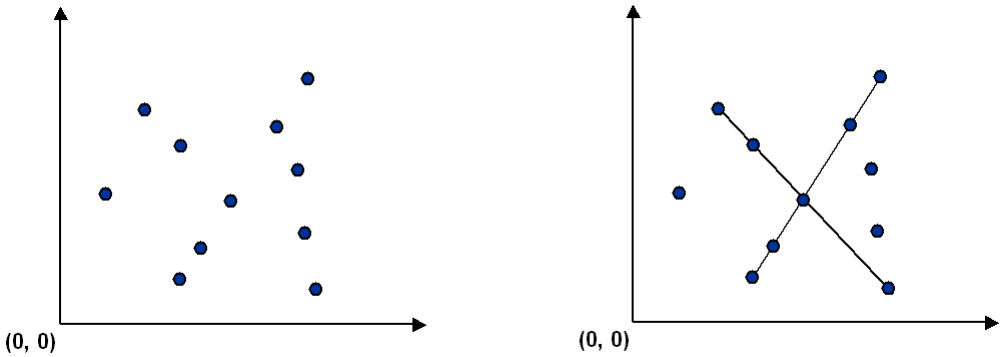
Programming Assignment 3: Pattern Recognition

Write a program to recognize line patterns in a given set of points.

Computer vision involves analyzing patterns in visual images and reconstructing the real-world objects that produced them. The process in often broken up into two phases: *feature detection* and *pattern recognition*. Feature detection involves selecting important features of the image; pattern recognition involves discovering patterns in the features. We will investigate a particularly clean pattern recognition problem involving points and line segments. This kind of pattern recognition arises in many other applications such as statistical data analysis.

The problem. Given a set of *N* distinct points in the plane, draw every (maximal) line segment that connects a subset of 4 or more of the points.



Point data type. Create an immutable data type Point that represents a point in the plane by implementing the following API:

```
public class Point implements Comparable<Point> {
   public final Comparator<Point> SLOPE_ORDER;       // compare points by slope to this point

   public Point(int x, int y)                        // construct the point (x, y)

   public   void draw()                              // draw this point
   public   void drawTo(Point that)                  // draw the line segment from this point to that
point
   public String toString()                          // string representation

   public    int compareTo(Point that)               // is this point lexicographically smaller than that
point?
   public double slopeTo(Point that)                 // the slope between this point and that point
}
```

To get started, use the data type Point.java, which implements the constructor and the draw(), drawTo(), and toString() methods. Your job is to add the following components.

- The compareTo() method should compare points by their *y*-coordinates, breaking ties by their *x*-coordinates. Formally, the invoking point (x_0, y_0) is *less than* the argument point (x_1, y_1) if and only if either $y_0 < y_1$ or if $y_0 = y_1$ and $x_0 < x_1$.
- The slopeTo() method should return the slope between the invoking point (x_0, y_0) and the argument point (x_1, y_1), which is given by the formula $(y_1 - y_0) / (x_1 - x_0)$. Treat the slope

of a horizontal line segment as positive zero [added 7/29]; treat the slope of a vertical line segment as positive infinity; treat the slope of a degenerate line segment (between a point and itself) as negative infinity.

- The SLOPE_ORDER comparator should compare points by the slopes they make with the invoking point (x_0, y_0). Formally, the point (x_1, y_1) is *less than* the point (x_2, y_2) if and only if the slope $(y_1 - y_0) / (x_1 - x_0)$ is less than the slope $(y_2 - y_0) / (x_2 - x_0)$. Treat horizontal, vertical, and degenerate line segments as in the slopeTo() method.

