Project 4

Name: Zian Wang

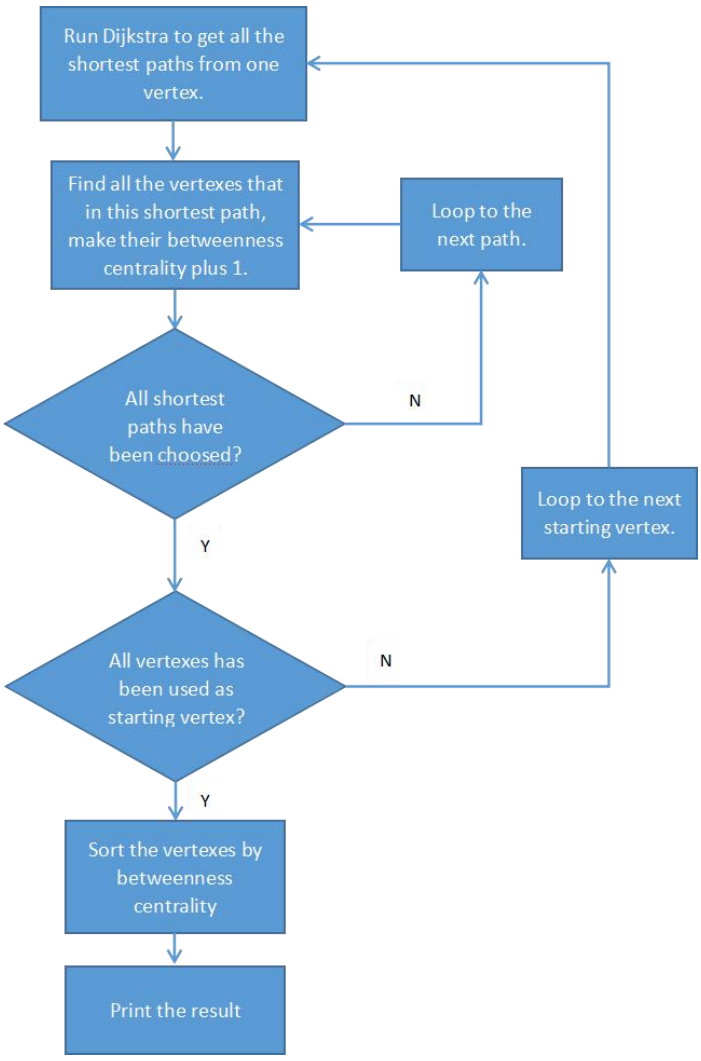
Index:

Problem 1:

(Here are the test cases.)

Program Description:

I implement Dijkstra algorithm which I used in Project 2 in this program to calculate all the shortest paths from each vertex to the other vertices. Then the program extracts all the shortest paths one by one and count how many times each vertex be in these shortest paths, these number of times is the betweenness centrality of the vertices. Then we use Bubble sort to sort the vertices by their betweenness centrality, then print the result. Here is the procedure.



Most parts of this program is the same as the program I use in Project 2, I use a two-dimension array in this program as the adjacency matrix, I add another two-dimension array called betweenness to store the betweenness centrality of each vertex, the first dimension stands for the index of the vertex, the second dimension is the value of betweenness centrality of correspond vertex. Here are the functions.

int[][] readArray(String path, int num):
Input: the path of the input file, the number of the nodes in the graph.
Output: A two dimension array which stores the adjacency matrix.
Function: Read the input file and make the data inside a adjacency matrix, and return it.

Map Dijkstra(int n, int[][] W, int startNode):
Input: the number of the nodes in the graph: n, the adjacency matrix, the index of the start node.
Output: a Map that contains two objects: the set of the edge: F, and the touch array.
Function: apply the Dijkstra algorithm, read the adjacency matrix, return F and touch array.

int[] getPath(int[] touch, int numOfNodes, int startNode, int endNode, int[][] adjacency)
Input: the touch array, the number of the nodes in the graph, the index of start node and end node, the adjacency matrix.
Output: an array that stores the path from start node to end node.
Function: get the path from assigned start node and end node.

int[][] completeAdjacency(int[][] adjacency, int numOfNodes)
Input: the adjacency matrix, the number of the nodes.
Output: the completed adjacency matrix.
Function: the adjacency matrix which is returned from readArray() is not complete. The diagonal elements are not 0, and the weight between two points which are not connected is not infinity. So this function is used to make diagonal elements 0 and the infinite length infinity.

