N Queen Problem:

The N Queen problem is a problem relating to chess. In chess, the queen is the most powerful piece in the game as it has the ability to move any number of squares vertically, horizontally or diagonally.

N Queen is the problem of placing N chess queens on an N x N chessboard so that no two queens can attack each other. In other words, each queen must be placed in such a way that it does not attack any other queen on the board.

Simulated Annealing Algorithm:

Simulated Annealing Algorithm is a local search maximization algorithm. Main difference from the other local search algorithms is that Simulated Annealing algorithm intentionally allows some bad moves, or downhill movements, to avoid being stuck with local maxima so that a global maximum can be found. Any move made is accepted with a random probability which becomes more likely as temperature approaches 0 (zero).

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a maximization problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                      T, a "temperature" controlling the probability of downward
steps
    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T=0 then return current
        next ← a randomly selected successor of current
        $\Delta E$ ← VALUE[next] – VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

Algorithm Analysis:

Simulated annealing is a metaheuristic and it uses two important random variables as input. These variables are Temperature (t) and cooling factor (c). For our specific problem it also takes an input which is the number of queens (q), and the algorithm terminates when temperature becomes nearly zero (α). Termination of the algorithm highly depend on these variables. For instance, in i^{th} iteration temperature will be equal to:

$$t_i = t_0 * c^i$$

If we assume that the main loop of the algorithm terminates after k iterations, k will be equal to:

$$k = \log_c \frac{\alpha}{t_0}$$

In each iteration, there is an $O(n)$ loop that checks the energy difference of the current and potential next state. Hence, we should add this to our running time:

$$O(\log_c \frac{\alpha}{t_0}) + O(n)$$

Before and after the algorithm runs, program also calculates the total energy of the board. Therefore, we finally have:

$$O(\log_c \frac{\alpha}{t_0}) + O(n) + 2 * O(n^2)$$

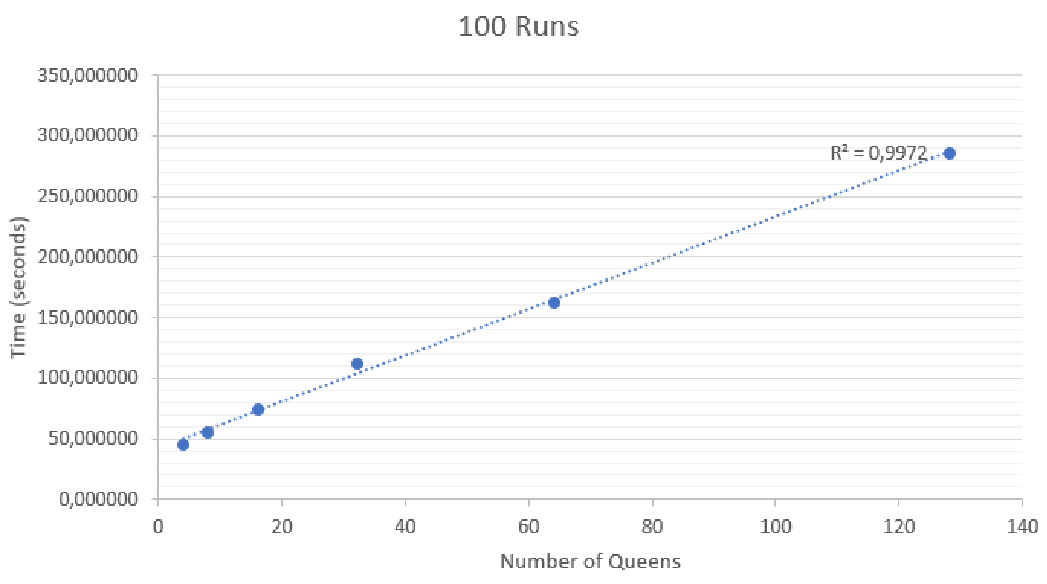
$$= O(\log_c \frac{\alpha}{t_0} + n^2)$$