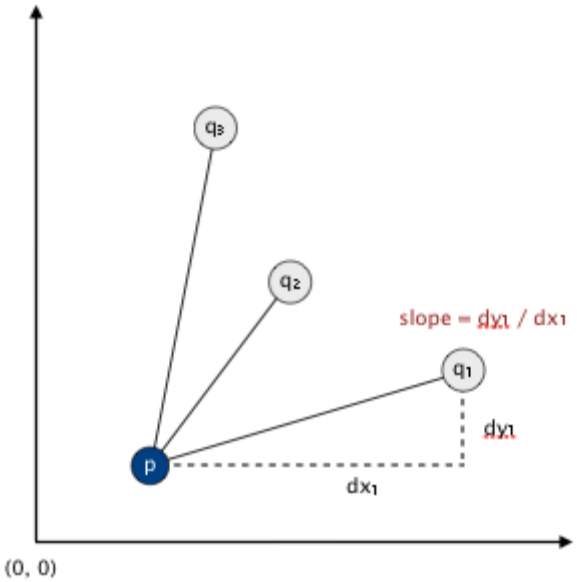
Brute force. Write a program Brute.java that examines 4 points at a time and checks whether they all lie on the same line segment, printing out any such line segments to standard output and drawing them using standard drawing. To check whether the 4 points*p*, *q*, *r*, and *s* are collinear, check whether the slopes between *p* and *q*, between *p* and *r*, and between *p* and *s* are all equal.

The order of growth of the running time of your program should be N^4 in the worst case and it should use space proportional to N.

A faster, sorting-based solution. Remarkably, it is possible to solve the problem much faster than the brute-force solution described above. Given a point *p*, the following method determines whether *p* participates in a set of 4 or more collinear points.

- Think of *p* as the origin.
- For each other point *q*, determine the slope it makes with *p*.
- Sort the points according to the slopes they makes with *p*.
- Check if any 3 (or more) adjacent points in the sorted order have equal slopes with respect to *p*. If so, these points, together with *p*, are collinear.

Applying this method for each of the N points in turn yields an efficient algorithm to the problem. The algorithm solves the problem because points that have equal slopes with respect to *p* are collinear, and sorting brings such points together. The algorithm is fast because the bottleneck operation is sorting.



Write a program Fast.java that implements this algorithm. The order of growth of the running time of your program should be $N^2 \log N$ in the worst case and it should use space proportional to N.

APIs. [Added 7/25] Each program should take the name of an input file as a command-line argument, read the input file (in the format specified below), print to standard output the line segments discovered (in the format specified below), and draw to standard draw the line segments discovered (in the format specified below). Here are the APIs.

```
public class Brute {
   public static void main(String[] args)
}

public class Fast {
   public static void main(String[] args)
}
```

Input format. Read the points from an input file in the following format: An integer *N*, followed by *N* pairs of integers (*x*, *y*), each between 0 and 32,767. Below are two examples.

% **more input6.txt**

```
6
19000  10000
18000  10000
32000  10000
21000  10000
 1234   5678
14000  10000
```

% **more input8.txt**

```
8
10000      0
    0  10000
 3000   7000
 7000   3000
20000  21000
 3000   4000
14000  15000
 6000   7000
```

Output format. Print to standard output the line segments that your program discovers, one per line. Print each line segment as an *ordered* sequence of its constituent points, separated by " -> ".

% **java Brute input6.txt**

```
(14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (21000, 10000)
(14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (32000, 10000)
(14000, 10000) -> (18000, 10000) -> (21000, 10000) -> (32000, 10000)
(14000, 10000) -> (19000, 10000) -> (21000, 10000) -> (32000, 10000)
(18000, 10000) -> (19000, 10000) -> (21000, 10000) -> (32000, 10000)
```

% **java Brute input8.txt**

```
(10000, 0) -> (7000, 3000) -> (3000, 7000) -> (0, 10000)
(3000, 4000) -> (6000, 7000) -> (14000, 15000) -> (20000, 21000)
```

% **java Fast input6.txt**

```
(14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (21000, 10000) -> (32000, 10000)
```

% **java Fast input8.txt**

```
(10000, 0) -> (7000, 3000) -> (3000, 7000) -> (0, 10000)
(3000, 4000) -> (6000, 7000) -> (14000, 15000) -> (20000, 21000)
```

Also, draw the points using draw() and draw the line segments using drawTo(). Your programs should call draw() once for each point in the input file and it should call drawTo() once for each line segment discovered. Before drawing, use StdDraw.setXscale(0, 32768) and StdDraw.setYscale(0, 32768) to rescale the coordinate system.

