests that i knows and doesn't know for the current guest list. An iterative version of algorithm is given as follows:

```
LargestGuests(S):
  largest ← n                    ⟨⟨largest number of guests⟩⟩
  for i ← 1 to n
      present[i] ← true
  stop ← false
  while largest > 0 and stop = false
      x ← 0
      for i ← 1 to n             ⟨⟨find an unsatisfied guest⟩⟩
          if present[i] = true and (Y[i] < 5 or N[i] < 5)
              x ← i
      if x = 0                   ⟨⟨all guests satisfy the constraints⟩⟩
          stop ← true
      else                       ⟨⟨remove this guest⟩⟩
          present[i] ← false
          largest ← largest - 1
          for i ← 1 to n
              if K[x][i] = true  ⟨⟨i knows x⟩⟩
                  Y[i] ← Y[i] − 1
              else               ⟨⟨i doesn't know x⟩⟩
                  N[i] ← N[i] − 1
  return largest
```

Clearly, the while block loops at most n times. There is an $O(n)$ for loop inside. Hence, this algorithm runs in $O(n^2)$. It uses $O(n^2)$ to store the know and unknown information. ■

4. (•) ***I understand the course policies.***

(a) Prove that this heuristic returns a vertex cover of G.

Proof: Suppose the set S obtained by the heuristic is not a vertex cover of G. Then, there must be some edge e that is not incident to any vertex in the set. In other words, e is incident to two leaf nodes a and b in the spanning tree, i.e., $e = (a, b)$. But this is impossible because a and b is leaf nodes. By contradiction, S is a vertex cover of G. □

(b) Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G.

Proof: Consider a depth-first spanning tree T of G. Let the number of nodes in G be n, C represent the set of non-leaf nodes in T which is what the heuristic returned, and M represent a maximum matching of T. In this proof, we show that for any M of T, we can transform M into a maximum matching with a special property which would be explained later, and the maximum matching with the special property helps to prove the heuristic returns a 2-approximation to the minimum vertex cover of G.

First, $|C| = n - |leaves|$, where $leaves$ represents the set of leaf nodes in T. Consider any node $v \in C$. Let $child(v)$ represent the set of child nodes of v in T, and $parent(v)$ represent the parent node of v. We show that $|leaves| = 1 + \sum_{v \in C}(|child(v)| - 1)$. If T has only one node, then $|leaves| = 1+(0-1) = 0$. If the depth of T is 1, then $|leaves| = 1+(|child(root)|-1) = |child(root)|$. If the depth of T is larger than 1, we start from root downwards to the bottommost level of the tree. First, the root could introduce $1 + (|child(root)| - 1) = |child(root)|$ leaf nodes. Then for other nodes $v \in T$ whose $|child(v)| \neq 0$, v could introduce $|child(v)|$ leaf nodes but it makes itself (node v) becomes a non-leaf node (v is computed as a leaf node when we computed $parent(v)$ in the upper level), so we have to exclude v, and then v could introduce $(|child(v)| - 1)$ leaf nodes. So for a spanning tree T, $|leaves| = 1 + \sum_{v \in T}(|child(v)| - 1)$ where $|child(v)| \neq 0$. Thus, $|leaves| = 1 + \sum_{v \in C}(|child(v)| - 1)$.

Consider a maximum matching M of a T. We define a node v as a FREE node when v doesn't exist in M. Moreover, a FREE node v is called FREE* if $v \neq$ the root of T and $parent(v)$ has at least one other child node, say u, while edge$(parent(v), u)$ exists in M. Now we claim that any T has a maximum matching M with a special property that every FREE node, excluding the root, is FREE*.

Proof: Consider a maximum matching M of T. Suppose M doesn't have the special property which is stated above, and then in the following we show we can transform M into a maximum matching with the special property. Let the lowest FREE node v which is not FREE* exists in the level k of T. Without loss of generality, assume $k > 0$ which means v is not the root of T. We show we can make v become either not FREE or FREE* (which means satisfying the special property), while only creating more FREE nodes that are not FREE* in levels $< k$. Consider any v, we can have the following two cases. Let u represent $parent(v)$.

Case 1: $degree(u) = 1$. Because v is a FREE node, u must exist in M, otherwise $edge(u, v)$ can be added into M which contradicts the fact that M is a maximum matching. Since u exists in M, there is an $edge(parent(u), u) \in M$. We can delete $edge(parent(u), u)$ from M but add $edge(u, v)$ into it, while M is still a maximum matching of T. Because v exists in M after this change, v becomes not FREE.

Case 2: $degree(u) > 1$. Because v is a FREE node that is not FREE*, for any child node w of u, there is no $edge(w, u) \in M$, otherwise v would be a FREE* node. By the same argument of Case 1, u must exist in M (otherwise M is not a maximum matching), so there is an $edge(parent(u), u) \in M$. Similarly, we can delete $edge(parent(u), u)$ from M but add $edge(u, v)$ into it, while M is still a maximum matching of T. After this change, v becomes not FREE and other child nodes of u that were FREE become FREE*.