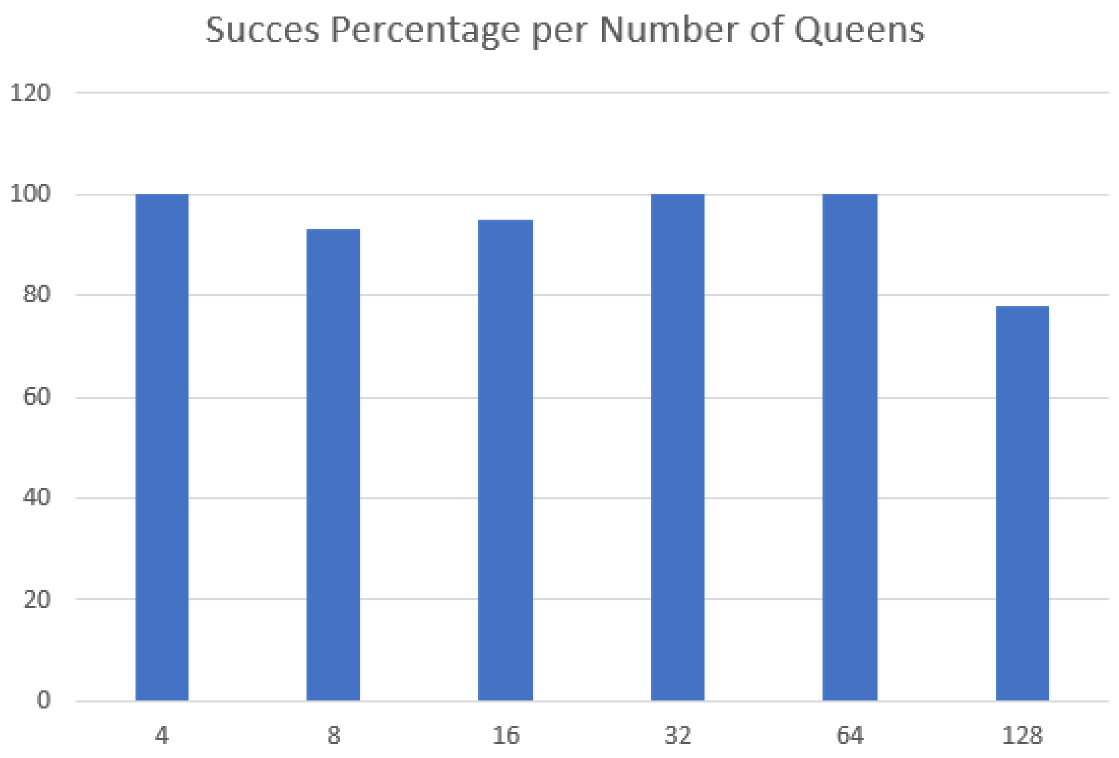
Note that we neglect $O(n)$ since there is a higher order of the same variable.

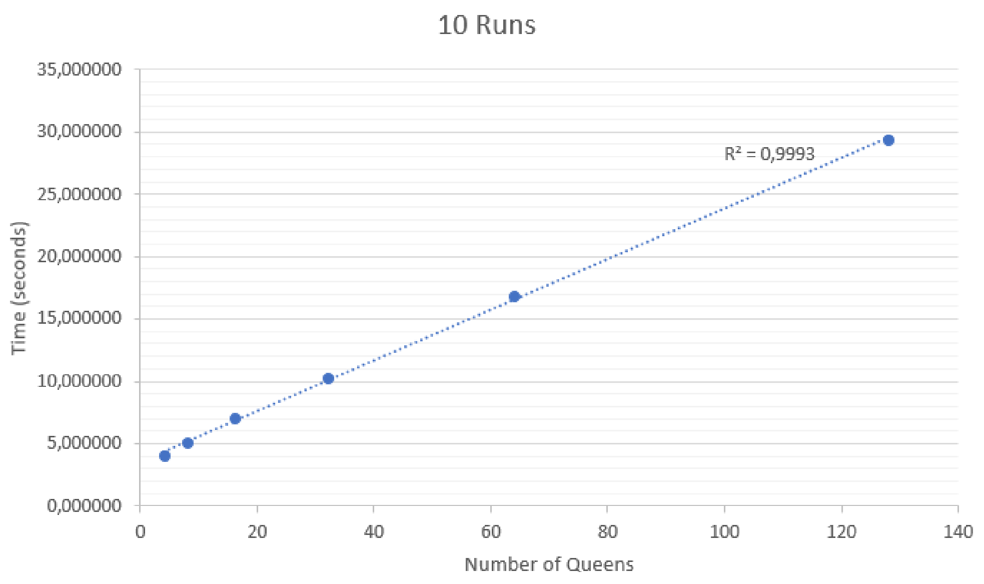
An optimal implementation has the complexity of $O(n^2)$.

