1. You are given a set of \( n \) distinct coloured discs on a stack \( S_1 \). You are also provided with two empty stacks \( S_2 \) and \( S_3 \) whose capacities are big enough to hold all these \( n \) disks. A certain order of these discs (a specific colour sequence) is the password restricting access to some very important resource. If the discs are present in the right order at any time on any of the stacks, then the password restricted resource becomes available.

Attempting to solve this problem (reaching the correct permutation) is like sorting; the difference is that in sorting you know the target order and can compare pairs of elements to guide you to the solution, whereas here the only way to succeed is to ask if the entire sequence is correct or wrong. There is no means of figuring out if two elements are in correct relative order; you are only told if the entire order is correct or wrong.

Give an algorithm (using only push and pop operations of course) to move the discs around the three stacks such that each permutation of the discs appears systematically on one of the stacks till the correct permutation is reached. Analyse the worst case running time of your algorithm (this worst case running time cannot be better than \( n! \), since that is the number of permutations).

2. Given a binary search tree as input:
   (a) Write an algorithm which computes the height of each node.
   (b) For each internal node (non-leaf) compute its imbalance factor. This is the difference of the heights of its two children. If it has only one child, then its imbalance factor is defined to be the height of its only child +1.
   (c) Make the Binary Search Tree approximately balanced by performing suitable rotations (left/right) at appropriate nodes until the imbalance factor at each node becomes either 0 or 1.

   Analyse the running time of your algorithms.

   [NOTE: By definition, the height of a node is the number of edges on a longest downward path to a leaf. Thus the height of a leaf node is ZERO]

3. We know that the breath first search (BFS) algorithm on an undirected graph \( G \) run with source vertex \( s \) gives us the number of edges in a shortest path from \( s \) to every other vertex \( v \), and the results are stored in an array \( d \) with an entry for each vertex.

   ONE INSTANCE of a shortest path is also available from \( s \) to each vertex and this instance is obtained by using the array of parent pointers \( \pi \) with an entry for each vertex in the graph. Using this data, design an algorithm for finding ALL shortest paths from \( s \) to any one fixed vertex \( v \) in the graph. Analyse the running time of your algorithm.

   [HINT: You need to consider VERTICAL CROSS EDGES]
4. Consider the representation of a graph by the following type of matrix $M$. The rows correspond to the cycles in the graph. The columns correspond to the edges. The last row is used for marking edges NOT present on any cycle. An entry of $M[i,j] = 0$ indicates that the cycle $C_i$ does not contain the edge $e_j$. If edge $e_j$ lies on cycle $C_i$ then the entry $M[i,j]$ has the actual weight of edge $e_j$ in the weighted undirected graph. For edges not lying on any cycle their weight appears only on the entry corresponding to the last row and their respective column in the matrix.

The example graph shown below has four cycles (three of length 3 each and one of length 4), and 9 edges. Thus the corresponding matrix has 5 rows and 9 columns as shown. The three cycles of length 3 are labelled $C_1, C_2, C_3$ respectively and the unique cycle of length 4 is labelled $C_4$. The row in the matrix corresponding to cut-edges/ bridges/ edges not on any cycle is labelled $B$.

Assuming the original graph $G$ is connected, design an algorithm for finding a minimum weight spanning tree given this cycle-edge matrix as input. You need to specify your output as the list of edges which get selected for the minimum weight spanning tree. Analyse running time of your algorithm assuming the graph has $|C|$ cycles and $|E|$ edges and express the running time as a function of $|C|$ and $|E|$. Thus the matrix is a $|C| + 1$ by $|E|$ matrix.

![Graph Diagram]

<table>
<thead>
<tr>
<th>Table 1: $Q4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1$</td>
</tr>
<tr>
<td>$C_1$</td>
</tr>
<tr>
<td>$C_2$</td>
</tr>
<tr>
<td>$C_3$</td>
</tr>
<tr>
<td>$C_4$</td>
</tr>
<tr>
<td>$B$</td>
</tr>
</tbody>
</table>
5. Consider the following algorithm.

```plaintext
Graph Search(G, s)
1 for each v ∈ V(G)
2 \( d(v) \leftarrow -\infty \)
3 \( \pi(v) \leftarrow nil \)
4 \( colour[v] \leftarrow white \)
5 \( d(s) \leftarrow 0 \)
6 BUILD MAX-HEAP(\( H \)) on \( V(G) \) keyed on \( d \) values
7 while \( (H \neq \emptyset) \)
8 \( u \leftarrow extract-max(\( H \)) \)
9 for each \( v \in Adj[u] \)
10 \( \text{if } (colour[v] = white \text{ and } d[u] + w(u, v) > d[v]) \)
11 \( \text{then } d[v] \leftarrow d[u] + w(u, v) \)
12 \( \pi[v] \leftarrow u \)
13 \( colour[u] \leftarrow black \)
```

This algorithm takes as input a positive weighted, undirected graph along with a specified source vertex. It is assumed that non-edges of the graph are represented by a \(-\infty\) entry in the adjacency matrix.

(a) Demonstrate the working of the algorithm on the graph \( G \) represented by the following adjacency matrix, using vertex \( A \) as the source vertex.

(b) What does the output of the algorithm represent?

(c) Analyse the running time of the algorithm

---

Table 2: Q5

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>8</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>0</td>
<td>6</td>
<td>−∞</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>C</td>
<td>−∞</td>
<td>6</td>
<td>0</td>
<td>15</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>−∞</td>
<td>−∞</td>
<td>15</td>
<td>0</td>
<td>19</td>
<td>−∞</td>
</tr>
<tr>
<td>E</td>
<td>−∞</td>
<td>4</td>
<td>10</td>
<td>19</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>12</td>
<td>5</td>
<td>−∞</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>