For full credit, do not print *permutations* of points on a line segment (e.g., if you output $p \to q \to r \to s$, do not also output either $s \to r \to q \to p$ or $p \to r \to q \to s$). Also, for full credit in Fast.java, do not print or plot *subsegments* of a line segment containing 5 or more points (e.g., if you output $p \to q \to r \to s \to t$, do not also output either $p \to q \to s \to t$ or $q \to r \to s \to t$); you may print out such subsegments in Brute.java.

Deliverables. Submit only Brute.java, Fast.java, and Point.java. We will supply stdlib.jar and algs4.jar. Your may not call any library functions other than those in java.lang, java.util, stdlib.jar, and algs4.jar.

This assignment was developed by Kevin Wayne.

Programming Assignment

8 Puzzle

Write a program to solve the 8-puzzle problem (and its natural generalizations) using the A* search algorithm.

The problem. The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order, using as few moves as possible. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an *initial board* (left) to the *goal board* (right).

```
    1  3        1     3        1  2  3        1  2  3        1  2  3
 4  2  5   =>   4  2  5   =>   4     5   =>   4  5     =>   4  5  6
 7  8  6        7  8  6        7  8  6        7  8  6        7  8

initial        1 left          2 up          5 left          goal
```

Best-first search. Now, we describe a solution to the problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a *search node* of the game to be a board, the number of moves made to reach the board, and the previous search node. First, insert the initial search node (the initial board, 0 moves, and a null previous search node) into a priority queue. Then, delete from the priority queue the search node with the minimum priority, and insert onto the priority queue all neighboring search nodes (those that can be reached in one move from the dequeued search node). Repeat this procedure until the search node dequeued corresponds to a goal board. The success of this approach hinges on the choice of *priority function* for a search node. We consider two priority functions:

- *Hamming priority function.* The number of blocks in the wrong position, plus the number of moves made so far to get to the search node. Intutively, a search node with a small number of blocks in the wrong position is close to

the goal, and we prefer a search node that have been reached using a small number of moves.

- *Manhattan priority function.* The sum of the Manhattan distances (sum of the vertical and horizontal distance) from the blocks to their goal positions, plus the number of moves made so far to get to the search node.

For example, the Hamming and Manhattan priorities of the initial search node below are 5 and 10, respectively.

```
 8  1  3        1  2  3     1  2  3  4  5  6  7  8    1  2  3  4  5  6  7  8
 4     2        4  5  6     ----------------------    ----------------------
 7  6  5        7  8        1  1  0  0  1  1  0  1    1  2  0  0  2  2  0  3

 initial           goal          Hamming = 5 + 0          Manhattan = 10 + 0
```