Experimental Results:

Size	Time	StandardDeviation	StandardError	%90-CL	%95-CL	Success
4	4,075870	0,005784	0,001829	0,406636 - 0,40853	0,406655 - 0,40852	100
8	5,074150	0,010513	0,003324	0,505689 - 0,50914	0,505722 - 0,50911	100
16	7,103440	0,011991	0,003792	0,708372 - 0,71231	0,70841 - 0,712278	100
32	10,328300	0,026347	0,008332	1,0285 - 1,03716	1,02858 - 1,03708	100
64	16,839400	0,018853	0,005962	1,68084 - 1,68704	1,6809 - 1,68698	100
128	29,462300	0,098668	0,031201	2,93001 - 2,96245	2,93032 - 2,96214	100





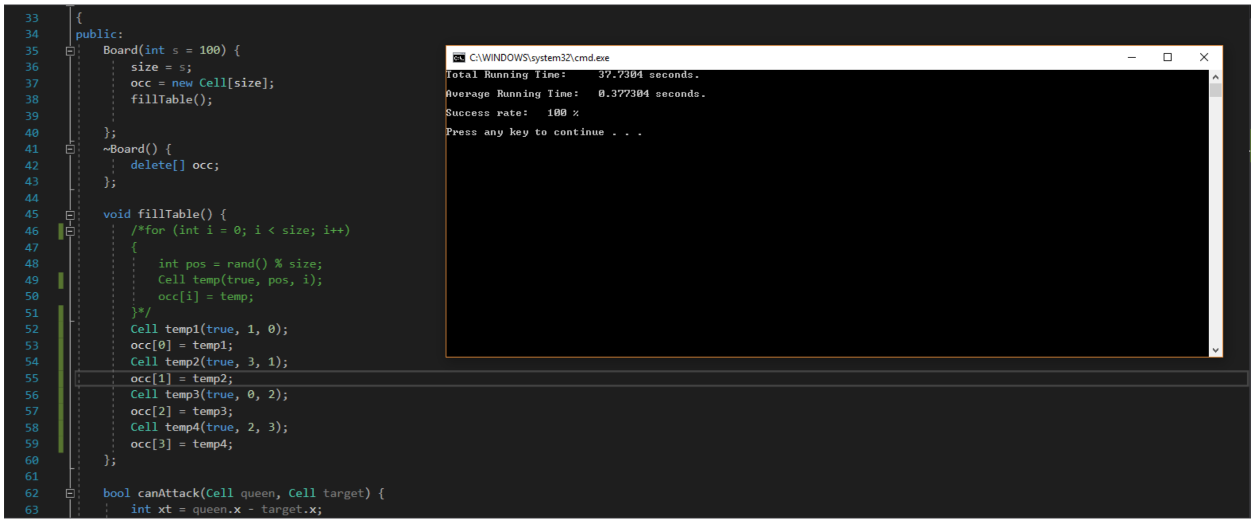


Size	Time	StandardDeviation	StandardError	%90-CL	%95-CL	Success
4	45,549900	0,022415	0,002241	0,454333-0,45664	0,454356-0,456642	100
8	55,874500	0,022472	0,002247	0,557577-0,559913	0,557599-0,559891	93
16	75,310200	0,020734	0,002074	0,752024-0,75418	0,752045-0,75416	95
32	112,456000	0,053938	0,005394	1,12175 - 1,12736	1,12181 - 1,12731	100
64	162,315000	0,066479	0,006648	1,61969 - 1,62661	1,61976 - 1,62654	100
128	285,651000	0,003531	0,000353	2,85468 - 2,85835	2,85471-2,85831	78

Testing:

