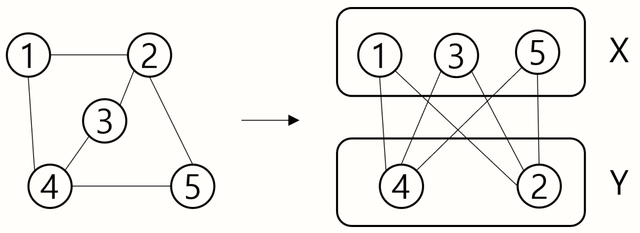
For each of the problems below, transform the input into a graph and apply a standard graph algorithm that you've seen in class. Whenever you use a standard graph algorithm, you ***must*** provide the following information. (I recommend actually using a bulleted list.)

- What are the vertices?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What problem do you need to solve on this graph?
- What standard algorithm are you using to solve that problem?
- What is the running time of your entire algorithm, *including* the time to build the graph, *as a function of the original input parameters*?

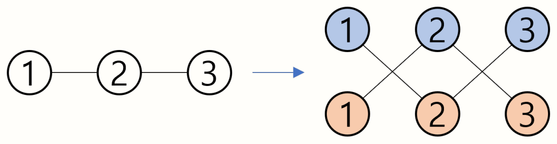
---

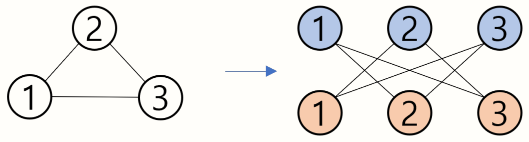
1. You are planning the seating arrangement for a wedding given a list of guests, V . For each guest g you have a list of all other guests who are on bad terms with them. Feelings are reciprocal: if h is on bad terms with g, then g is on bad terms with h. Your goal is to arrange the seating such that no pair of guests sitting at the same table are on bad terms with each other. There will be only two tables at the wedding. Give an efficient algorithm to find an acceptable seating arrangement if one exists.

Solution: Let $E = \{(h, g) \mid h, g \in V, h \text{ is on bad terms with } g\}$ be the set of unordered pairs that represents the feelings of the guests. Let $G = (V, E)$ be an undirected graph with V as the vertices, and E as the edges. The algorithm must return two sets $X, Y \subseteq V$ such that $V \setminus X = Y$ and $\forall (h, g) \in E$, $h \in X$ and $g \in Y$ (or equivalently $g \in Y$ and $h \in X$, since the edges are undirected). That is, it must divide V into two parts so that edges connects one vertex from one part and on vertex from the other part.



There are multiple ways to solve the problem, one would be running DFS on a modified graph. We define a modified graph $G' = (V', E')$ where $V' = V \times \{red, blue\}$ and $E' = \{((v_1, w_1), (v_2, w_2)) \mid (v_1, v_2) \in E,\ w_1, w_2 \in \{red, blue\},\ w_1 \neq w_2\}$, as described in the figure below.





We choose an arbitrary node $u \in V$ such that both (u, red) and $(u, blue)$ are not visited, and run DFS on G' starting from (u, red) until for every $v \in V$, at least one of (v, red) and $(v, blue)$ is visited. Then, we check if there is a node u such that both (u, red) and $(u, blue)$ were visited. If there exists such u, then there exists no acceptable seating arrangement. Otherwise, we return $X = \{u \in V \mid (u, red) \text{ is visited}\}$ and $Y = \{u \in V \mid (u, blue) \text{ is visited}\}$. Since $|V'| = 2|V|$ and $|E'| = 2|E|$, the runtime of the algorithm is $O(V + E)$.

■

2. Plum blossom poles are a Kung Fu training technique, consisting of n large posts partially sunk into the ground, with each pole pi at position (xi, yi). Students practice martial arts techniques by stepping from the top of one pole to the top of another pole. In order to keep balance, each step must be more than d meters but less than 2d meters. Give an efficient algorithm to find a safe path from pole ps to pt if it exists.

Solution: We will have an input of list of n xy-coordinates, the value for minimum distance d, source coordinate and destination coordinate. We will use this data to build a graph by calculating the Euclidean distance between every pair of coordinates and adding an undirected edge between pairs where the distance lies in the range of d to 2d. This algorithm will take $O(n^2)$. We will use a BlackBox algorithm $BFSPath$ that takes a Graph, source and destination vertices and returns True if a path exists between the two points and False if path does not exist. The runtime complexity of running the BFS algorithm will be $O(V+E)$ where V is the number of vertices and E is the number of edges.

