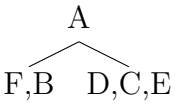
Problem 1

Since the problem only states that we have to find a *legal* binary tree, without stating the tree is full, the result might be ambiguity.

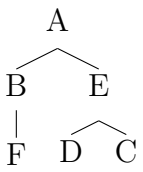
(1)

We can assume the tree is full, then the leftmost element in the pre-order traversal list pre_1 is the root of tree. Since the tree is full, and the array size is more than 1, the value pre_2 must be the left child of root.

After knowing the root of the left child is pre_2, assume $post_i$ equals to pre_2, then all nodes before $post_i$ must be in the left subtree. Taking the example from the problem, we can visualize the tree as



after isolating the root of the tree, the pre-order traversal list lefts $\{pre_2, pre_3, \cdots\}$, aka, {B, F, E, D, C}. We recursively doing the procedure describe above, and it leads to



Describe the above operations into an algorithm,

Algorithm 1: TREE(*pre*, *post*, *index*, *start*, *end*) reconstruct the binary tree using traversal lists.

Input: *pre* is the pre-order traversal list, while *post* is the post-order traversal list. *start* and *end* hold the position of the subarray in *post*, and *index* holds the position in *pre*.

Output: Reconstructed tree.

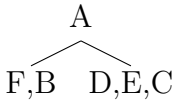
```
1  if index > Size(pre) then
2      return null
   /* First node in pre is the root, assign it.                    */
3  root ← NewNode(pre[index++])
4  if start == end then
5      return root
   /* Search the next element of pre in post.                      */
6  for i ← 1 to end do
7      if pre[index] == post[i] then
8          break the loop
9  if i ≤ high then
10     root.left ← Tree(pre, post, index, start, i)
11     root.right ← Tree(pre, post, index, i + 1, end)
12 return root
```

Reference: `http://goo.gl/DscHcu`

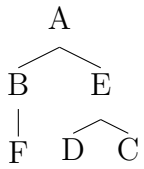
(2)

By definition, we know that the leftmost element in a pre-order traversal list is the root of the tree. Taking pre_1 as an example, if we search pre_1 in the in-order traversal list, we can find out all elements on the left *and* right side of pre_1 (using pre_1 as the pivot).

Using the same example provided by the problem, we now visualize the tree as



We recursively doing the procedure describe above, and it leads to



which is just what we need as well. Now, we can convert the procedure into the algorithm.

Algorithm 2: TREE(*pre*, *in*, *index*, *start*, *end*) reconstruct the binary tree using traversal lists.

Input: *pre* is the pre-order traversal list, while *in* is the in-order traversal list. *start* and *end* hold the position of the subarray in *in*, and *index* holds the position in *pre*.

Output: Reconstructed tree.

```
1  if start > end then
2      return null
   /* First node in pre is the root, assign it.                    */
3  root ← NewNode(pre[index++])
4  if start == end then
5      return root
   /* Search the next element of pre in in.                        */
6  for i ← start to end do
7      if pre[index] == in[i] then
8          break the loop
9  root.left ← Tree(pre, in, index, start, i − 1)
10 root.right ← Tree(pre, in, index, i + 1, end)
11 return root
```

The time complexity is $O(n^2)$, with the worst case occurs when the tree is skewed.

Reference: `http://goo.gl/2eziBC`

(3)

Similar to pre-order traversal list but in a reversed manner, post-order traversal list has the last element as the root, stated it as $post_1$. If in_i is $post_1$, then we can identify the left and right portion of the tree, as the numbers before and after in_i - which is essentially the same procedure as the one in previous question.

Therefore, with some slight modifications,

Algorithm 3: TREE(*in*, *post*, *inStart*, *inEnd*, *postStart*, *postEnd*) reconstruct the binary tree using traversal lists.

Input: *in* is the in-order traversal list, while *post* is the post-order traversal list. *inStart*, *inEnd*, *postStart* and *postEnd* hold the positions of the subarray in both *in*, and *post*.