int[][] resetAdjacency(int[][] adjacency, int numOfNodes)
Input: the adjacency matrix, the number of the nodes.
Output: a copy of the adjacency matrix.
Function: in Java, input an array into a function is inputting its location, so when we operate the array in the function, it will be change inside and outside the function. Therefore, if we simply input the same adjacency matrix into the Dijkstra function every time, it will be changed, so every time(except the first time) we input the matrix, actually it has already been changed. So we use this function to create a exactly same matrix to input into the function.

int getNumOfNodes(String path)

Input: the path of the input file.
Output: the number of the nodes in the graph.
Function: to get the number of the nodes.

void run(String filePath, int numOfNodes)
Input: the path of the input file, the number of the nodes.
Output: none.
Function: used to call other functions in this program in a proper order and get the results we want.

Test Cases:

```
"C:\Program Files\Java\jdk-15.0.2\bin\java.exe" "-javaagent:A:\IntelliJ IDEA Community
Here are the top 20 betweenness centrality vertices in the graph in descending order:

NodeId      Betweenness centrality
760           364999
590           321542
859           286812
757           283052
858           281928
1021           265968
374           262332
92           255850
931           255292
857           242166
758           240924
393           238408
496           232837
775           232244
932           231183
1093           230694
1094           229580
835           228979
589           228942
751           228580

Process finished with exit code 0
```

One thing that should be noticed is that in this input file, some tiny groups of vertices are not connected to the main part of the graph. For example, node 1031 and node 1392 are just connected to each other, they are like isolated apart from the other vertices. This will influence that if we want to find the shortest path from the other points to these

points, for example: the shortest path from 0 to 1031, the path will make no sense because actually 1031 is not connected to the main part. In this program, I delete these paths by judge whether a path's distance is extremely big(disconnected vertices' distance is 99999999 in this program), so that the result should be more precise.

Problem 2:

(Here are the test cases.)

Program Description:

I create 6 java classes to solve this problem, 4 of them are the classes that I implement the algorithms in, the rest 2 classes are for priority queue and the node data structure.

Problem2_dynamic_1D.java: uses a one-dimensional array for storing only the elements of the lower triangle in the adjacency matrix, and uses dynamic algorithm.

Problem2_dynamic_2D.java: uses a two-dimensional array as the adjacency matrix and uses dynamic algorithm.

Problem2_Branch_and_Bound_1D.java: uses a one-dimensional array and uses branch-and-bound algorithm.

Problem2_Branch_and_Bound_2D.java: uses a two-dimensional array and uses branch-and-bound algorithm.

The dynamic algorithm is a little bit complex because this algorithm needs us to use sets as index of D and P. I solve this by using binary number, I create arrays of 1 and 0 to represent sets. For example, if there is a graph contains 5 nodes, and I want to represent a set that contains the 1st, 2nd, and the 4th nodes, the array will be [1, 1, 0, 1, 0], then we convert this binary number to the decimal number, which is 26, then we use 26 as the index in D and P.

The bound function in branch-and-bound programs uses the same method described in the text book for the TSP problem.

I use the method that Professor mentioned in the lecture to store the adjacency matrix into the one-dimensional array named A. The function named read1DArray() will do this. When we read the input file, we will see if the NodeID: i is larger than ConnectedNodeID: j, if the NodeID is larger, this means the corresponding distance is the element of the lower triangle, so its index in A should be i* (i - 1)/ 2 + j, and we will store it in the A[i* (i - 1)/ 2 + j]. In other situation, if i<=j, this means this distance is the upper triangle element or the diagonal element, we will not store it because the graph is an undirected graph.

The dynamic algorithm and the Branch-and-Bound algorithm are the same as them in the text book. Here are the functions:

Functions to read input graph:

public static int[] read1DArray(String path, int num):
Inputs: path of the input file, the number of the nodes in the file.
Outputs: an one-dimensional array that only contains the lower triangle elements.
Function: read the input graph and store it in the one-dimensional array.

public static int[][] read2DArray(String path, int num)
Inputs: path of the input file, the number of the nodes in the file.
Outputs: a two-dimensional array that only contains the lower triangle elements.
Function: read the input graph and store it in the two-dimensional array. This function is the same as the function I used in Project2.

