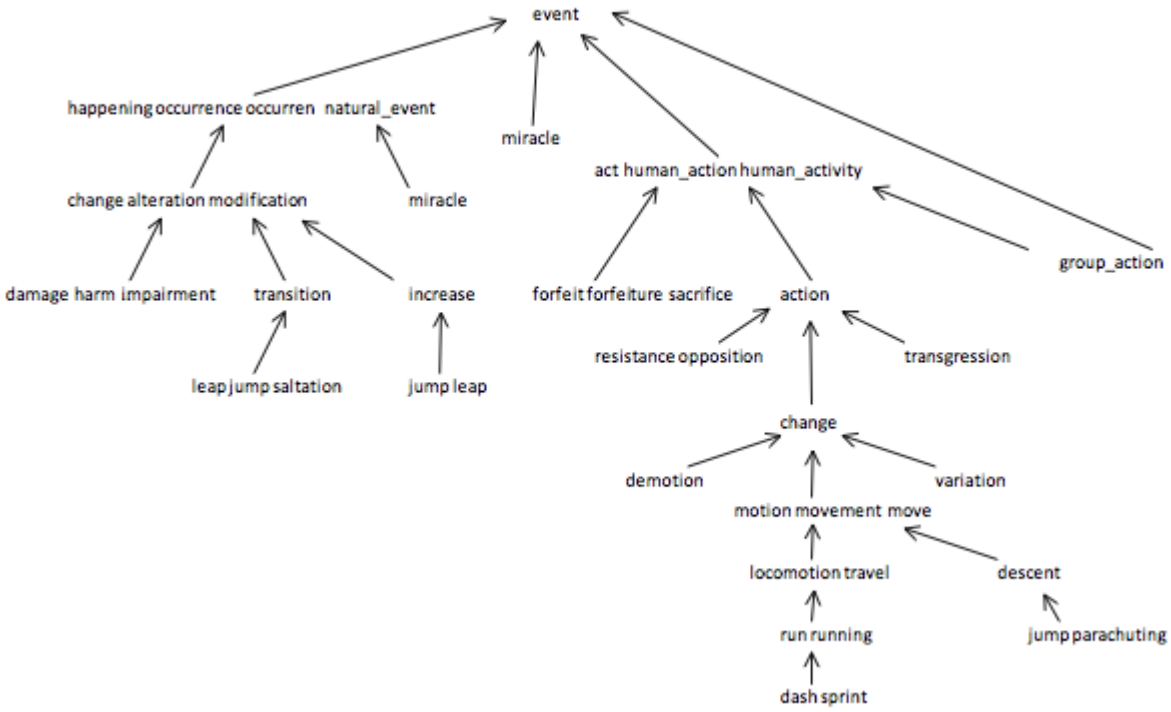
Programming Assignment 1: WordNet

WordNet is a semantic lexicon for the English language that is used extensively by computational linguists and cognitive scientists; for example, it was a key component in IBM's Watson. WordNet groups words into sets of synonyms called *synsets* and describes semantic relationships between them. One such relationship is the *is-a* relationship, which connects a *hyponym* (more specific synset) to a *hypernym* (more general synset). For example, *locomotion* is a hypernym of *running* and *running* is a hypernym of *dash*.

The WordNet digraph. Your first task is to build the wordnet digraph: each vertex *v* is an integer that represents a synset, and each directed edge *v*→*w* represents that *w* is a hypernym of *v*. The wordnet digraph is a *rooted DAG*: it is acylic and has one vertex that is an ancestor of every other vertex. However, it is not necessarily a tree because a synset can have more than one hypernym. A small subgraph of the wordnet digraph is illustrated below.



The WordNet input file formats. We now describe the two data files that you will use to create the wordnet digraph. The files are in *CSV format*: each line contains a sequence of fields, separated by commas.

- *List of noun synsets.* The file synsets.txt lists all the (noun) synsets in WordNet. The first field is the *synset id* (an integer), the second field is the synonym set (or *synset*), and the third field is its dictionary definition (or *gloss*). For example, the line
    - 36,AND_circuit AND_gate,a circuit in a computer that fires only when all of its inputs fire

  means that the synset { AND_circuit, AND_gate } has an id number of 36 and it's gloss is a circuit in a computer that fires only when all of its inputs fire. The individual nouns that comprise a synset are separated by spaces (and a synset element is not permitted to contain a space). The *S* synset ids are numbered 0 through $S - 1$; the id numbers will appear consecutively in the synset file.

- *List of hypernyms.* The file hypernyms.txt contains the hypernym relationships: The first field is a synset id; subsequent fields are the id numbers of the synset's hypernyms. For example, the following line
    - 164,21012,56099

  means that the the synset 164 ("Actifed") has two hypernyms: 21012 ("antihistamine") and 56099 ("nasal_decongestant"), representing that Actifed is both an antihistamine and a nasal decongestant. The synsets are obtained from the corresponding lines in the file synsets.txt.

  > 164,Actifed,trade name for a drug containing an antihistamine and a decongestant...
  > 21012,antihistamine,a medicine used to treat allergies...
  > 56099,nasal_decongestant,a decongestant that provides temporary relief of nasal...