Output: Reconstructed tree.

```
1  if inStart > inEnd ∨ postStart > postEnd then
2      return null
   /* Last node in post is the root, assign it.                    */
3  root ← NewNode(post[postEnd])
   /* Find the pivot in in.                                        */
4  for i ← inStart to inEnd do
5      if in[i] == root then
6          k ← i
7          break the loop
8  root.left ← Tree(in, post, inStart, k − 1, postStart, postStart + k − inStart − 1)
9  root.right ← Tree(in, post, k + 1, inEnd, postStart + k − inStart, postEnd − 1)
10 return root
```

Reference: `http://goo.gl/ZPbAU9`

Problem 2

(1)

The problem requires us to divide the numbers into two groups, which essentially separates the number pairs that can be eliminated at once. In order to do so, we iterate through all the numbers, and split the one in trials to the other group, whenever their

bit difference is only one.

Algorithm 4: MAGIC(*numbers*) split the numbers that can be eliminated at once into different group.

```
Input: numbers contains all the numbers to be tested.
Output: group1 and group2 are the asked groups.
while SIZE(numbers) > 0 do
    for j ← 1 to Size(numbers) do
        if i != j then
            if BitDifference(numbers[0], numbers[j]) == 1 then
                group1.Add(numbers[0])
                group2.Add(numbers[j])
                numbers.Remove(numbers[0])
                numbers.Remove(numbers[j])
return group1, group2
```

(2)

The Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson method for computing the maximum flow in a flow network in $O(VE^2)$ time. It's used to computes the maximum flow in a flow network, and the idea behind the algorithm is: as long as there is a path from the start node to the end node, with available capacity on all edges in the path, we send *flow* along one of the paths, then we find another path, and so on.

The running time of $O(VE^2)$ is found by showing that each valid path can be found in $O(E)$ time, whenever one of the E edges becomes saturated, the distance from the saturated edge to the start node along the valid path must be longer than last time it was saturated, and that the length is at most V.

(3)

Treat the elimination process as traversal through all the vertices in the graph, and the weight of each edges that connected the vertices, are defined as $3 - m$, where m is the amount of magic power needed to eliminate the connected vertices. The total path length will be the minimum number of magic power required to eliminate all the integers.

The capacity matrix can simply be built upon the groups, aka, the numbers with the same group is destined to have the value in the matrix as $3 - 2 = 1$. In this way, we only have to explicitly test for the value in *the other group*, in order to identify which are the number pairs, those who are paired, have the path weight designated as $3 - 1 = 2$.

Using the Edmonds-Karp algorithm, it will give the path with maximum flow, by subtracting it with $3N$, we get the minimum magic power required.

Problem 3

(1)

Since G is a con-word graph, by definition, $\mu^*(G) = \min_c \mu(c)$, therefore, $\mu(c) >= 0$, meaning that G has no negative cycles.

(2a)

Under the worst case, where the shortest path compose of traversing through all the vertices in order to get from s to v, if all N edges are traversed, than we are getting back to the starting point, forming a loop. Therefore, the shortest path composes of at most $N - 1$ edges.

(2b)

Since graph G is a con-word graph, the shortest path length of it is 0, which creates the lower bound for the inequality. The subtraction $d_k(v)$ denotes the procedure of subtracting the loop on the path. Since the minimum of the entire path (eventually, represents as $d_N(v)$) is non-negative, removing $d_k(v)$ hints us as add back the additional cost generates by the loop, therefore, the overall maximum value shall increase.

(3a)

Since c is a con-weight cycle, it means that the weight on all of its edges are non-negative. Therefore, moving from u to v, require at least 0 (cost) by crossing any edge that are connected between u and v. Hence, we are guaranteed to state that $d(v) = d(u)+w(e)$.

(3b)

In order to have $d_N(v) = d(v)$ to be valid, it means that after walking through all the edges, the distance is equal to the shortest path. Meaning that either there is only a single path to reach v from s, or it's impossible to get there, hence the value is ∞.