Functions in the dynamic algorithm:

public static Map TSP(int[] adjacency, int n)
Inputs: the adjacency array, and the number of the nodes: n.
Output: a hashmap that contains the optimal path and the minimum length.
Function: this is the dynamic algorithm that solves TSP problem, it is the same as it in the text book.

public static int combNumber(int n, int k)
Inputs: int n and k, k is smaller than n.

Outputs: the number of combinations that pick k items from n.
Function: calculate the number of combinations that pick k items from n. Because this function uses factorial to calculate the value, and some factorial is pretty large that exceeds the range of int in java, we use java.math.BigInteger to deal with these extremely large numbers in this function.

public static boolean inSet(int i, int[] set)
Inputs: a vertex i, and a set.
Outputs: whether this vertex is in the set.
Function: judge whether a vertex is in a given set.

public static void moveA(int[] A, int n)
Inputs: a set A, the number of the nodes in set A.
Outputs: none.
Function: this function will change the set A to the next combination that contains the same chosen items, for example, this function will change [1, 0, 0, 1, 1] to [0, 1, 0, 1, 1].

public static int[] Aminusj(int[] A, int j, int n)
Inputs: a set A, a vertex j, the number of vertices in the set A.
Output: a modified set which is created by removing j from A.
Function: a new set that is the same as set A, but vertex j is removed.

public static int convertSet(int[] A, int n)
Inputs: a set A, the number of the nodes: n in the set.
Outputs: a int n that represents the set A.
Function: convert the set A into a int, the function does this by convert the binary number that represents the set into a decimal number.

public static int getLength(int i, int j, int[] adjacency)
Inputs: two int i, j, and the adjacency array.
Outputs: the distance between vertex i and j.
Function: extract the length between i and j from the one-dimensional adjacency array.

public static int getNumOfNodes(String path)
Inputs: the path of the input file.
Outputs: the number of the nodes in the file.
Function: to get the number of the nodes in the input file.

Functions in the branch and bound algorithm:

public static Map TSP (int[] adjacency, int n)
Inputs: the adjacency array, the number of the nodes in the graph.
Outputs: a hashmap that contains the optimal path and the minimum path length.
Function: this is the branch and bound algorithm for TSP problem, this algorithm is the same as it

in the text book.

public static int getLength(int i, int j, int[] adjacency)
Inputs: the nodeID of two nodes, the adjacency array.
Outputs: the distance between them.
Function: extract the distance between two vertices from the one-dimensional adjacency array.

public static boolean inPath(int[] path, int x)
Inputs: a path, a vertex whose nodeID is x.
Outputs: whether the node is in the path.
Function: judge whether a node x is in a given path.

public static double bound(Node node, int[] adjacency, int n)
Inputs: a node, the adjacency array, the number of the nodes in the graph.
Outputs: the bound of this node.
Function: calculate the bound of the node. This function uses the method that can calculate the bounds of the nodes for TSP problem on the text book, I code that method into this function.

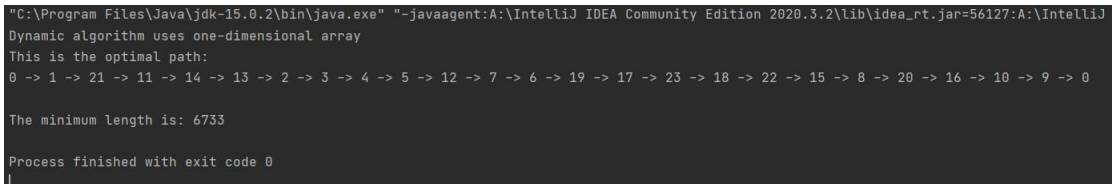
public static void finalize(Node node)
Inputs: a node.
Outputs: a finalized node.
Function: this function is used when we find a candidate solution(the search has come to a leaf), this function will add the last one vertex which is not visited to this node's path, and then add the starting node to this node's path.

public static int length(Node node, int[] adjacency)
Inputs: a node, the adjacency array.
Outputs: the length of this node's path.
Function: calculate the length of the node's path.

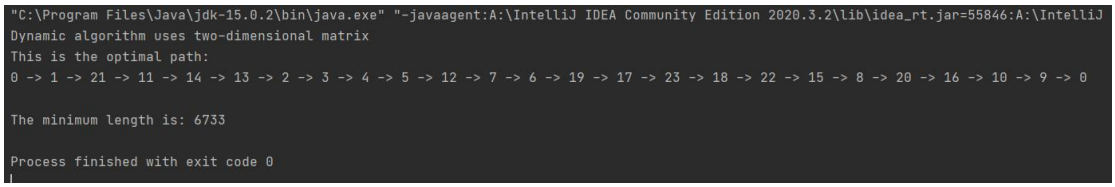
public static int getNumOfNodes(String path)
Inputs: the path of the input file.
Outputs: the number of the nodes in the file.
Function: to get the number of the nodes in the input file.