To test our algorithm, we preferred to use blackbox testing. Normally, the algorithm goes through random states from a randomly generated initial state. Hence it is best to consider some extreme cases that could happen throughout the solution process. There are 4 cases to be tested:

1. Initial state is already the optimal solution.



```cpp
{
public:
    Board(int s = 100) {
        size = s;
        occ = new Cell[size];
        fillTable();

    };
    ~Board() {
        delete[] occ;
    };

    void fillTable() {
        /*for (int i = 0; i < size; i++)
        {
            int pos = rand() % size;
            Cell temp(true, pos, i);
            occ[i] = temp;
        }*/
        Cell temp1(true, 1, 0);
        occ[0] = temp1;
        Cell temp2(true, 3, 1);
        occ[1] = temp2;
        Cell temp3(true, 0, 2);
        occ[2] = temp3;
        Cell temp4(true, 2, 3);
        occ[3] = temp4;
    };

    bool canAttack(Cell queen, Cell target) {
        int xt = queen.x - target.x;
```

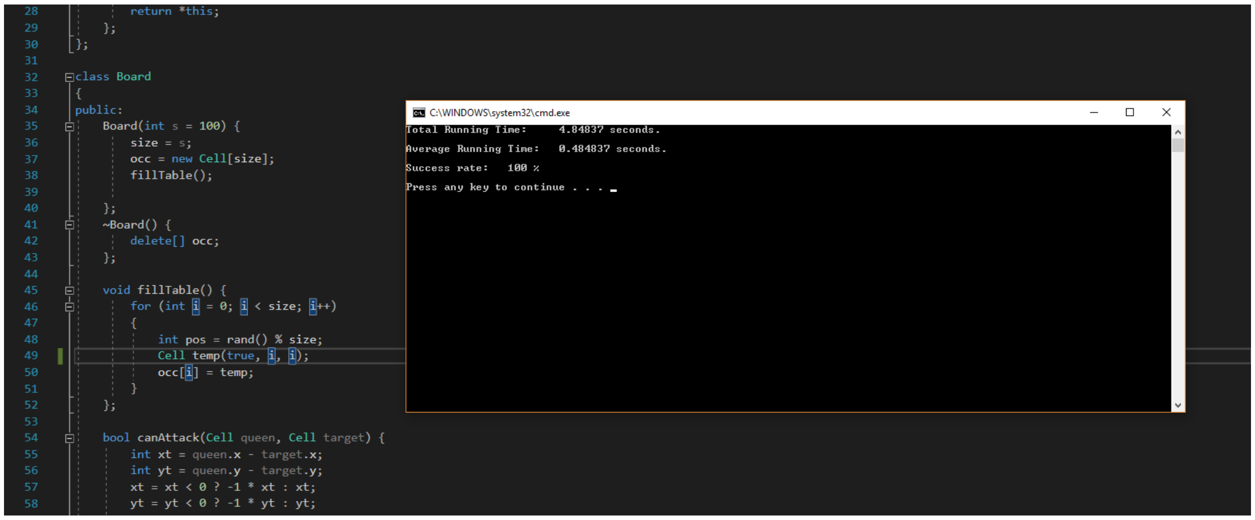
```
Total Running Time:     37.7304 seconds.

Average Running Time:   0.377304 seconds.

Success rate:   100 %

Press any key to continue . . .
```