In either way, when $d_N(v) = d(v)$ is satisfied, it also means that $d_N(v) - d(v) = 0$, therefore,

$$\max_{0 \le k \le N-1} \frac{d_N(v) - d_k(v)}{N - k} = 0 \tag{1}$$

is trivial.

(3c)

The statement

$$\min_{v \in V} \max_{0 \le k \le N-1} \frac{d_N(v) - d_k(v)}{N - k} = 0 \tag{2}$$

means that, the equality is true for all vertices in V. For a con-word graph, it has $\mu^*(G) = 0$, which cause the minimum result in the fraction

$$\frac{d_n(v) - d_k(v)}{N - k} \tag{3}$$

to be 0, due to the fact that the definition states

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i) \tag{4}$$

for the minimum of μ to be 0, the sum along the path shall be 0 as well, so will be the minimum of the shortest length, when $d_N(v)$ requires one to walk through all the path, and $d_k(v)$ tests for all the possibility, causing $d_N(v) = d_k(v) = 0$ is inevitably to occur.

(4a)

From the problem statement, $p = d_N(v)$. The value can only be reached when there are existence of loops, otherwise we are implying $s = v$, which is useless in this case. In the loops, if all the edges are non-negative values, then subtracting the additional distance generates by the loop will simply become the new shortest path, aka,

$$d_N(v) - d_{N-l}(v) = W \tag{5}$$

However, if there are negative weighted edges exist in the loop, than during the subtraction, more length will get subtracted, causing the actual value to be smaller. So when we remove the additional length growth(or shrink) cause by the loop through $d_{N-l}(v)$, it will get larger then W, leads to the second condition

$$d_N(v) - d_{N-1}(v) > W \tag{6}$$

(4b)

If there exists a con-weight cycle, it means that the path length of that region is 0. By the equation stated in the question, it has to be true for $d_k(v)$, otherwise, $d_N(v) - d_k(v)$ will never reach 0, not to mention the maximum value of all the fractions.

(5a)

From previous statements, we know that we can remove the additional path length by subtracting $d_k(v)$ during the process, so the shortest cycle path shall compose by 1) use up all the edges 2) subtract as much cycles as we can 3) pick out the shortest path - which is what the equation is describing from right to left.

$$\mu^*(G) = \min_{v \in V} \max_{0 \le k \le N-1} \frac{d_N(v) - d_k(v)}{N - k} \tag{7}$$

(5b)

Algorithm 5: MMC(G, c) find the minimum mean cycle of the weighted directed graph G.

Input: A weighted directed graph $G = (V, E)$ and the function of cost c, defined on the edge set E.

Output: The minimum mean cycle of the graph G.

1 **Step 1** Form the linear problem
2 **Step 2** Determine the optimal solution z^* of the problem
3 **Step 3** Determine edges $e \in E$ for which $z^*(e) > 0$.
4 **return** the determined edges.

The problem of finding minimum mean cycle in the weighted directed graph, can be reduced to a linear programming problem, which is also *the problem* stated in the algorithm. The linear programming problem represents the continue model of the minimum mean cycle problem in the directed graph. The mathematical formulation of the problem can be given as:

$$\sum_{e \in E} c(e)w(e) \to \min \tag{8}$$

$$\begin{cases} \displaystyle\sum_{e \in V^+(v)} w(e) - \sum_{e \in V^-(v)} w(e) = 0, \forall v \in V & (9) \\ \displaystyle\sum_{e \in E} w(e) = 1 & (10) \\ w(e) \ge 0 & (11) \end{cases}$$

where $V^+(v)$ is the set of edges, which have their extremities in v, $V^-(v)$ is the set of edges originated in v, and $w(e)$ is a cost associated to each edge $e \in E$.

Reference: Algorithms for finding the minimum cycle mean in the weighted directed graph. D. Lozovanu, C. Petic. 1998. Computer Science Journal of Moldova.