By applying the above techniques on each FREE node in T from the lowest level k upward towards the root level, we can successfully make all FREE nodes become either FREE* or not FREE. Therefore for any T, we can always have a maximum matching M with the property that every FREE node, excluding the root, is FREE*. □

Now suppose for a T, we have a maximum matching M with the special property that every FREE node excluding the root is FREE*. Let F represent the set of FREE nodes in M, where $|F| = n - 2|M|$.

$$|leaves| = 1 + \sum_{v \in C}(|child(v)| - 1) \geq 1 + \sum_{v \in P}(|child(v)| - 1),$$

where P represent a set of nodes $\in C$ in which every node has at least one FREE child node (So $|P| \subseteq |C|$). Since every FREE node is FREE*, for each node v in P, at least one child node of v is not FREE. So $(|child(v)| - 1) \geq$ the number of the FREE child nodes of v. Thus, $\sum_{v \in P}(|child(v)| - 1) \geq |F| - 1$. Note, since root might in F, the root has to be excluded in the summation of FREE child nodes (because root is obviously not a child node). Then we have,

$$|leaves| = 1 + \sum_{v \in C}(|child(v)| - 1) \geq 1 + \sum_{v \in P}(|child(v)| - 1) \geq 1 + (|F| - 1) = |F| = n - 2|M|,$$

$$|leaves| \geq n - 2|M|,$$

$$|C| = n - |leaves| \leq n - (n - 2|M|) = 2|M|.$$

Let OPT represent the number of nodes in the minimum vertex cover of G. At least one node of every edge in the maximum matching M must be in the minimum vertex cover, so $|M| \leq OPT$. So

$$M \leq OPT \Rightarrow 2M \leq 2 \cdot OPT \Rightarrow |C| \leq 2|M| \leq 2 \cdot OPT \Rightarrow |C| \leq 2 \cdot OPT$$

Therefore, we can prove that the heuristic returns a 2-approximation to the minimum vertex cover of G.

Reference material: http://www.cs.uu.nl/docs/vakken/amc/lecture03-4.ps

(c) Describe an infinite family of graphs for which this heuristic returns a vertex cover of size $2 \cdot OPT$.

Solution: When the graph is a path of positive even length, the heuristic always returns a vertex cover of size $2 \cdot OPT$.

Proof: Suppose the length of the path is $2n$, and there are $2n + 1$ nodes in this path. We label the nodes as $0, 1, \ldots, 2n$. Apparently, the best we can do is to just select the nodes that has an odd number of index, since each node can at most connect to two edges. Hence, the optimal solution contains exactly n nodes and $OPT = n$. However, when we apply a DFS to the path from the vertex of degree 1, we will obtain a spanning tree that is actually the path itself. There is only one leaf node in this tree. Hence, the heuristic will return $2n$ nodes, which is two times as OPT. □

Thus, the infinite family of graphs that contains all the paths of positive even length is the solution we want.

■

5. (•) ***I understand the course policies.***

Describe and analyze an efficient 2-approximation algorithm for this problem.

Solution: We can use the following strategy to solve the problem: for each call, if the clockwise routing length is less than or equal to $\lfloor n/2 \rfloor$, route the call clockwise. Otherwise, route it counterclockwise.

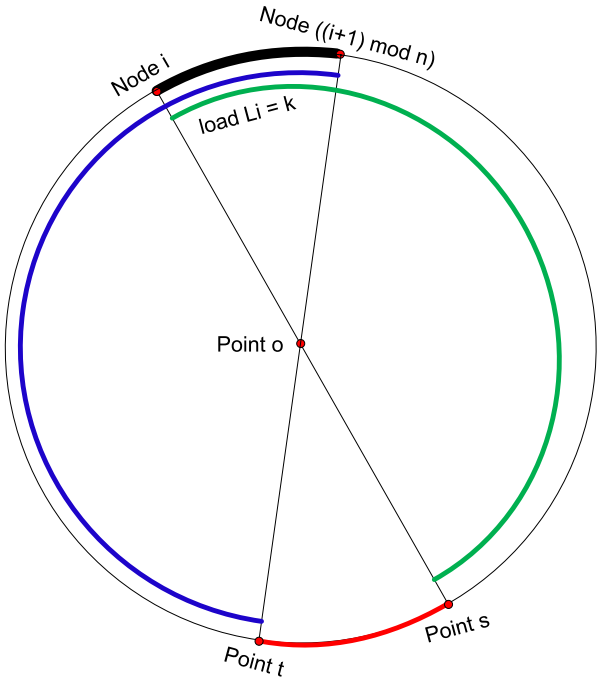

Figure 2. The overlap relationship among the calls covering edge $(i, (i+1) \mod n)$.

Proof: Suppose the overall load computed by our algorithm is k, the edge $(i, (i+1) \mod n)$ has the maximum load $L_i = k$. Define the center of the circle as point o. Connect node i and point o and extend the line to the other side of the circle, we get the intersecting point s. Similarly, we can connect node $((i+1) \mod n)$ and point o and get point t. This is shown in Figure 2. Note that points s and t may not neccessarily be the nodes on the circle.

According to our algorithm, the length of every routed call is less than or equal to half of the perimeter of the circle. So all the routed calls overlapping each other at edge $(i, (i+1) \mod n)$ must be either between point t and node $((i+1) \mod n)$ (blue) or between node i and point s (green). For all of these k calls, they can either be routed clockwise or counterclockwise, which means they either use edge $(i, (i+1) \mod n)$ or not. Assume there are m calls that use L_i and thus there are $k-m$ calls that do not use L_i and cover (s,t) in the optimal solution. Then we have $OPT \geq max(m, k-m) \geq k/2$. So $k \leq 2 \cdot OPT$ and thus our algorithm is 2-approximation.

□

■