CSCI 135 Software Design and Analysis, C++
Homework 6
Solution

Saad Mneimneh
Hunter College of CUNY

Problem 1: A Point class
Consider the following class declaration for a point in 2D:

```
class Point {
  double x_coord;
  double y_coord;

 public:
  Point(double x, double y);
  void moveTo(double x, double y);
  double x();
  double y();
  void print(); //outputs x_coord followed by ','
};              //followed y_coord followed by new line
```

(a) Implement the five public functions including the constructor.

(b) Modify the class so that the following will compile:

```
int main() {
  const int n=...; //some value
  Point a[n];
  .
  .
  .
}
```

(c) Add a public function for class Point called seesToLeft with the following signature:

```
bool seesToLeft(Point p, Point q)
```

Imagine yourself standing at a point a. A function call a.seeToLeft(b,c) should return true iff when walking towards point b, point c is on your left. We will not worry about three points being on the same line. It is not obvious how to implement this function, so here's the condition for this to be true:

```
(a.y()-c.y())*(a.x()-b.x())>(a.y()-b.y())*(a.x()-c.x())
```

(d) Declare an array of 50 points randomly situated in the square with corners (0,0) and (100,100). To generate a random number in $[0, U]$ use

```
(double)rand()/RAND_MAX*U
```

Solution:

```
#include <iostream>
#include <cstdlib>

using std::cout;

class Point {
  double x_coord;
  double y_coord;

 public:

  //this default constructor is for part (b)
  Point() {
    x_coord=y_coord=0;
  }

  //part (a)
  Point(double x, double y) {
    moveTo(x,y);
  }

  void moveTo(double x, double y) {
    x_coord=x;
    y_coord=y;
  }
  double x() {
    return x_coord;
  }

  double y() {
    return y_coord;
  }

  void print() {
    cout<<x_coord<<',''<<y_coord<<'\n';
  }

  //part (c)
  bool seesToLeft(Point p, Point q) {
    return ((y_coord-p.y_coord)*(x_coord-q.x_coord)
           <(y_coord-q.y_coord)*(x_coord-p.x_coord));
  }
};

//part (d)
int main() {
  srand(time(0));
  const int n=50;
  Point a[n];
  for (int i=0; i<n; i=i+1) {
    a[i].moveTo((double)rand()/RAND_MAX*2*n,
                (double)rand()/RAND_MAX*2*n);
    a[i].print();
  }
}
```

Problem 2: Zoolander
This problem is inspired by a movie with the same title about a fashion model who cannot turn left. Given an array of points, the goal is to sort the array in a special way: if we have a walk that goes through $a[0], a[1], \ldots, a[n-1]$, we never make a left turn. We can always start at the lowest point, i.e. the one with the smallest y coordinate. Let's call this the zoolander sorting.

(a) Write a function called lowest that takes as parameters an array of points and its length and returns the index of the lowest point in the array.

(b) Write a function called swap that takes as parameters an array a of points, and two indices i and j, and swaps the two points $a[i]$ and $a[j]$.

(c) Write a function called next that takes as parameters an array a of points, and two indices i and j and returns an index k in $[i, j]$ such that no l in $[i, j]$

makes $a[i-1]$.seesToLeft$(a[k], a[l])$ true.

(d) Using parts (a), (b), and (c), zoolander sort the array that you create in Problem 1. Then display all the points in the array, cut and paste in excel, and verify using a scatter line chart that the walk makes no left turns, i.e. it is a clockwise spiral to the center.

Solution:

```
int lowest(Point * a, int n) {
  double y=a[0].y();
  int index=0;
  for (int i=1; i<n; i=i+1) //this is the standard
    if (a[i].y()<y) {       //algorithm to find the
      y=a[i].y();           //lowest by keeping track
      index=i;              //of it
    }
  return index;
}

//this should be trivial by now
void swap(Point * a, int i, int j) {
  Point temp=a[i];
  a[i]=a[j];
  a[j]=temp;
}

int next(Point * a, int i, int j) {
  int k=i;
  for (int l=i+1; l<=j; l=l+1)
    if (a[i-1].seesToLeft(a[k], a[l])) //if l is to the left
      k=l;                             //change k to l
  return k;                            //(similar to logic of
}                                      // finding min or max)

//straight forward given the above
//see chapter 6 for sorting arrays
void zoolanderSort(Point * a, int n) {
  swap(a,0,lowest(a,n));
  for (int i=1; i<n-1; i=i+1)
    swap(a, i, next(a, i, n-1));
}
```

```
int main() {
  srand(time(0));
  const int n=50;
  Point a[n];
  for (int i=0; i<n; i=i+1) {
    a[i].moveTo((double)rand()/RAND_MAX*2*n,
                (double)rand()/RAND_MAX*2*n);
    a[i].print();
  }
  cout<<"\n\n\n"; //just a separator
  zoolanderSort(a,n);
  for (int i=0; i<n;; i=i+1)
    a[i].print();
}
```

Problem 3: A rectangle learning game
Consider the following class:

```
class Rectangle {
  Point LL;  //lower left corner of rectangle
  Point UR;  //upper right corner of rectangle

 public:
  Rectangle() {...}              //creates a random rectangle
                                 //inside the square with
                                 //corners (0,0) and (100,100)

  bool contains(Point p) {...}   //returns true iff p inside rectangle
};
```

(a) Complete the implementation of the class Rectangle.

(b) Observe that once you create a rectangle, there is no way to tell what that rectangle is. Write a program that first creates a rectangle and then attempts to learn what the rectangle is by using the member function contains repeatedly on random sample points. The idea is to keep track of the largest rectangle that is contained in the original one.

(c) Define the probability of error to be the probability that a point in the original rectangle lies outside your learned rectangle. Let R1 be the original rectangle and R2 your learned rectangle. Therefore,

$$Prob(\text{error}) = \frac{\text{number of points in R1 but not in R2}}{\text{number of points in R1}}$$

Compute this probability. Experiment with different numbers of sample points and observe their effect on the probability of error.

Solution:

```
class Rectangle {
  Point LL;
  Point UR;
 public:
  static const int n; //this is a way to create a "global" variable in Rectangle
  Rectangle() {
    LL=Point((double)rand()/RAND_MAX*n, (double)rand()/RAND_MAX*n);
    UR=Point(LL.x()+(double)rand()/RAND_MAX*(n-LL.x()),
             LL.y()+(double)rand()/RAND_MAX*(n-LL.y()));
  }
  bool contains(Point p) {
    return (LL.x()<=p.x() && LL.y()<=p.y() && UR.x()>=p.x() && UR.y()>=p.y());
  }
};

const int Rectangle::n=100; //initialize "global" n in Rectangle

int main() {
  //you can experiment with these
  int learn_num=100000; //number of points for learning
  int test_num=100000;  //number of test points

  srand(time(0));
  Rectangle r=Rectangle();

  //initial guess is empty rectangle
  int n=Rectangle::n;
  Point LL=Point(n,n);
  Point UR=Point(0,0);

  //now let's learn the rectangle
  for (int i=0; i<learn_num; i=i+1) {
    Point p=Point((double)rand()/RAND_MAX*n, (double)rand()/RAND_MAX*n);
    if (r.contains(p)) {  //update the learned rectangle if necessary
      if (LL.x()>p.x())
        LL.moveTo(p.x(), LL.y());
      if (UR.x()<p.x())
        UR.moveTo(p.x(), UR.y());
      if (LL.y()>p.y())
        LL.moveTo(LL.x(),p.y());
      if (UR.y()<p.y())
        UR.moveTo(UR.x(), p.y());
    }
  }
  cout<<''the learned rectangle is\n'';
  LL.print();
  UR.print();

  //now check how good our learning is

  //these will count the hits in the original and learned rectangles
  int hit_original=0;
  int hit_learned=0;

  for (int i=0; i<test_num; i=i+1) {
    Point p=Point((double)rand()/RAND_MAX*n, (double)rand()/RAND_MAX*n);
    //update counts
    if (r.contains(p)) {
      hit_original=hit_original+1;
      bool b=LL.x()<=p.x() &&
             LL.y()<=p.y() &&
             UR.x()>=p.x() &&
             UR.y()>=p.y();  //is point inside the learned rectangle too?
      if (b)                 //relevant only when we hit the original rectangle
        hit_learned=hit_learned+1;
    }
  }

  if (hit_original!=0)
    cout<<''probability of error is ``
        <<1-(double)hit_learned/hit_original<<'\n';
  else
    cout<<''original rectangle was never hit!\n'';
}
```