Test Cases:

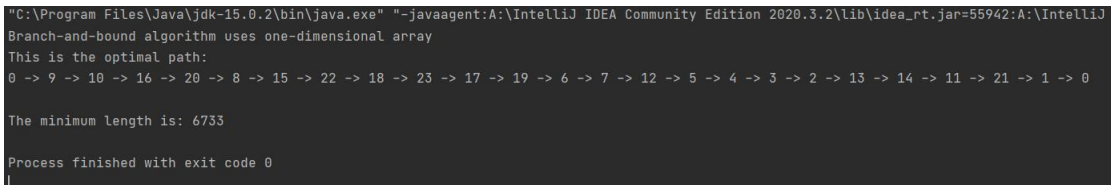
Dynamic algorithm uses one-dimensional array:



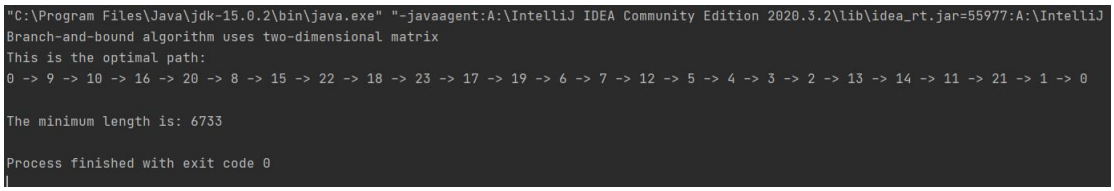
Dynamic algorithm uses two-dimensional matrix:



Branch-and-bound algorithm uses one-dimensional array:



Branch-and-bound algorithm uses two-dimensional matrix:



There is a thing that should be noticed, because the input graph is an undirected graph, the optimal path is also undirected. In other words, the optimal path can be go through in both direction. Therefore, we can see the optimal path output by dynamic algorithm is reversed to the path output by Branch-and-bound algorithm. This is okay because actually these two paths are the same, just in the reversed direction.

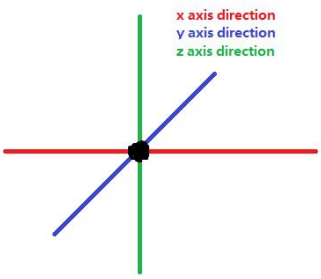
Problem 3:

(Here are the test cases.)

Program Description:

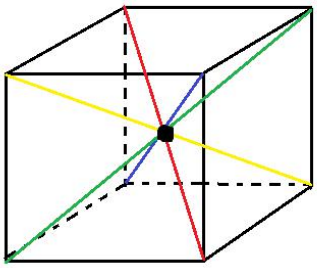
I modify the original backtracking algorithm for 2D n-queens to solve this 3D version n-queens problem. I use depth-first search to travel all the candidate solutions and backtrack. In this program, the coordinates start from 1. The program will set the first queen in the first cube: (1, 1, 1), then judge whether this placement is promising, if not promising, we will place the first queen in the second cube: (1, 1, 2), then see if it is promising again, if the first placement is promising, we will place the second queen in the same order and see if it is promising and loop until the last queen is placed. In this order, the programming is doing a depth-first search, and prune the tree by promising function.

Promising function judges whether 3 rules are been obeyed.
The 1st rule is two queens can not in the same line that paralleled to x or y or z axis, if they are, they can attack each other. The program judge this by judging whether two queens' x and y coordinates are the same, or x and z are the same, or y and z are the same, or x, y, z are all the same. Here is the plot shows this rule.



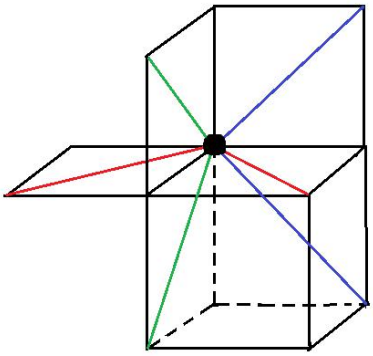
If there is a queen in the black dot, no queens will be allowed in the red, blue and green line.

