1. Consider the following code:

```c
int max (int a, int b, int c) { return (a<b)?(b<c)?c:b:(a<c)?c:a; }
int ncities = 0;
int *visited = new int[100];
int **links = new int*[100];

int mystery_func(int current, int parent) {
    visited[current] = 1;
    for (int i=0 ; i <= ncities ; i++) {
        if (!links[current][i] || current == i) continue;
        if (!visited[i]) {
            if (mystery_func(i, current)) return 1;
        } else if (i != parent) return 1;
    }
    return 0;
}

int main(int argc, char **argv) {
    int city1, city2;

    for (int i=0 ; i < 100 ; i++) {
        visited[i] = 0;
        links[i] = new int[100];
        for (int j=0 ; j < 100 ; j++) links[i][j] = 0;
    }

    ifstream fin(argv[1], ios::in);
    while (fin >> city1 >> city2) {
        links[city1][city2] = 1;
        ncities = max(ncities, city1, city2);
    }

    cout << mystery_func(0,-1) << "\n";
    return 1;
}
```

(a) Suppose the input is a list of pairs of numbers where all numbers are separated by blanks. Describe, on the next page, the meaning of `links` (you can say `links[i][j] = X` if and only if `Y` where you decide what `X` and `Y` are).

(b) On the next page state what function `mystery_func` does (your answer should be “`mystery_func` returns `X` if and only if `Y`” where you decide what `X` and `Y` are).

(c) What is printed if the code is run on an input file whose contents is: 1 2 0 4 0 1 1 5 5 6 2 3 5 7 (put answer on next page)?
2. Recall the problem of finding the path from origin to destination that passes through the least number of cities in a given network of cities. A queue was used to simulate simultaneity. A class `City` was defined as follows:

```cpp
class City {
    Queue *q; // Neighbors
    City *from; // To trace a path back to origin
    string name; // Name of the city
    bool mark; // City has been visited if true

public:
    City () { q = new Queue(); from = NULL; mark = false; };

    void setName (string name) { this->name = name; };
    string getName () { return name; };

    void setNeighbor (City* city) { r->enqueue(city); };
    City *getNextNeighbor () { return (City*)q->dequeue(); };

    void setFrom (City* city) { from = city; };

    void setMark () { mark = true; };
    int isMarked () { return mark; };

    void displayBestRoute () {
        cout << name << " ";
        if (from == NULL) cout << "\n";
        else from->displayBestRoute();
    }

    Queue *getQueue () { return q; };

    ~City() { delete q; }
};
```

Consider now the problem of finding the path from origin to destination that passes through the greatest number of cities. Suppose we maintain the top-level queue as before and use it to determine the next city to visit.

(a) What fields and methods in class `City` are no longer needed or need to be replaced?
(b) What new fields should be added to the class and what is their intended function?
(c) Now the `back` field will possibly need to be updated. Describe how this should be done.

Answers can be written on the next page.
Queue events;

```cpp
class Gate {
    protected:
        list<Connect> outfunc; // output connections
        char *name; // name of the logic element
        bool inp[MAX_INPUTS]; // MAX_INPUTS inputs per gate, max
        bool output_value; // output value
        int ninputs; // number of inputs
        int max_inps; // max number of inputs

    public:
        Gate (const char *n);
        char *getName ();
        void setName (char *s);
        virtual void propagate (bool, int);
        void setVal (bool);
        void connectV (Gate*);
    }

class Nor : public Gate {
    public:
        Nor (const char *name) : Gate(name) {
            max_inps = MAX_INPUTS;
            ninputs=0;
            for (int i=0 ; i < max_inps ; i++) inp[i] = false;
            output_value = true;
        }

        void propagate (bool v, int i) {
            if (i >= ninputs || inp[i] == v) return;
            inp[i] = v;
            bool flag = true;
            for (int k=0 ; k < ninputs ; k++) if (inp[k]) { flag = false; break; }
            if (flag != output_value) events.enqueue(new Connect(this, 0, flag));
        }
    };
```

We want to support an additional gate, called XOR, at the primitive level. An XOR gate has no input limit. The output of an XOR gate is true (or 1 if you prefer) if and only if an odd number of its inputs is true. On the next page write the code needed to support XOR.
First dot: A super-magic square $M_n$ is an $n \times n$ matrix in which all integers from 1 to $n^2$ appear in a cell of $M_n$ exactly one time and there is a number $P_n$ such that the sum of all the numbers in any row of $M_n$ is $P_n$, the sum of all the numbers in any column of $M_n$ is $P_n$, the sum of all the numbers on any diagonal of $M_n$ is $P_n$, and the integers from any $n$ cells of $M_n$ that do not occupy a complete row, column, or diagonal do not sum to $P_n$. For example, consider $M_3$:

\[
\begin{array}{ccc}
<p>| &amp; 4 &amp; |
|---|---|---|</p>
<table>
<thead>
<tr>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
</table>
8 | 1 | 6 |
\end{array}
\]

Observe that the sum of the numbers in all the rows, columns, and diagonals is the same. Also observe that all the integers from 1 to 9 are each in exactly one cell of $M_3$. Finally, observe that the sum of the numbers in any three cells that are not a row, column, or diagonal is not equal to the sum of any row.

Because all rows sum to $P_n$, there are $n$ rows, and every integer from 1 to $n^2$ is in $M_n$ exactly once, $n \times P_n = \sum_{1 \leq i \leq n^2} i = (n^2 + 1) \times (n^2)/2$. Hence, $P_n = (n^2 + 1) \times (n/2)$. For example, $P_5 = 26 \times (5/2) = 65$.

Second dot: The following is a description of a two player game called boingo. A round of boingo is played with all cards from the diamond suit of any ordinary deck of playing cards minus the picture cards and the 10 (a total of 9 cards are played). All the cards are initially spread out on the table. Then players alternate in picking up and adding to their hand a single card (one card per turn) from those remaining on the table. The first player whose hand contains three cards that sum to 15 wins. The ace of diamonds has value 1.

Third dot: A fact everyone knows since childhood. It is so obvious that if I state it here the whole problem would be solved trivially without any thought.

Connect (the question): Does there exist a boingo strategy which, if followed by the first player to pick up a card, always results in a win for the first player? Explain, leaving out no details (what is the third dot and how does it connect dots 1 and 2?).