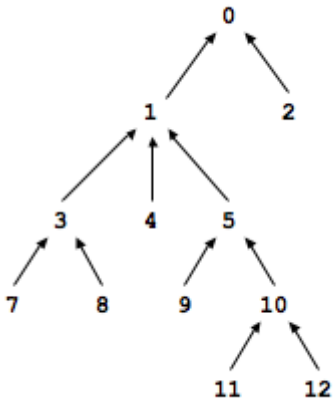
WordNet data type. Implement an immutable data type WordNet with the following API:

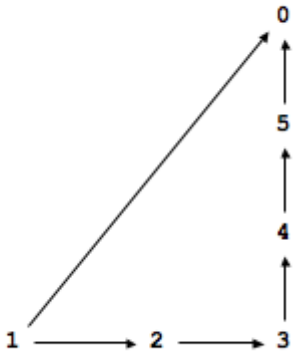
```
// constructor takes the name of the two input files
public WordNet(String synsets, String hypernyms)

// the set of nouns (no duplicates), returned as an Iterable
public Iterable<String> nouns()

// is the word a WordNet noun?
public boolean isNoun(String word)

// distance between nounA and nounB (defined below)
public int distance(String nounA, String nounB)

// a synset (second field of synsets.txt) that is the common ancestor of nounA and nounB
// in a shortest ancestral path (defined below)
public String sap(String nounA, String nounB)

// for unit testing of this class
public static void main(String[] args)
```

The constructor should throw a java.lang.IllegalArgumentException if the input does not correspond to a rooted DAG. The distance() and sap() methods should throw a java.lang.IllegalArgumentException unless both of the noun arguments are WordNet nouns.

Your data type should use space linear in the input size (size of synsets and hypernyms files). The constructor should take time linearithmic (or better) in the input size. The method isNoun() should run in time logarithmic (or better) in the number of nouns. The methods distance() and sap() should run in time linear in the size of the WordNet digraph.

Shortest ancestral path. An *ancestral path* between two vertices *v* and *w* in a digraph is a directed path from *v* to a common ancestor *x*, together with a directed path from *w* to the same ancestor *x*. A *shortest ancestral path* is an ancestral path of minimum total length. For example, in the digraph at left (digraph1.txt), the shortest ancestral path between 3 and 11 has length 4 (with common ancestor 1). In the digraph at right (digraph2.txt), one ancestral path between 1 and 5 has length 4 (with common ancestor 5), but the shortest ancestral path has length 2 (with common ancestor 0).





SAP data type. Implement an immutable data type SAP with the following API:

```
// constructor takes a digraph (not necessarily a DAG)
public SAP(Digraph G)

// length of shortest ancestral path between v and w; -1 if no such path
public int length(int v, int w)

// a common ancestor of v and w that participates in a shortest ancestral path; -1 if no such path
public int ancestor(int v, int w)

// length of shortest ancestral path between any vertex in v and any vertex in w; -1 if no such path
public int length(Iterable<Integer> v, Iterable<Integer> w)

// a common ancestor that participates in shortest ancestral path; -1 if no such path
public int ancestor(Iterable<Integer> v, Iterable<Integer> w)

// for unit testing of this class (such as the one below)
public static void main(String[] args)
```

All methods should throw a java.lang.IndexOutOfBoundsException if one (or more) of the input arguments is not between 0 and G.V() - 1. You may assume that the iterable arguments contain at least one integer. All methods (and the constructor) should take time at most proportional to $E + V$ in the worst case, where E and V are the number of edges and vertices in the digraph, respectively. Your data type should use space proportional to $E + V$.

Test client. The following test client takes the name of a digraph input file as as a command-line argument, constructs the digraph, reads in vertex pairs from standard input, and prints out the length of the shortest ancestral path between the two vertices and a common ancestor that participates in that path:

```
public static void main(String[] args) {
   In in = new In(args[0]);
   Digraph G = new Digraph(in);
   SAP sap = new SAP(G);
   while (!StdIn.isEmpty()) {
      int v = StdIn.readInt();
```

```
        int w = StdIn.readInt();
        int length   = sap.length(v, w);
        int ancestor = sap.ancestor(v, w);
        StdOut.printf("length = %d, ancestor = %d\n", length, ancestor);
    }
}
```

Here is a sample execution:

```
% more digraph1.txt          % java SAP digraph1.txt
13                           3 11
11                           length = 4, ancestor = 1
 7  3
 8  3                        9 12
 3  1                        length = 3, ancestor = 5
 4  1
 5  1                        7 2
 9  5                        length = 4, ancestor = 0
10  5
11 10                         1 6
12 10                         length = -1, ancestor = -1
 1  0
 2  0
```

Measuring the semantic relatedness of two nouns. Semantic relatedness refers to the degree to which two concepts are related. Measuring semantic relatedness is a challenging problem. For example, most of us agree that *George Bush* and *John Kennedy*(two U.S. presidents) are more related than are *George Bush* and *chimpanzee* (two primates). However, not most of us agree that *George Bush* and *Eric Arthur Blair* are related concepts. But if one is aware that *George Bush* and *Eric Arthur Blair* (aka George Orwell) are both communicators, then it becomes clear that the two concepts might be related.

We define the semantic relatedness of two wordnet nouns *A* and *B* as follows:

- *distance(A, B)* = distance is the minimum length of any ancestral path between any synset *v* of *A* and any synset *w* of *B*.

This is the notion of distance that you will use to implement the distance() and sap() methods in the WordNet data type.

Outcast detection. Given a list of wordnet nouns $A_1, A_2, ..., A_n$, which noun is the least related to the others? To identify *an outcast*, compute the sum of the distances between each noun and every other one:

$$d_i = \text{dist}(A_i, A_1) + \text{dist}(A_i, A_2) + ... + \text{dist}(A_i, A_n)$$

and return a noun A_t for which d_t is maximum.

Implement an immutable data type Outcast with the following API:

```
// constructor takes a WordNet object
public Outcast(WordNet wordnet)

// given an array of WordNet nouns, return an outcast
```

```
public String outcast(String[] nouns)

// for unit testing of this class (such as the one below)
public static void main(String[] args)
```

Assume that argument array to the outcast() method contains only valid wordnet nouns (and that it contains at least two such nouns).