The 2nd is two queens can not in the same vertical+horizonal diagonal, if they are, they can attack each other. The program judge this by judging whether their 3 distances between x-coordinates, y-coordinates and z-coordinates are the same. Here is the plot shows this rule.



This is a cubic, and the black dot is the center of the cubic. This cubic is not the n*n*n cubic we use in the problem, this cubic is just for explaining. If there is a queen at the black dot, no queens will be allowed in the red, blue, green and yellow line.

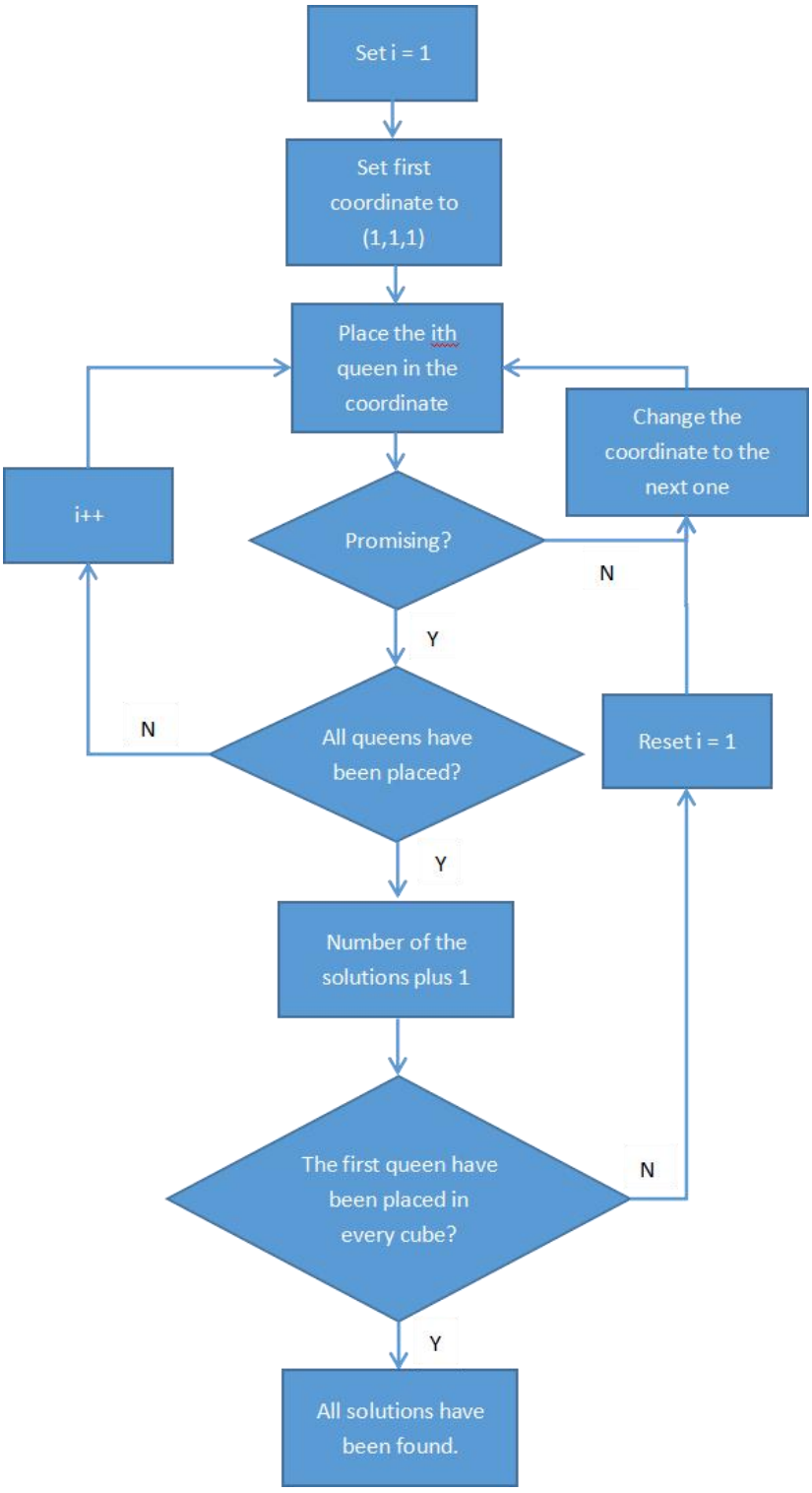
The 3rd rule is two queens can not be in the same diagonal that in the same flat, if they are, they can attack each other. The program judge this by judging whether two queens distances between x and y coordinates are the same and the z-coordinates are the same, or the distances between x and z coordinates are the same and the y-coordinates are the same, or the distance between y and z coordinates are the same and the x-coordinates are the same. Here is the plot shows this rule.