- Each vertex is (x_i, y_i) representing the xy coordinates of the Plum blossom poles
- An edge between 2 vertices indicates that the distance between the vertices is between d and $2d$. They are undirected.
- Since we are determining whether a path exists, we do not need to have a value associated with the edge.
- The problem we are trying to solve is whether a path lies between two points in the graph. Whether we can start at a given vertex and traverse through the graph and reach the destination vertex.
- We can use a DFS or BFS algorithm to check whether a path exists between two vertices.

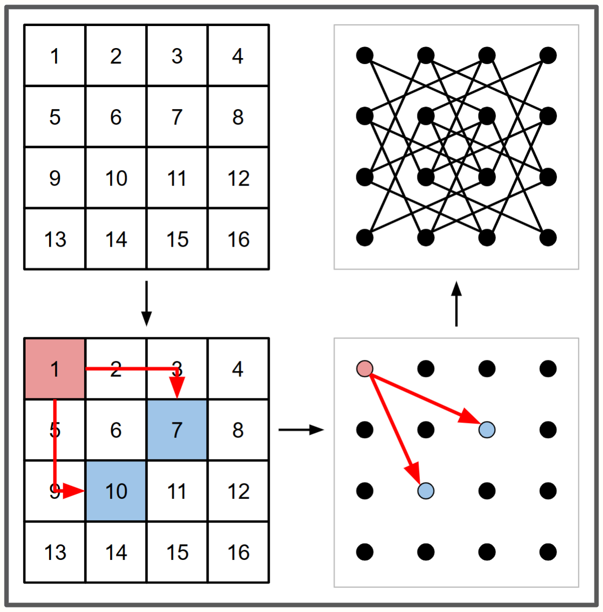
```
BuildGraph($A[(x_1, y_1), (x_2, y_2), ...(x_n, y_n)], d$):
  Let $g \leftarrow$ Empty Graph with n nodes
  for $i \leftarrow 1$ to $n-1$
      for $j \leftarrow i+1$ to $n$
          dist = $SquareRoot((x_i - x_j)^2 + (y_i - y_j)^2)$
          if $d <= dist <= 2d$
              add an Undirected edge between node i and j


  return $g$
```

```
PlumBlossomPath($A[(x_1, y_1), (x_2, y_2), ...(x_n, y_n)], d, src, dest$):
  graph = $BuildGraph(A, d)$
  $PathExists = BFSPath(graph, src, dest)$
  return $PathExists$
```

■

3. The knight's tour is a classic problem that asks given a $n \times n$ board, and a knight with a particular starting position (x, y), is there a sequence of moves that the knight can make so that it visits every square exactly once. Divise an algorithm that finds a knight's tour (if there is one). Don't worry about your algorithm being efficient.

Solution: First we construct the graph representation of the moves a knight can have on any given square. Each square becomes a vertex and we connect 2 vertices together if the knight can move between the corresponding squares on the board.



Now we need to generate an algorithm to determine if there is a Hamiltonian path starting from a given vertex. To do this we use a modified DFS algorithm where when the first depth terminates (i.e. runs out of vertices to visit) we check to see how many edges were used. If the number is not $n^2 - 1$, then we go back one vertex and rerun DFS from a different neighbor. If none of these neighbors yield a $n^2 - 1$ edges then we go back one more vertex. Repeat until all iterations are checked or until the criteria is satisfied. This is effectively checking all the different permutations that could come from the DFS algorithm to see if one is a path.

```
CheckPath($G, v$):
    $N \leftarrow$ neighbors($v$)
    $M \leftarrow$ size($G$)
    $a \leftarrow 0$
    if ($N$ is not empty)
        for $y$ in $N$
            if ($a \neq M - 1$)
                $b \leftarrow 1 +$CheckPath($G - v, y$)
                $a \leftarrow \max\{a, b\}$
    return $a$
```

Where G is a graph and v is a vertex in that graph. We would then run this algorithm on the graph we generated earlier for some given vertex.

There are at most 8 moves a knight can do from a square. This means that constructing the graph will have at most 8 edges for each of the n^2 vertices. So the graph construction takes $\boldsymbol{O(n^2)}$ ***time***. Because there are at most 8 moves, there are at most 8^{n^2-1} permutations for the DFS. The algorithm has constant work done during each iteration which means the algorithm takes $\boldsymbol{O(8^{n^2})}$ ***time***. ■