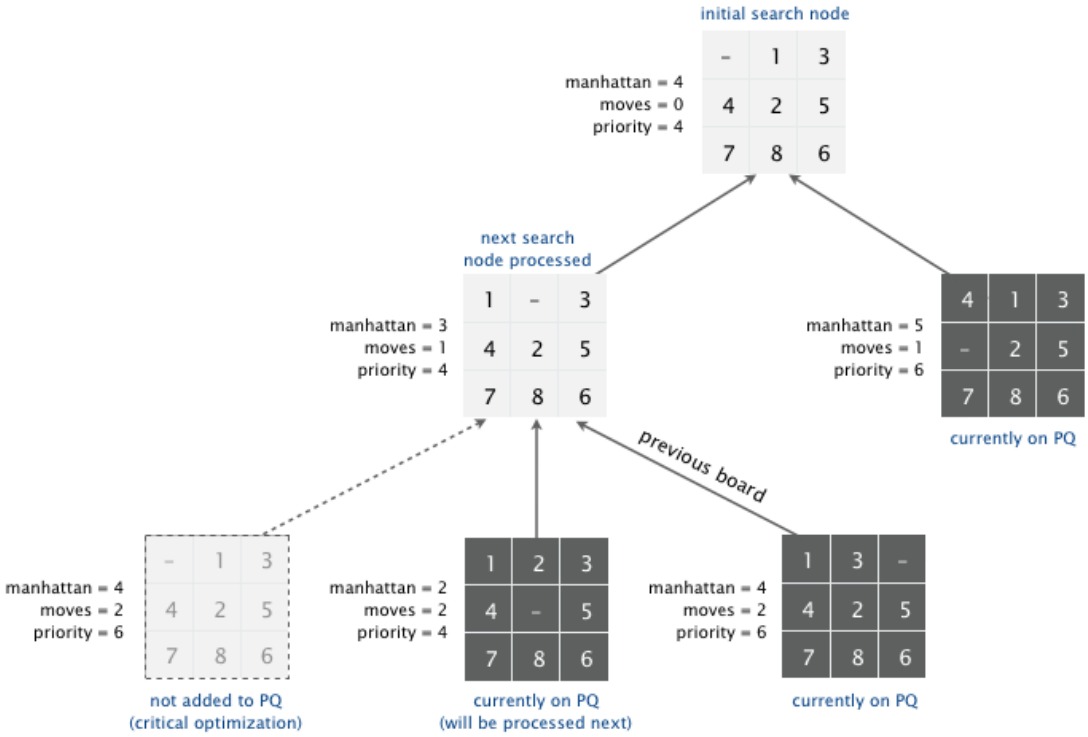
We make a key oberservation: To solve the puzzle from a given search node on the priority queue, the total number of moves we need to make (including those already made) is at least its priority, using either the Hamming or Manhattan priority function. (For Hamming priority, this is true because each block that is out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each block must move its Manhattan distance from its goal position. Note that we do not count the blank square when computing the Hamming or Manhattan priorities.) Consequently, when the goal board is dequeued, we have discovered not only a sequence of moves from the initial board to the goal board, but one that makes the fewest number of moves. (Challenge for the mathematically inclined: prove this fact.)

A critical optimization. Best-first search has one annoying feature: search nodes corresponding to the same board are enqueued on the priority queue many times. To reduce unnecessary exploration of useless search nodes, when considering the neighbors of a search node, don't enqueue a neighbor if its board is the same as the board of the previous search node.

```
 8  1  3        8  1  3        8  1        8  1  3     8  1  3
 4     2        4  2           4  2  3     4     2     4  2  5
 7  6  5        7  6  5        7  6  5     7  6  5     7  6

 previous     search node     neighbor    neighbor     neighbor
                                         (disallow)
```

Game tree. One way to view the computation is as a *game tree*, where each search node is a node in the game tree and the children of a node correspond to its neighboring search nodes. The root of the game tree is the initial search node; the internal nodes have already been processed; the leaf nodes are maintained in a priority queue; at each step, the A* algorithm removes the node with the smallest priority from the priority queue and processes it (by adding its children to both the game tree and the priority queue).



Detecting infeasible puzzles. Not all initial boards can lead to the goal board such as the one below.

```
 1  2  3
 4  5  6
 8  7

infeasible
```

To detect such situations, use the fact that boards are divided into two equivalence classes with respect to reachability: (i) those that lead to the goal board and (ii) those that lead to the goal board if we modify the initial board by swapping any pair of adjacent (non-blank) blocks in the same row. (Difficult challenge for the mathematically inclined: prove this fact.) To apply the fact, run the A* algorithm simultaneously on two puzzle instances—one with the initial board and one with the initial board modified by swapping a pair of adjacent blocks in the same row. Exactly one of the two will lead to the goal board.

Board and Solver data types. Organize your program by creating an immutable data type `Board` with the following API:

```
public class Board {
    public Board(int[][] blocks)           // construct a board from an N-by-
N array of blocks
                                           // (where blocks[i][j] = block in
row i, column j)
    public int dimension()                 // board dimension N
    public int hamming()                   // number of blocks out of place
    public int manhattan()                 // sum of Manhattan distances
between blocks and goal
    public boolean isGoal()                // is this board the goal board?
```

```
    public Board twin()                    // a board obtained by exchanging
two adjacent blocks in the same row
    public boolean equals(Object y)        // does this board equal y?
    public Iterable<Board> neighbors()     // all neighboring boards
    public String toString()               // string representation of the
board (in the output format specified below)
}
```

and an immutable data type `Solver` with the following API:

```
public class Solver {
    public Solver(Board initial)           // find a solution to the initial
board (using the A* algorithm)
    public boolean isSolvable()            // is the initial board solvable?
    public int moves()                     // min number of moves to solve
initial board; -1 if no solution
    public Iterable<Board> solution()      // sequence of boards in a
shortest solution; null if no solution
    public static void main(String[] args) // solve a slider puzzle (given
below)
}
```

To implement the A* algorithm, you must use the `MinPQ` data type from `algs4.jar` for the priority queues.

Solver test client. Use the following test client to read a puzzle from a file (specified as a command-line argument) and print the solution to standard output.

```
public static void main(String[] args) {
    // create initial board from file
    In in = new In(args[0]);
    int N = in.readInt();
    int[][] blocks = new int[N][N];
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            blocks[i][j] = in.readInt();
    Board initial = new Board(blocks);

    // solve the puzzle
    Solver solver = new Solver(initial);

    // print solution to standard output
    if (!solver.isSolvable())
        StdOut.println("No solution possible");
    else {
        StdOut.println("Minimum number of moves = " + solver.moves());
        for (Board board : solver.solution())
            StdOut.println(board);
    }
}
```

Input and output formats. The input and output format for a board is the board dimension *N* followed by the *N*-by-*N* initial board, using 0 to represent the blank square. As an example,

```
% more puzzle04.txt
3
 0  1  3
 4  2  5
 7  8  6
```

```
% java Solver puzzle04.txt
Minimum number of moves = 4

3
 0  1  3
 4  2  5
 7  8  6

3
 1  0  3
 4  2  5
 7  8  6

3
 1  2  3
 4  0  5
 7  8  6

3
 1  2  3
 4  5  0
 7  8  6

3
 1  2  3
 4  5  6
 7  8  0
% more puzzle-unsolvable3x3.txt
3
 1  2  3
 4  5  6
 8  7  0

% java Solver puzzle3x3-unsolvable.txt
No solution possible
```

Your program should work correctly for arbitrary *N*-by-*N* boards (for any $2 \le N < 128$), even if it is too slow to solve some of them in a reasonable amount of time.

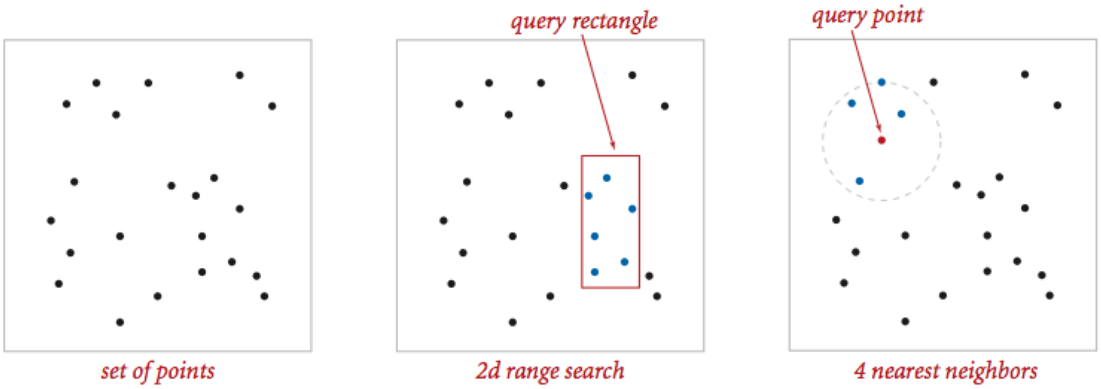
Deliverables. Submit the files `Board.java` and `Solver.java` (with the Manhattan priority). We will supply `stdlib.jar` and `algs4.jar`. Your may not call any library functions other than those in `java.lang`, `java.util`, `stdlib.jar`, and `algs4.jar`. You must use the `MinPQ` data type from `algs4.jar` for the priority queues.