This is a cubic and the black dot is at a corner of the cubic. If there is a queen at the black dot, no queens will be allowed in these 2 red lines, 2 blue lines and 2 green lines. This cubic is also just for explaining, it is not the cubic we use in the problem.

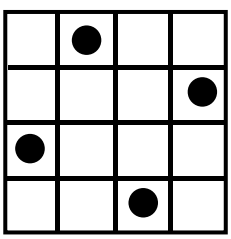
These 3 rules contains all the attack methods that the queen can take. Therefore, if any one of these 3 rules is broken, the node is non-promising, and the promising() function will return false. Otherwise, if all of these 3 rules are obeyed, this node is promising, the promising() function will return true.

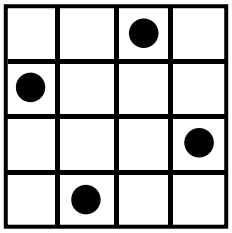
The procedure is in the next page.



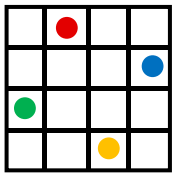
The program will stop when the first queen is placed in all of the cubes, from (1, 1, 1) to (n, n, n). This will make a problem that one solution will be reached multiple times and will be considered multiple times. We can see this from an 2D example. Assume we are solving the normal, or 2D

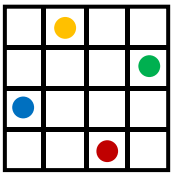
version of the 4-queens problem, we should get 2 solutions shown below:

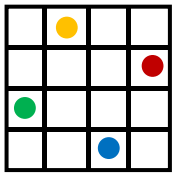


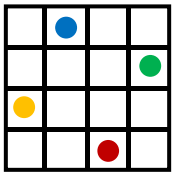


However, the program will consider all of these solution as different solutions:









In these 4 plots, we denote different queens by different colors. These are some examples, actually we have 4! = 24 number of sub-solutions for this single solution. Therefore, we can see actually we are finding a solution, and then change the locations of the queens, then we think we get a “new” solution. Therefore, we have to divide the final number of the solutions by n! to get the correct number. The numbers shows in the screen are the correct ones.

There is other ways to solve this problem, we can place one queen in only some of the cubes rather than try to place them in all the cubes. This is just like the 2D version n-queens problem: we can only place the first queen in the first row, then second queen in the second row, so on

and so force. However, in the 3D version, the situation is much more complex, so it is hard to find which cubes can one queen be placed in to make the solutions not repeat. Another way is that we can store all the solutions, and when we get a new solution, we can judge whether it is the same of one of the solutions we get before, this way also has a problem, which is we have to store a very large amount of solutions. When n=4 we have 27880 non-repeated solutions, so when we judge the last solution we get, we have already stored tens of thousands of solutions, and if n=5, we have to store 8637762 solutions, this will cost so much space, so I do not choose this way either.

I use a two-dimensional array called Queens[][] to store the coordinates of the queens. The first dimension is the index of the queen, the second dimension is the coordinates of the queen. For example, Queens[0][1] stores the y-coordinate of the 1st queen, Queens[0][2] stores the z-coordinates of the 1st queen. Here are the functions in the program.

public Problem3(int n)
Inputs: number of the queens: n.
Outputs: none.
Function: the constructor of the class.

public int abs(int a)
Inputs: int a.
Outputs: the absolute value of a.
Function: calculate the absolute value of the input.

public int factorial(int n)
Inputs: int n.
Outputs: the factorial of n: n!.
Function: calculate n!.

public boolean promising(int i)
Inputs: the index of the queen that needed to be judged: i.
Outputs: whether this placement of the ith queen is promising.
Function: judge whether a placement of ith queen is promising.

public void queens3D(int i)
Inputs: the index of the queen that needed to be placed.
Outputs: none.
Function: the function that uses backtracking algorithm to solve n-queens problem in 3D.

public static void run(int n)
Inputs: number of the queens: n.
Outputs: none.
Function: this function is a wrapper of the other functions, input the number of the queens and this function will call the constructor and the queens3D function and print the n and number of the results.

Test Cases: