CODE Explanation :

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>


using namespace std;
```

// This class represents a point in two-dimensional space. It has two public members, x and y, which represent the x-coordinate and y-coordinate of the point, respectively.

```
class Point {
public:
  int x, y;


  Point() : x(0), y(0) {}

  Point(int x, int y) : x(x), y(y) {}
};
```

// This class defines a dynamic array that can hold Point objects. It uses a vector to store the elements of the array, and it automatically resizes the array when it needs to add more elements.

```
class DynamicArray {
public:
  DynamicArray(int capacity = 10) {
    array_ = vector<Point>(capacity);
    size_ = 0;
  }


  void append(Point point) {
    if (size_ >= array_.size()) {
      array_.resize(array_.size() * 2);
    }
    array_[size_] = point;
    size_++;
  }


  Point get(int index) {
    if (index < 0 || index >= size_) {
      throw std::out_of_range("Index out of bounds");
    }
    return array_[index];
  }


  int getSize() {
    return size_;
  }
```

```
private:
    vector<Point> array_;
    int size_;
};
```

// This function merges two sorted dynamic arrays into a single sorted dynamic array. It does this by comparing the elements of the two arrays one by one and adding the smaller element to the merged array. The *function terminates when both input arrays are empty.*

```
void mergeArrays(DynamicArray& left, DynamicArray& right, DynamicArray& merged) {
    int leftIndex = 0;
    int rightIndex = 0;


    while (leftIndex < left.getSize() && rightIndex < right.getSize()) {
        if (left.get(leftIndex).x < right.get(rightIndex).x) {
            merged.append(left.get(leftIndex));
            leftIndex++;
        } else {
            merged.append(right.get(rightIndex));
            rightIndex++;
        }
    }


    while (leftIndex < left.getSize()) {
        merged.append(left.get(leftIndex));
        leftIndex++;
    }


    while (rightIndex < right.getSize()) {
        merged.append(right.get(rightIndex));
        rightIndex++;
    }
}
```

// This function sorts a dynamic array using the merge sort algorithm. It works by recursively dividing the array into two halves, sorting each half, and then merging the sorted halves back together.

```
void mergeSort(DynamicArray& array) {
    if (array.getSize() <= 1) {
        return;
    }


    int mid = array.getSize() / 2;
    DynamicArray left(mid);
    DynamicArray right(array.getSize() - mid);
```

```
    for (int i = 0; i < mid; i++) {
        left.append(array.get(i));
    }

    for (int i = mid; i < array.getSize(); i++) {
        right.append(array.get(i));
    }

    mergeSort(left);
    mergeSort(right);

    mergeArrays(left, right, array);
}
```

// This function finds the top-right point in a staircase pattern. It first sorts the input DynamicArray of points using merge sort algorithm and then returns the last(top-right) point.

```
Point topRightStaircase(DynamicArray& points) {
    mergeSort(points);

    // Returning the top-right point.
    return points.get(points.getSize() - 1);
}
```

// This function Reads a specified number of points from the user and returns a DynamicArray containing those points.

```
DynamicArray readPoints() {
    DynamicArray points;

    int numPoints;
    cout << "Enter the number of points: ";
    cin >> numPoints;

    for (int i = 0; i < numPoints; i++) {
        int x, y;
        cout << "Enter point " << i + 1 << ": ";
        cin >> x >> y;

        points.append(Point(x, y));
    }

    return points;
```

```
}

// This function prints the coordinates of a given Point object.

void printPoint(Point point) {
   cout << "(" << point.x << ", " << point.y << ")";
}

// The main function reads points from the user, finds the top-right point using the topRightStaircase function, and prints it.
Then, it measures the elapsed time for sorting arrays of different sizes using the merge Sort and topRightStaircase functions.

int main() {
   DynamicArray points = readPoints();

   // Calling the topRightStaircase() function.
   Point topRightPoint = topRightStaircase(points);

   // Printing the top-right point.
   cout << "Top-right point is: ";
   printPoint(topRightPoint);
   cout << endl;

   // Printing the elapsed time for different values of n
   for (int i = 10; i <= 100000000; i *= 10) {
      for (int j = 0; j < i; j++) {
         points.append(Point(j, j));
      }

      // Starting the timer.
      auto start = std::chrono::high_resolution_clock::now();

      // Calling the topRightStaircase() function.
      topRightPoint = topRightStaircase(points);



      // Stopping the timer.
      auto end = std::chrono::high_resolution_clock::now();

      // Calculating the time elapsed.
      auto elapsedTime = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();



      // Printing the time elapsed to the console.
      std::cout << "Time elapsed for n = " << i << ": " << elapsedTime << " nanoseconds" << std::endl;
   }

   return 0;
}
```