****

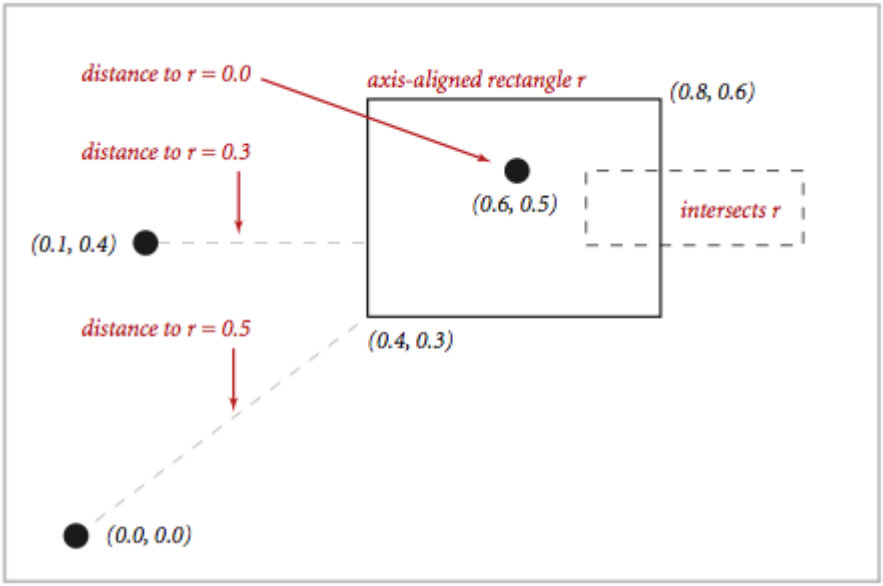
Programming Assignment 5: Kd-Trees

Write a data type to represent a set of points in the unit square (all points have *x*- and *y*-coordinates between 0 and 1) using a *2d-tree* to support efficient *range search* (find all of the points contained in a query rectangle) and *nearest neighbor search* (find a closest point to a query point). 2d-trees have numerous applications,

ranging from classifying astronomical objects to computer animation to speeding up neural networks to mining data to image retrieval.



Geometric primitives. To get started, use the following geometric primitives for points and axis-aligned rectangles in the plane.



Use the immutable data type Point2D.java (part of `algs4.jar`) for points in the plane. Here is the subset of its API that you may use:

```
public class Point2D implements Comparable<Point2D> {
   public Point2D(double x, double y)              // construct the point (x,
y)
   public double x()                               // x-coordinate
   public double y()                               // y-coordinate
   public double distanceTo(Point2D that)          // Euclidean distance
between two points
   public double distanceSquaredTo(Point2D that)   // square of Euclidean
distance between two points
   public int compareTo(Point2D that)              // for use in an ordered
symbol table
   public boolean equals(Object that)              // does this point equal
that?
   public void draw()                              // draw to standard draw
```

```
   public String toString()                          // string representation
}
```

Use the immutable data type RectHV.java (not part of `algs4.jar`) for axis-aligned rectanges. Here is the subset of its API that you may use:

```
public class RectHV {
   public RectHV(double xmin, double ymin,          // construct the rectangle
[xmin, xmax] x [ymin, ymax]
                 double xmax, double ymax)          // throw a
java.lang.IllegalArgumentException if (xmin > xmax) or (ymin > ymax)
   public double xmin()                             // minimum x-coordinate of
rectangle
   public double ymin()                             // minimum y-coordinate of
rectangle
   public double xmax()                             // maximum x-coordinate of
rectangle
   public double ymax()                             // maximum y-coordinate of
rectangle
   public boolean contains(Point2D p)               // does this rectangle
contain the point p (either inside or on boundary)?
   public boolean intersects(RectHV that)           // does this rectangle
intersect that rectangle (at one or more points)?
   public double distanceTo(Point2D p)              // Euclidean distance from
point p to the closest point in rectangle
   public double distanceSquaredTo(Point2D p)       // square of Euclidean
distance from point p to closest point in rectangle
   public boolean equals(Object that)               // does this rectangle
equal that?
   public void draw()                               // draw to standard draw
   public String toString()                         // string representation
}
```

Do not modify these data types.

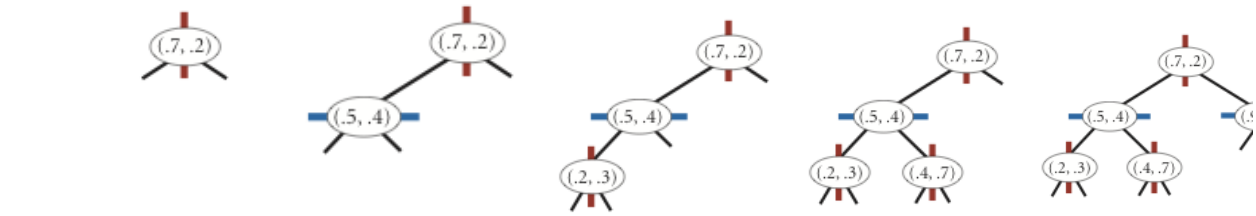
Brute-force implementation. Write a mutable data type `PointSET.java` that represents a set of points in the unit square. Implement the following API by using a red-black BST (using either `SET` from `algs4.jar` or `java.util.TreeSet`).

```
public class PointSET {
   public PointSET()                                // construct an empty set
of points
   public boolean isEmpty()                         // is the set empty?
   public int size()                                // number of points in the
set
   public void insert(Point2D p)                    // add the point p to the
set (if it is not already in the set)
   public boolean contains(Point2D p)               // does the set contain
the point p?
   public void draw()                               // draw all of the points
to standard draw
   public Iterable<Point2D> range(RectHV rect)      // all points in the set
that are inside the rectangle
   public Point2D nearest(Point2D p)                // a nearest neighbor in
the set to p; null if set is empty
}
```

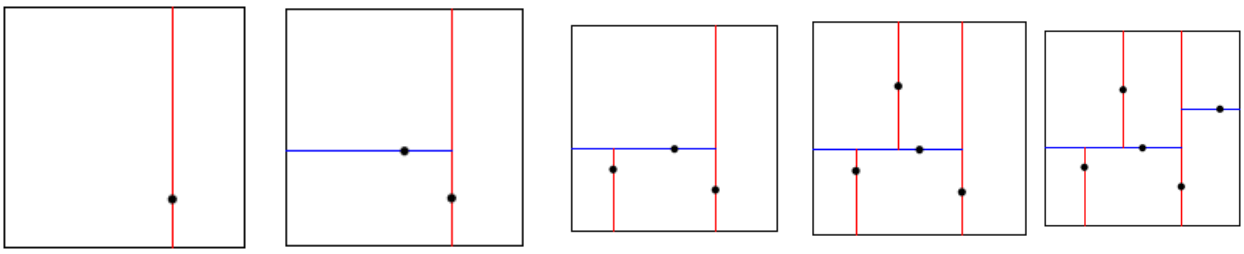
Your implementation should support `insert()` and `contains()` in time proportional to the logarithm of the number of points in the set in the worst case; it should support `nearest()` and `range()` in time proportional to the number of points in the set.

2d-tree implementation. Write a mutable data type `KdTree.java` that uses a 2d-tree to implement the same API (but replace `PointSET` with `KdTree`). A *2d-tree* is a generalization of a BST to two-dimensional keys. The idea is to build a BST with points in the nodes, using the *x*- and *y*-coordinates of the points as keys in strictly alternating sequence.

- *Search and insert.* The algorithms for search and insert are similar to those for BSTs, but at the root we use the *x*-coordinate (if the point to be inserted has a smaller *x*-coordinate than the point at the root, go left; otherwise go right); then at the next level, we use the *y*-coordinate (if the point to be inserted has a smaller *y*-coordinate than the point in the node, go left; otherwise go right); then at the next level the *x*-coordinate, and so forth.



insert (0.7, 0.2) *insert (0.5, 0.4)* *insert (0.2, 0.3)* *insert (0.4, 0.7)* *insert (0.9, 0.6)*



- *Draw.* A 2d-tree divides the unit square in a simple way: all the points to the left of the root go in the left subtree; all those to the right go in the right subtree; and so forth, recursively. Your `draw()` method should draw all of the points to standard draw in black and the subdivisions in red (for vertical splits) and blue (for horizontal splits). This method need not be efficient—it is primarily for debugging.

The prime advantage of a 2d-tree over a BST is that it supports efficient implementation of range search and nearest neighbor search. Each node corresponds to an axis-aligned rectangle in the unit square, which encloses all of the points in its subtree. The root corresponds to the unit square; the left and right children of the root corresponds to the two rectangles split by the *x*-coordinate of the point at the root; and so forth.

- *Range search.* To find all points contained in a given query rectangle, start at the root and recursively search for points in *both* subtrees using the following *pruning rule*: if the query rectangle does not intersect the rectangle corresponding to a node, there is no need to explore that node (or its subtrees). A subtree is searched only if it might contain a point contained in the query rectangle.
- *Nearest neighbor search.* To find a closest point to a given query point, start at the root and recursively search in *both* subtrees using the following *pruning rule*: if the closest point discovered so far is closer than the distance between the query point and the rectangle corresponding to a node, there is no need to explore that node (or its subtrees). That is, a node is searched only if it might contain a point that is closer than the best one found so far. The effectiveness of the pruning rule depends on quickly finding a nearby point. To do this, organize your recursive method so that when there are two possible subtrees to go down, you always choose *the subtree that is on the same side of the splitting line as the query point* as the first subtree to explore—the closest point found while exploring the first subtree may enable pruning of the second subtree.

Clients. You may use the following interactive client programs to test and debug your code.

- KdTreeVisualizer.java computes and draws the 2d-tree that results from the sequence of points clicked by the user in the standard drawing window.
- RangeSearchVisualizer.java reads a sequence of points from a file (specified as a command-line argument) and inserts those points into a 2d-tree. Then, it performs range searches on the axis-aligned rectangles dragged by the user in the standard drawing window.
- NearestNeighborVisualizer.java reads a sequence of points from a file (specified as a command-line argument) and inserts those points into a 2d-tree. Then, it performs nearest neighbor queries on the point corresponding to the location of the mouse in the standard drawing window.

Analysis of running time and memory usage (optional and not graded).

- Give the total memory usage in bytes (using tilde notation) of your 2d-tree data structure as a function of the number of points *N*, using the memory-cost model from lecture and Section 1.4 of the textbook. Count all memory that is used by your 2d-tree, including memory for the nodes, points, and rectangles.
- Give the expected running time in seconds (using tilde notation) to build a 2d-tree on *N* random points in the unit square. (Do not count the time to read in the points from standard input.)
- How many nearest neighbor calculations can your 2d-tree implementation perform per second for input100K.txt (100,000 points) and input1M.txt (1

million points), where the query points are random points in the unit square? (Do not count the time to read in the points or to build the 2d-tree.) Repeat this question but with the brute-force implementation.

Submission. Submit only `PointSET.java` and and `KdTree.java`. We will supply `RectHV.java`, `stdlib.jar`, and `algs4.jar`. Your may not call any library functions other than those in `java.lang`, `java.util`, `stdlib.jar`, and `algs4.jar`.

This assignment was developed by Kevin Wayne.