2. All queens are initially located on a same diagonal of the matrix.



```cpp
            return *this;
    };
};

class Board
{
public:
    Board(int s = 100) {
        size = s;
        occ = new Cell[size];
        fillTable();

    };
    ~Board() {
        delete[] occ;
    };

    void fillTable() {
        for (int i = 0; i < size; i++)
        {
            int pos = rand() % size;
            Cell temp(true, i, i);
            occ[i] = temp;
        }
    };

    bool canAttack(Cell queen, Cell target) {
        int xt = queen.x - target.x;
        int yt = queen.y - target.y;
        xt = xt < 0 ? -1 * xt : xt;
        yt = yt < 0 ? -1 * yt : yt;
```

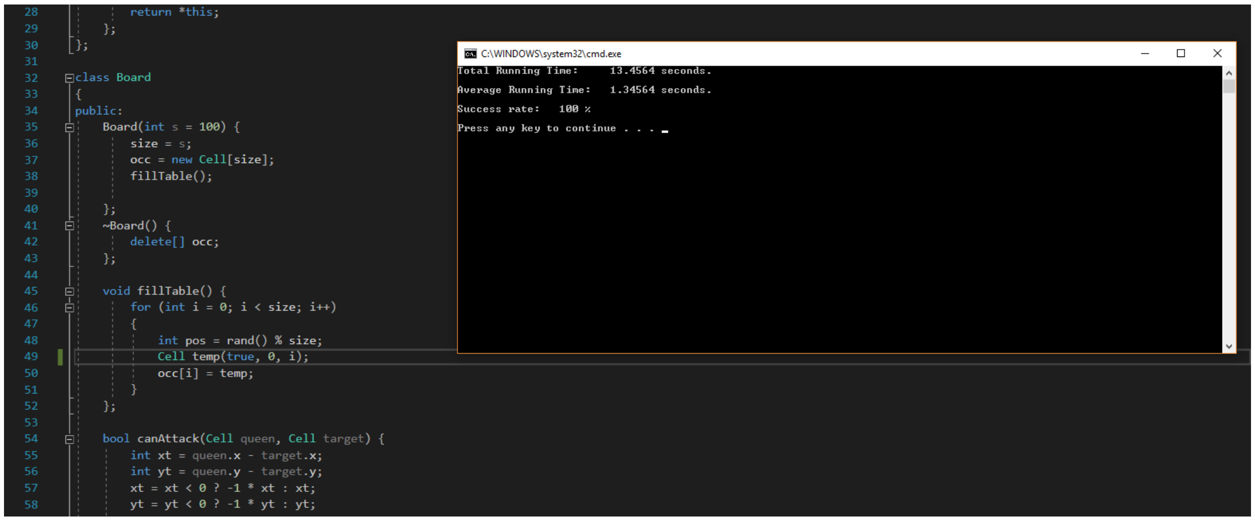
```
Total Running Time:     4.84837 seconds.

Average Running Time:   0.484837 seconds.

Success rate:   100 %

Press any key to continue . . . _
```

3. All queens are initially located on a same row.



```
            return *this;
        };
};

class Board
{
public:
        Board(int s = 100) {
            size = s;
            occ = new Cell[size];
            fillTable();

        };
        ~Board() {
            delete[] occ;
        };

        void fillTable() {
            for (int i = 0; i < size; i++)
            {
                int pos = rand() % size;
                Cell temp(true, 0, i);
                occ[i] = temp;
            }
        };

        bool canAttack(Cell queen, Cell target) {
            int xt = queen.x - target.x;
            int yt = queen.y - target.y;
            xt = xt < 0 ? -1 * xt : xt;
            yt = yt < 0 ? -1 * yt : yt;
```

C:\WINDOWS\system32\cmd.exe

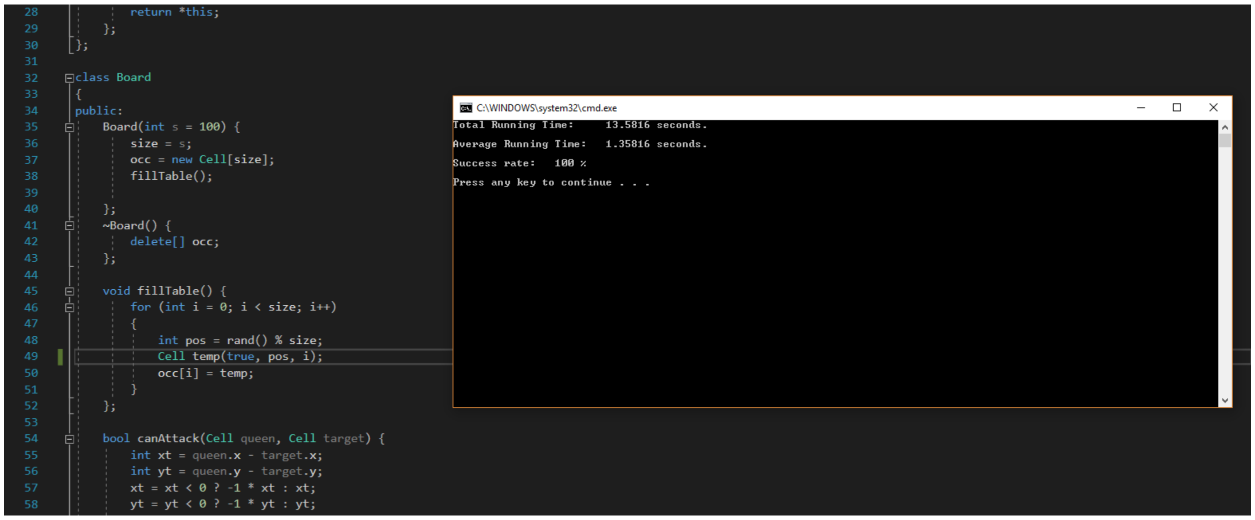
```
Total Running Time:     13.4564 seconds.

Average Running Time:   1.34564 seconds.

Success rate:   100 %

Press any key to continue . . .
```

4. All queens are initially randomly located.



```
            return *this;
        };
};

class Board
{
public:
        Board(int s = 100) {
            size = s;
            occ = new Cell[size];
            fillTable();

        };
        ~Board() {
            delete[] occ;
        };

        void fillTable() {
            for (int i = 0; i < size; i++)
            {
                int pos = rand() % size;
                Cell temp(true, pos, i);
                occ[i] = temp;
            }
        };

        bool canAttack(Cell queen, Cell target) {
            int xt = queen.x - target.x;
            int yt = queen.y - target.y;
            xt = xt < 0 ? -1 * xt : xt;
            yt = yt < 0 ? -1 * yt : yt;
```

C:\WINDOWS\system32\cmd.exe

```
Total Running Time:     13.5816 seconds.

Average Running Time:   1.35816 seconds.

Success rate:   100 %

Press any key to continue . . .
```

<u>Conclusions:</u>

In the main program we are dealing with the temperature and the effect on whether it reduces or not. What basically happens is that this algorithm picks up a tweaked solution and computes in energy. If the energy of the solution is lower than the energy of the current solution, it replaces the current solution with the tweaked solution. But if the energy is higher it calculates the difference between current and tweaked energy and calculates its probability and then compares it to a randomly generated threshold value.

The reason for doing this is because say the algorithm encounters a local optimum, it may get stuck in the same place thus causing the algorithm to fail. In this algorithm we avoid this by allowing the solution to take values which may not be best for that particular iteration.

The solution that we have thus far obtained is then compared to the best solution we have so far. If its energy is lower, we copy the new solution into the best one at the end of each temperature change.

The program will terminate when we reach our chosen final temperature thus giving us an optimal solution.