In the $n$-Queens problem, you want to place $n$ queens on an $n \times n$ chessboard, so that no two queens can attack each other (directly horizontal, vertical, or diagonally).

```
__Q__
__Q__
__Q__
Q___
```

**Question 1.** Why is this not a valid solution to the 4-Queens problem?

**Question 2.** How could you fix it?

The following is a generic outline for all backtracking solutions:

```c
void search (int row) {
    if (row==n) printSolution(); //Show the layout
    else {
        for(q[row]=0; q[row] < n; q[row]++)
            search(row+1);
    }
}
```

Note that all backtracking solutions have:

- A function that will be called recursively (with a parameter to keep track of where you are)
- A base case
- A loop to try all possible choices, with a recursive function call inside.

Here is the complete recursive function:

```c
int *q; //positions of the queens
int n; //size of the grid
int **t; //threatened squares
void search (int row) {
    if (row==n) printSolution(); //Show the layout
    else {
        for(q[row]=0; q[row] < n; q[row]++)
            if (t[row][q[row]] == 0) {
                changeThreats(row, q[row], 1);//queen placed here!
                search(row+1);
                changeThreats(row, q[row], -1);//queen removed from here!
            }
    }
}
```
Now we also have to implement changeThreats. Since we are only worrying about placing on the rest of the board, we will only track threats on rows after the current one:

```c
void changeThreats(int row, int column, int change) {
    for (int j = row+1; j < n; j++) {
        t[j][column] += change;
        if (column+(j-row) < n) t[j][column+(j-row)] += change;
        if (column-(j-row) >= 0) t[j][column-(j-row)] += change;
    }
}
```

```c
void printSolution() {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            if (j == q[i]) cout << "Q";
            else cout << "_";
        }
        cout << endl;
    }
}
```

```c
int main(void) {
    cin >> n;
    q = new int[n];
    t = new int* [n];
    for (int i = 0; i < n; i++) {
        t[i] = new int[n];
        for (int j = 0; j < n; j++) t[i][j] = 0;
    }
    search(0);
    delete [] q;
    delete [] t; //one new, one delete!
    return 0;
}
```

Are there any problems with the above code?
PageRank

The key insight that resulted in the dominance of the Google Search Engine was to think of the world wide web as a graph.

- If page A links to page B, this is an endorsement from A of B.
- Not all links are created equal. If A links to B, to determine how good B is requires us to first know how good A is. Sounds circular, but there is a nice solution to the problem.
- Number of links is important. If A and B are equally good pages, but A has one outgoing link to C, and B has 10,000 outgoing links, one of them to D, we would say that the link from A to C confers more support (since it was chosen more carefully).

\[
\begin{align*}
  r_A &= 0.5*r_B \\
  r_B &= r_C \\
  r_C &= r_A + r_D + 0.5*r_E \\
  r_D &= 0.5*r_E \\
  r_E &= 0.5*r_B \\
  r_A + r_B + r_C + r_D + r_E &= 1
\end{align*}
\]

To solve this system of equations requires lots of linear algebra, and the actual PageRank has a few extra optimizations. There is a simpler way to explain the solution without linear algebra:

- You have a web-surfer who starts at a page chosen uniformly at random, and chooses an outgoing link uniformly at random.
- If the surfer gets stuck (no outgoing links) they move to a page chosen uniformly at random. The surfer can get stuck in a less obvious trap (two pages linking only to each other), so there is usually a small probability (around 15%) that at each step the surfer moves to a page chosen uniformly at random.
- We can iteratively calculate the probability the surfer is at any given page at any given step. At the first step, each page is equally likely. We use those values to calculate the probability the surfer is at any given page on the second step, and repeat.
- Typically this process will stop at some small number of iterations (such as 20), but in theory you would want to find the limit as the number of iterations approaches infinity.

This system is called PageRank, and started as a research project by two grad students at Stanford. It was so much better than the competition, that it quickly became a serious product that made them filthy stinkin’ rich.
Trees

A **tree** is an undirected, connected graph, without cycles.

- A tree is **d-ary** if all nodes have between 0 and \( d \) children. A **binary** tree is a 2-ary tree.
- A d-ary tree is **full** if every node has exactly 0 or \( d \) children, and all the leaf nodes are at the same level.
- A d-ary tree is **complete** if you fill each level from left-to-right, and you don’t start a new level until the previous one is complete.

**Question 3.** How should we store the edges of a tree?

Class definition for a binary tree:

```cpp
template <typename T>
class Node {
  T data;
  Node<T> *parent;
  Node<T> *leftChild, *rightChild;
};
```

If we’re storing our nodes in an array, we can simply store the indices of a node rather than pointers.

In a complete binary tree, we don’t even need this much information, since we can directly calculate where the parent, leftchild, and rightchild are based on the current index.

**Binary Search Trees**

A binary tree is a **search tree** if the following property holds for all nodes:

The value of the leftChild is \( \leq \) the value of the parent (or is NULL), which is \( \leq \) the value of the rightChild (unless rightChild is NULL).

**Question 4.** What benefits would a BST have over a sorted array?

**Question 5.** How long does our search algorithm take for a BST?

**Tree Traversal**

- **Pre-order**: Visit parent, then children.
- **In-order**: Visit first child, then parent, then other children
- **Post-order**: Visit children, then parent.