The following test client takes from the command line the name of a synset file, the name of a hypernym file, followed by the names of outcast files, and prints out an outcast in each file:

```
public static void main(String[] args) {
   WordNet wordnet = new WordNet(args[0], args[1]);
   Outcast outcast = new Outcast(wordnet);
   for (int t = 2; t < args.length; t++) {
      String[] nouns = In.readStrings(args[t]);
      StdOut.println(args[t] + ": " + outcast.outcast(nouns));
   }
}
```

Here is a sample execution:

% **more outcast5.txt**
horse zebra cat bear table

% **more outcast8.txt**
water soda bed orange_juice milk apple_juice tea coffee

% **more outcast11.txt**
apple pear peach banana lime lemon blueberry strawberry mango watermelon potato

% **java Outcast synsets.txt hypernyms.txt outcast5.txt outcast8.txt outcast11.txt**
outcast5.txt: table
outcast8.txt: bed
outcast11.txt: potato

Analysis of running time (optional). Analyze the effectiveness of your approach to this problem by giving estimates of its time requirements.

- Give the order of growth of the *worst-case* running time of the length() and ancestor() methods in SAP as a function of the number of vertices V and the number of edges E in the digraph.
- Give the order of growth of the *best-case* running time of the same methods.

Deliverables. Submit WordNet.java, SAP.java, and Outcast.java that implement the APIs described above. Also submit any other supporting files (excluding those in 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.

Programming Assignment 2: Seam Carving

Seam-carving is a content-aware image resizing technique where the image is reduced in size by one pixel of height (or width) at a time. A *vertical seam* in an image is a path of pixels connected from the top to the bottom with one pixel in each row. (A*horizontal seam* is a path of pixels connected from the left to the right with one pixel in each column.) Below left is the original 505-by-287 pixel image; below right is the result after removing 150 vertical seams, resulting in a 30% narrower image. Unlike standard content-agnostic resizing techniques (e.g. cropping and scaling), the most interesting features (aspect ratio, set of objects present, etc.) of the image are preserved.

As you'll soon see, the underlying algorithm is quite simple and elegant. Despite this fact, this technique was not discovered until 2007 by Shai Avidan and Ariel Shamir. It is now a feature in Adobe Photoshop (thanks to a Princeton graduate student), as well as other popular computer graphics applications.

In this assignment, you will create a data type that resizes a *W*-by-*H* image using the seam-carving technique.

Finding and removing a seam involves three parts and a tiny bit of notation:

1. *Notation.* In image processing, pixel (*x*, *y*) refers to the pixel in column *x* and row *y*, with pixel (0, 0) at the upper left corner and pixel ($W - 1$, $H - 1$) at the bottom right corner. This is consistent with the Picture data type in stdlib.jar. *Warning*: this is the opposite of the standard mathematical notation used in linear algebra where (*i*, *j*) refers to row *i* and column *j* and with Cartesian coordinates where (0, 0) is at the lower left corner.

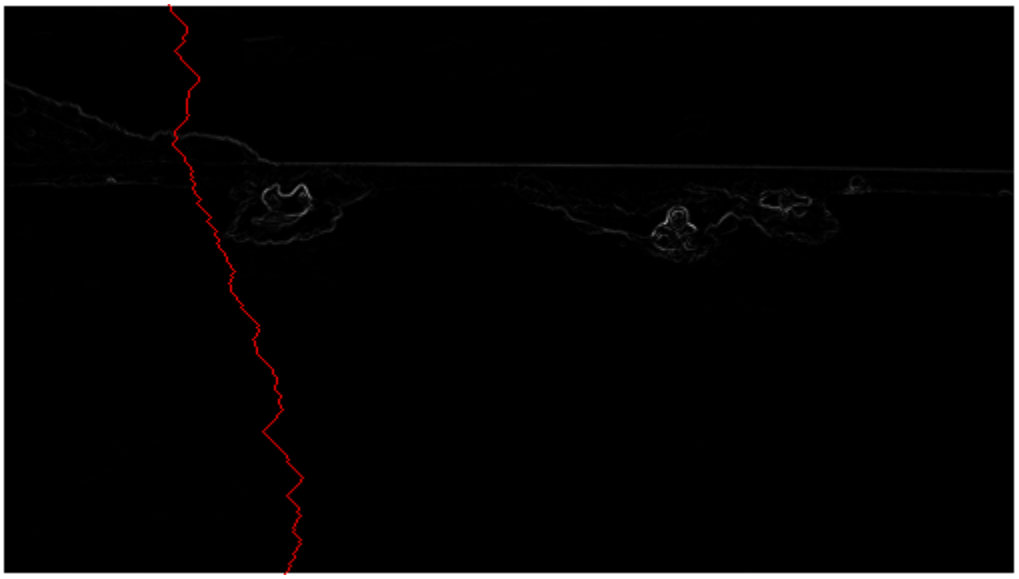
a 3-by-4 image		
(0, 0)	(1, 0)	(2, 0)
(0, 1)	(1, 1)	(2, 1)
(0, 2)	(1, 2)	(2, 2)
(0, 3)	(1, 3)	(2, 3)

2. We also assume that the color of a pixel is represented in RGB space, using three integers between 0 and 255. This is consistent with the java.awt.Color data type.
3. *Energy calculation.* The first step is to calculate the *energy* of each pixel, which is a measure of the importance of each pixel—the higher the energy, the less likely that the pixel will be included as part of a seam (as we'll see in the next step). In this assignment, you will implement the *dual gradient* energy function, which is described below. Here is the dual gradient of the surfing image above:

The energy is high (white) for pixels in the image where there is a rapid color gradient (such as the boundary between the sea and sky and the boundary between the surfing Josh Hug on the left and the ocean behind him). The seam-carving technique avoids removing such high-energy pixels.

4. *Seam identification.* The next step is to find a vertical seam of minimum total energy. This is similar to the classic shortest path problem in an edge-weighted digraph except for the following:
    - The weights are on the vertices instead of the edges.
    - We want to find the shortest path from any of the *W* pixels in the top row to any of the *W* pixels in the bottom row.

- The digraph is acyclic, where there is a downward edge from pixel (x, y) to pixels $(x - 1, y + 1)$, $(x, y + 1)$, and $(x + 1, y + 1)$, assuming that the coordinates are in the prescribed range.



5. *Seam removal.* The final step is to remove from the image all of the pixels along the seam.

The SeamCarver API. Your task is to implement the following mutable data type:

```
public class SeamCarver {
   public SeamCarver(Picture picture)
   public Picture picture()                    // current picture
   public     int width()                    // width  of current picture
   public     int height()                   // height of current picture
   public  double energy(int x, int y)          // energy of pixel at column x and row y in current
picture
   public   int[] findHorizontalSeam()          // sequence of indices for horizontal seam in current
picture
   public   int[] findVerticalSeam()            // sequence of indices for vertical   seam in current
picture
   public    void removeHorizontalSeam(int[] a)   // remove horizontal seam from current picture
   public    void removeVerticalSeam(int[] a)     // remove vertical   seam from current picture
}
```

- **Constructor.** The data type may not mutate the Picture argument to the constructor.
- **Computing the energy of a pixel.** We will use the *dual gradient energy function*: The energy of pixel (x, y) is $\Delta_x^2(x, y) + \Delta_y^2(x, y)$, where the square of the x-gradient $\Delta_x^2(x, y) = R_x(x, y)^2 + G_x(x, y)^2 + B_x(x, y)^2$, and where the central differences$R_x(x, y)$, $G_x(x, y)$, and $B_x(x, y)$ are the absolute value in differences of red, green, and blue components between pixel $(x + 1, y)$ and pixel $(x - 1, y)$. The square of the y-gradient $\Delta_y^2(x, y)$ is defined in an analogous manner. We define the energy of pixels at the border of the image to be $255^2 + 255^2 + 255^2 = 195075$.

  As an example, consider the 3-by-4 image with RGB values (each component is an integer between 0 and 255) as shown in the table below.

(255, 101, 51)	(255, 101, 153)	(255, 101, 255)
(255,153,51)	(255,153,153)	(255,153,255)
(255,203,51)	(255,204,153)	(255,205,255)
(255,255,51)	(255,255,153)	(255,255,255)

The ten border pixels have energy 195075. Only the pixels (1, 1) and (1, 2) are nontrivial. We calculate the energy of pixel (1, 2) in detail:

$R_x(1, 2) = 255 - 255 = 0$,
$G_x(1, 2) = 205 - 203 = 2$,
$B_x(1, 2) = 255 - 51 = 204$,
yielding $\Delta_x^2(1, 2) = 2^2 + 204^2 = 41620$.

$R_y(1, 2) = 255 - 255 = 0$,
$G_y(1, 2) = 255 - 153 = 102$,
$B_y(1, 2) = 153 - 153 = 0$,
yielding $\Delta_y^2(1, 2) = 102^2 = 10404$.

Thus, the energy of pixel (1, 2) is 41620 + 10404 = 52024. Similarly, the energy of pixel (1, 1) is $204^2 + 103^2 = 52225$.

195075.0	195075.0	195075.0
195075.0	52225.0	195075.0
195075.0	52024.0	195075.0
195075.0	195075.0	195075.0

- **Finding a vertical seam.** The findVerticalSeam() method should return an array of length *H* such that entry *x* is the column number of the pixel to be removed from row *x* of the image. For example, consider the 6-by-5 image below (supplied as6x5.png).

(97, 82,107)	(220,172,141)	(243, 71,205)	(129,173,222)	(225, 40,209)	(66,109,219)
(181, 78, 68)	(15, 28,216)	(245,150,150)	(177,100,167)	(205,205,177)	(147, 58, 99)
(196,224, 21)	(166,217,190)	(128,120,162)	(104, 59,110)	(49,148,137)	(192,101, 89)
(83,143,103)	(110, 79,247)	(106, 71,174)	(92,240,205)	(129, 56,146)	(121,111,147)
(82,157,137)	(92,110,129)	(183,107, 80)	(89, 24,217)	(207, 69, 32)	(156,112, 31)

- The corresponding pixel energies are shown below, with a minimum energy vertical seam highlighted in red. In this case, the method findVerticalSeam() should return the array { 2, 3, 3, 3, 2 }.

195075.0	195075.0	195075.0	195075.0	195075.0	195075.0
195075.0	23346.0	51304.0	31519.0	55112.0	195075.0
195075.0	47908.0	61346.0	35919.0	38887.0	195075.0
195075.0	31400.0	37927.0	14437.0	63076.0	195075.0
195075.0	195075.0	195075.0	195075.0	195075.0	195075.0

- When there are multiple vertical seams with minimal total energy (as in the example above), your method can return any such seam.
- **Finding a horizontal seam.** The behavior of findHorizontalSeam() is analogous to that of findVerticalSeam() except that it should return an array of length W such that entry y is the row number of the pixel to be removed from column y of the image.
- **Performance requirements.** The width(), height(), and energy() methods should take constant time in the worst case. All other methods should run in time at most proportional to $W H$ in the worst case. For faster performance, do not construct explicit DirectedEdge and EdgeWeightedDigraph objects.
- **Exceptions.** Your code should throw an exception when called with invalid arguments.
    - By convention, the indices x and y are integers between 0 and $W - 1$ and between 0 and $H - 1$ respectively. Throw a java.lang.IndexOutOfBoundsException if either x or y is outside its prescribed range.
    - Throw a java.lang.IllegalArgumentException if removeVerticalSeam() or removeHorizontalSeam() is called with an array of the wrong length or if the array is not a valid seam (i.e., either an entry is outside its prescribed range or two adjacent entries differ by more than 1).
    - Throw a java.lang.IllegalArgumentException if either removeVerticalSeam() or removeHorizontalSeam() is called when either the width or height is less than or equal to 1.

Analysis of running time (optional and not graded).

- Give the worst-case running time to remove R rows and C columns from a W-by-H image as a function of R, C, W, and H.
- Estimate empirically the running time (in seconds) to remove R rows and C columns from a W-by-H image as a function of R, C, W, and H. Use tilde notation to simplify your answer.

Submission. Submit SeamCarver.java, and any other files needed by your program (excluding those in stdlib.jar and algs4.jar). Your may not call any library functions other than those in java.lang, java.util, java.awt.Color, stdlib.jar, and algs4.jar.

Programming Assignment 3: Baseball Elimination

Given the standings in a sports division at some point during the season, determine which teams have been mathematically eliminated from winning their division.

The baseball elimination problem. In the baseball elimination problem, there is a division consisting of *N* teams. At some point during the season, team *i* has `w[i]` wins, `l[i]` losses, `r[i]` remaining games, and `g[i][j]` games left to play against team *j*. A team is mathematically eliminated if it cannot possibly finish the season in (or tied for) first place. The goal is to determine exactly which teams are mathematically eliminated. For simplicity, we assume that no games end in a tie (as is the case in Major League Baseball) and that there are no rainouts (i.e., every scheduled game is played).

The problem is not as easy as many sports writers would have you believe, in part because the answer depends not only on the number of games won and left to play, but also on the schedule of remaining games. To see the complication, consider the following scenario:

		w[i]	l[i]	r[i]	g[i][j]			
i	team	wins	loss	left	Atl	Phi	NY	Mon
0	Atlanta	83	71	8	-	1	6	1
1	Philadelphia	80	79	3	1	-	0	2
2	New York	78	78	6	6	0	-	0
3	Montreal	77	82	3	1	2	0	-

Montreal is mathematically eliminated since it can finish with at most 80 wins and Atlanta already has 83 wins. This is the simplest reason for elimination. However, there can be more complicated reasons. For example, Philadelphia is also mathematically eliminated. It can finish the season with as many as 83 wins, which appears to be enough to tie Atlanta. But this would require Atlanta to lose all of its remaining games, including the 6 against New York, in which case New York would finish with 84 wins. We note that New York is not yet mathematically eliminated despite the fact that it has fewer wins than Philadelphia.

It is sometimes not so easy for a sports writer to explain why a particular team is mathematically eliminated. Consider the following scenario from the American League East on August 30, 1996:

		w[i]	l[i]	r[i]	g[i][j]				
i	team	wins	loss	left	NY	Bal	Bos	Tor	Det
0	New York	75	59	28	-	3	8	7	3
1	Baltimore	71	63	28	3	-	2	7	4
2	Boston	69	66	27	8	2	-	0	0
3	Toronto	63	72	27	7	7	0	-	0
4	Detroit	49	86	27	3	4	0	0	-

It might appear that Detroit has a remote chance of catching New York and winning the division because Detroit can finish with as many as 76 wins if they go on a 27-game winning steak, which is one more than New York would have if they go on a 28-game losing streak. Try to convince yourself that Detroit is already mathematically eliminated. Here's one ad hoc explanation; we will present a simpler explanation below.

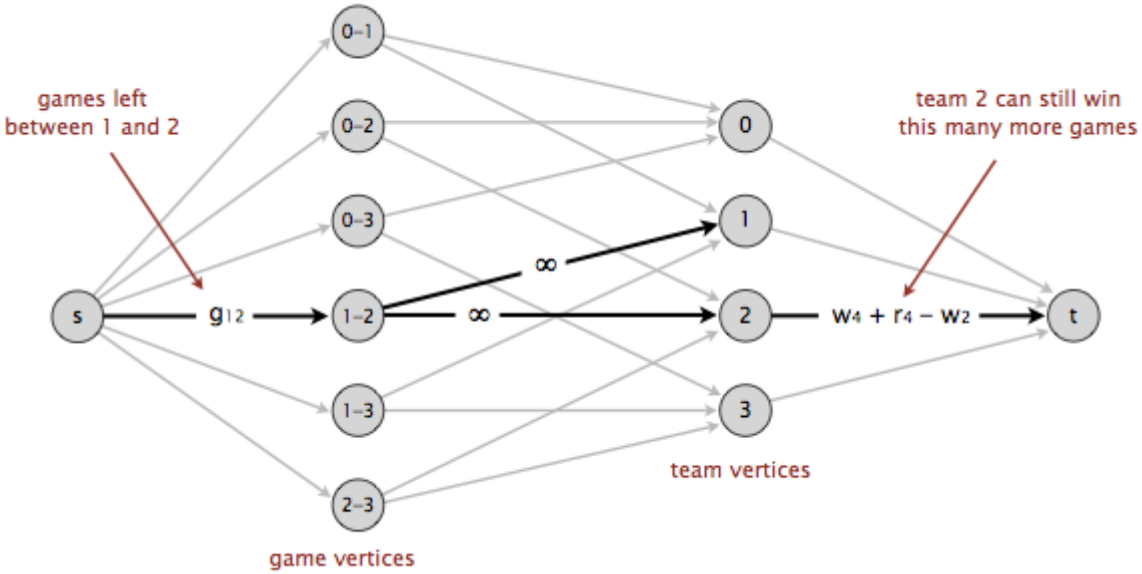
A maxflow formulation. We now solve the baseball elimination problem by reducing it to the maxflow problem. To check whether team *x* is eliminated, we consider two cases.

- *Trivial elimination.* If the maximum number of games team *x* can win is less than the number of wins of some other team *i*, then team *x* is trivially eliminated (as is Montreal in the example above). That is, if `w[x] + r[x] < w[i]`, then team *x* is mathematically eliminated.
- *Nontrivial elimination.* Otherwise, we create a flow network and solve a maxflow problem in it. In the network, feasible integral flows correspond to outcomes of the remaining schedule. There are vertices corresponding to teams (other than team *x*) and to remaining divisional games (not involving team *x*). Intuitively, each unit of flow in the network corresponds to a remaining game. As it flows through the network from *s* to *t*, it passes from a game vertex, say between teams *i* and *j*, then through one of the team vertices *i* or *j*, classifying this game as being won by that team.

  More precisely, the flow network includes the following edges and capacities.

  - We connect an artificial source vertex *s* to each game vertex *i-j* and set its capacity to `g[i][j]`. If a flow uses all `g[i][j]` units of capacity on this edge, then we interpret this as playing all of these games, with the wins distributed between the team vertices *i* and *j*.
  - We connect each game vertex *i-j* with the two opposing team vertices to ensure that one of the two teams earns a win. We do not need to restrict the amount of flow on such edges.
  - Finally, we connect each team vertex to an artificial sink vertex *t*. We want to know if there is some way of completing all the games so that team *x* ends up winning at least as many games as team *i*. Since team *x* can win as many as `w[x] + r[x]` games, we prevent team *i* from winning more than that many games in total, by including an edge from team vertex *i* to the sink vertex with capacity `w[x] + r[x] - w[i]`.

  If all edges in the maxflow that are pointing from *s* are full, then this corresponds to assigning winners to all of the remaining games in such a way that no team wins more games than *x*. If some edges pointing from *s* are not full, then there is no scenario in which team *x* can win the division. In the flow network below Detroit is team *x = 4*.



What the min cut tells us. By solving a maxflow problem, we can determine whether a given team is mathematically eliminated. We would also like to *explain* the reason for the team's elimination to a friend in nontechnical terms (using only grade-school arithmetic). Here's such an explanation for Detroit's elimination in the American League East example above. With the best possible luck, Detroit finishes the season with 49 + 27 = 76 wins. Consider the subset of teams R = { New York, Baltimore, Boston, Toronto }. Collectively, they already have 75 + 71 + 69 + 63 = 278 wins; there are also 3 + 8 + 7 + 2 + 7 = 27 remaining games among them, so these four teams must win at least an additional 27 games. Thus, on average, the teams in R win at least 305 / 4 = 76.25 games. Regardless of the outcome, one team in R will win at least 77 games, thereby eliminating Detroit.

In fact, when a team is mathematically eliminated there always exists such a convincing *certificate of elimination*, where R is some subset of the other teams in the division. Moreover, you can always find such a subset R by choosing the team vertices on the source side of a min s-t cut in the baseball elimination network. Note that although we solved a maxflow/mincut problem to find the subset R, once we have it, the argument for a team's elimination involves only grade-school algebra.

Your assignment. Write an immutable data type `BaseballElimination` that represents a sports division and determines which teams are mathematically eliminated by implementing the following API:

```
public BaseballElimination(String filename)
// create a baseball division from given filename in format
specified below
public              int numberOfTeams()
// number of teams
public Iterable<String> teams()
// all teams
public              int wins(String team)
// number of wins for given team
public              int losses(String team)
// number of losses for given team
```

```
public              int remaining(String team)
// number of remaining games for given team
public              int against(String team1, String team2)
// number of remaining games between team1 and team2
public          boolean isEliminated(String team)
// is given team eliminated?
public Iterable<String> certificateOfElimination(String team)
// subset R of teams that eliminates given team; null if not
eliminated
```

The last six methods should throw a `java.lang.IllegalArgumentException` if one (or both) of the input arguments are invalid teams.

Input format. The input format is the number of teams in the division *N* followed by one line for each team. Each line contains the team name (with no internal whitespace characters), the number of wins, the number of losses, the number of remaining games, and the number of remaining games against each team in the divsion. For example, the input files teams4.txt and teams5.txt correspond to the two examples discussed above.

```
% more teams4.txt
4
Atlanta        83 71  8   0 1 6 1
Philadelphia   80 79  3   1 0 0 2
New_York       78 78  6   6 0 0 0
Montreal       77 82  3   1 2 0 0

% more teams5.txt
5
New_York     75 59 28    0 3 8 7 3
Baltimore    71 63 28    3 0 2 7 4
Boston       69 66 27    8 2 0 0 0
Toronto      63 72 27    7 7 0 0 0
Detroit      49 86 27    3 4 0 0 0
```

You may assume that $N \geq 1$ and that the input files are in the specified format and internally consistent. Note that a team's total number of remaining games does not necessarily equal the number of remaining games against divisional rivals since teams may play opponents outside of their own division.

Output format. Use the following `main()` function, which reads in a sports division from an input file and prints out whether each team is mathematically eliminated and a certificate of elimination for each team that is eliminated:

```
public static void main(String[] args) {
    BaseballElimination division = new
BaseballElimination(args[0]);
    for (String team : division.teams()) {
        if (division.isEliminated(team)) {
            StdOut.print(team + " is eliminated by the subset R
= { ");
            for (String t :
division.certificateOfElimination(team))
                StdOut.print(t + " ");
```

```
            StdOut.println("}");
        }
        else {
            StdOut.println(team + " is not eliminated");
        }
    }
}
```

Below is the desired output:

```
% java BaseballElimination teams4.txt
Atlanta is not eliminated
Philadelphia is eliminated by the subset R = { Atlanta New_York
}
New_York is not eliminated
Montreal is eliminated by the subset R = { Atlanta }

% java BaseballElimination teams5.txt
New_York is not eliminated
Baltimore is not eliminated
Boston is not eliminated
Toronto is not eliminated
Detroit is eliminated by the subset R = { New_York Baltimore
Boston Toronto }
```

Analysis. Analyze the worst-case memory usage and running time of your algorithm.

- What is the order of growth of the amount of memory (in the worst case) that your program uses to determine whether *one* team is eliminated? In particular, how many vertices and edges are in the flow network as a function of the number of teams N?
- What is the order of growth of the running time (in the worst case) of your program to determine whether *one* team is eliminated as a function of the number of teams N? In your calculation, assume that the order of growth of the running time (in the worst case) to compute a maxflow in a network with V vertices and E edges is $V E^2$.

Also, use the output of your program to answer the following question:

- Consider the sports division defined in teams12.txt. Explain in nontechnical terms (using the results of certificate of elimination and grade-school arithmetic) why Japan is mathematically eliminated.

Extra credit. Create and submit an interesting test input file (in the specified format) and name it `teams.txt`. Your input file should contain one or more teams whose elimination would not be obvious to a sports writer. Ideally, your input file should be based on real-world data.

Submission. Submit `BaseballElimination.java` and any other files needed to compile your program (excluding those in `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`.

Programming Assignment 4: Boggle

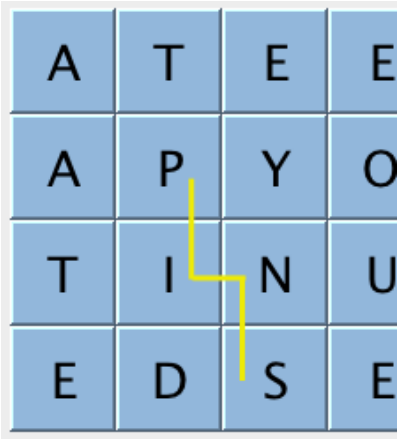
Warning: this programming assignment is new (Fall 2013): be on the lookout for any bugs, inconsistencies, or ambiguities in this assignment specification, autograder, or accompanying data files. Please report any problems in the discussion forums and we will do our best to fix.

Write a program to play the word game Boggle®.

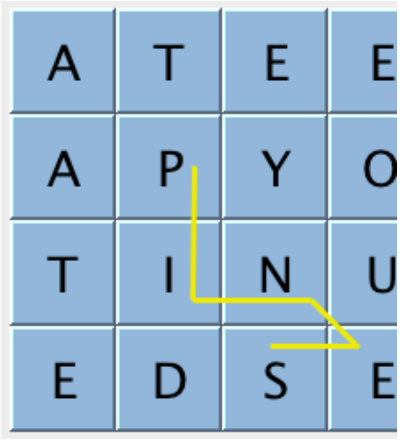
The Boggle game. Boggle is a word game designed by Allan Turoff and distributed by Hasbro. It involves a board made up of 16 cubic dice, where each die has a letter printed on each of its sides. At the beginning of the game, the 16 dice are shaken and randomly distributed into a 4-by-4 tray, with only the top sides of the dice visible. The players compete to accumulate points by building *valid* words out of the dice according to the following rules:

- A valid word must be composed by following a sequence of *adjacent dice*—two dice are adjacent if they are horizontal, vertical, or diagonal neighbors.
- A valid word can use each die at most once.
- A valid word must contain at least 3 letters.
- A valid word must be in the dictionary (which typically does not contain proper nouns).

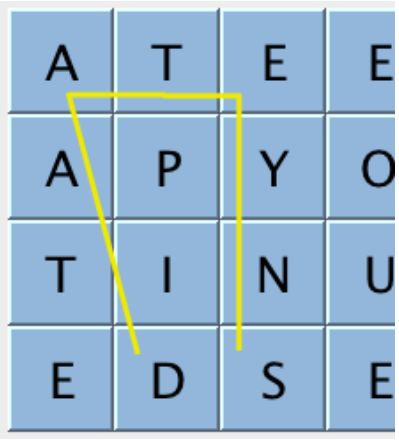
Here are some examples of valid and invalid words:



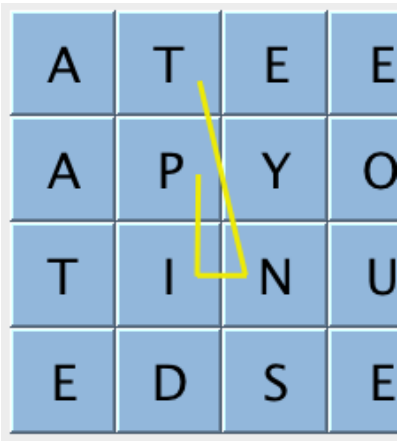
PINS
(*valid*)



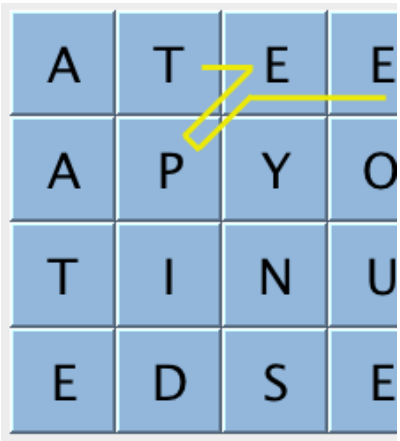
PINES
(*valid*)



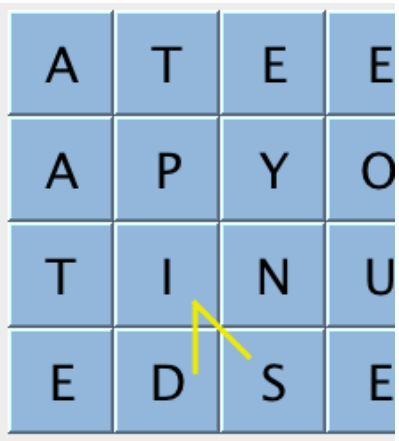
DATES
(*invalid—dice not adjacent*)



PINT
(*invalid—path not sequential*)



TEPEE
(*invalid—die used more than*



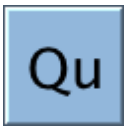
SID
(*invalid—word not in*

once) *dictionary*)

Scoring. Words are scored according to their length, using this table:

word length	*points*
0–2	0
3–4	1
5	2
6	3
7	5
8+	11

The *Qu* special case. In the English language, the letter `Q` is almost always followed by the letter `U`. Consequently, the side of one die is printed with the two-letter sequence `Qu` instead of `Q` (and this two-letter sequence must be used together when forming words). When scoring, `Qu` counts as two letters; for example, the word `QuEUE` scores as a 5-letter word even though it is formed by following a sequence of 4 dice.



Your task. Your challenge is to write a Boggle solver that finds *all* valid words in a given Boggle board, using a given dictionary. Implement an immutable data type `BoggleSolver` with the following API:

```
public class BoggleSolver
{
    // Initializes the data structure using the given array of
strings as the dictionary.
    // (You can assume each word in the dictionary contains only
the uppercase letters A through Z.)
    public BoggleSolver(String[] dictionary)

    // Returns the set of all valid words in the given Boggle
board, as an Iterable.
    public Iterable<String> getAllValidWords(BoggleBoard board)

    // Returns the score of the given word if it is in the
dictionary, zero otherwise.
    // (You can assume the word contains only the uppercase
letters A through Z.)
    public int scoreOf(String word)
}
```

It is up to you how you search for and store the words contained in the board, as well as the dictionary used to check them. If you need help getting started, some hints are available on the checklist.

The board data type. We provide an immutable data type BoggleBoard.java for representing Boggle boards. It includes constructors for creating Boggle boards from either the 16 Hasbro dice, the distribution of letters in the English language, a file, or a character array; methods for accessing the individual letters; and a method to print out the board for debugging. Here is the full API:

```
public class BoggleBoard
{
    // Initializes a random 4-by-4 Boggle board.
    // (by rolling the Hasbro dice)
    public BoggleBoard()

    // Initializes a random M-by-N Boggle board.
    // (using the frequency of letters in the English language)
    public BoggleBoard(int M, int N)

    // Initializes a Boggle board from the specified filename.
    public BoggleBoard(String filename)

    // Initializes a Boggle board from the 2d char array.
    // (with 'Q' representing the two-letter sequence "Qu")
    public BoggleBoard(char[][] a)

    // Returns the number of rows.
    public int rows()

    // Returns the number of columns.
    public int cols()

    // Returns the letter in row i and column j.
    // (with 'Q' representing the two-letter sequence "Qu")
    public char getLetter(int i, int j)

    // Returns a string representation of the board.
    public String toString()
}
```

Testing. We provide a number of dictionary and board files for testing.

- *Dictionaries.* A dictionary consists of a sequence of words, separated by whitespace, in alphabetical order. You can assume that each word contains only the uppercase letters `A` through `Z`. For example, here are the two files dictionary-algs4.txt anddictionary-yawl.txt:
    -

- % **more dictionary-algs4.txt** % **more dictionary-yawl.txt**
- ABACUS AA
- ABANDON AAH
- ABANDONED AAHED
- ABBREVIATE AAHING
- ... ...
- QUEUE
  PNEUMONOULTRAMICROSCOPICSILICOVOLCANOCONIOSIS
- ... ...
- ZOOLOGY ZYZZYVAS

The former is a list of 6,013 words that appear in *Algorithms 4/e*; the latter is a comprehensive list of 264,061 English words (known as *Yet Another Word List*) that is widely used in word-game competitions.

- *Boggle boards.* A boggle board consists of two integers *M* and *N*, followed by the *M* × *N* characters in the board, with the integers and characters separated by whitespace. You can assume the integers are nonnegative and that the characters are uppercase letters `A` through `Z` (with the two-letter sequence `Qu` represented as either `Q` or `Qu`). For example, here are the files board4x4.txt and board-q.txt:
  - % **more board4x4.txt** % **more board-q.txt**
  - 4 4 4 4
  - A T E E S N R T
  - A P Y O O I E L
  - T I N U E Qu T T
  - E D S E R S A T

The following test client takes the filename of a dictionary and the filename of a Boggle board as command-line arguments and prints out all valid words for the given board using the given dictionary.

```
public static void main(String[] args)
{
    In in = new In(args[0]);
    String[] dictionary = in.readAllStrings();
    BoggleSolver solver = new BoggleSolver(dictionary);
    BoggleBoard board = new BoggleBoard(args[1]);
    int score = 0;
    for (String word : solver.getAllValidWords(board))
    {
        StdOut.println(word);
        score += solver.scoreOf(word);
    }
    StdOut.println("Score = " + score);
}
```

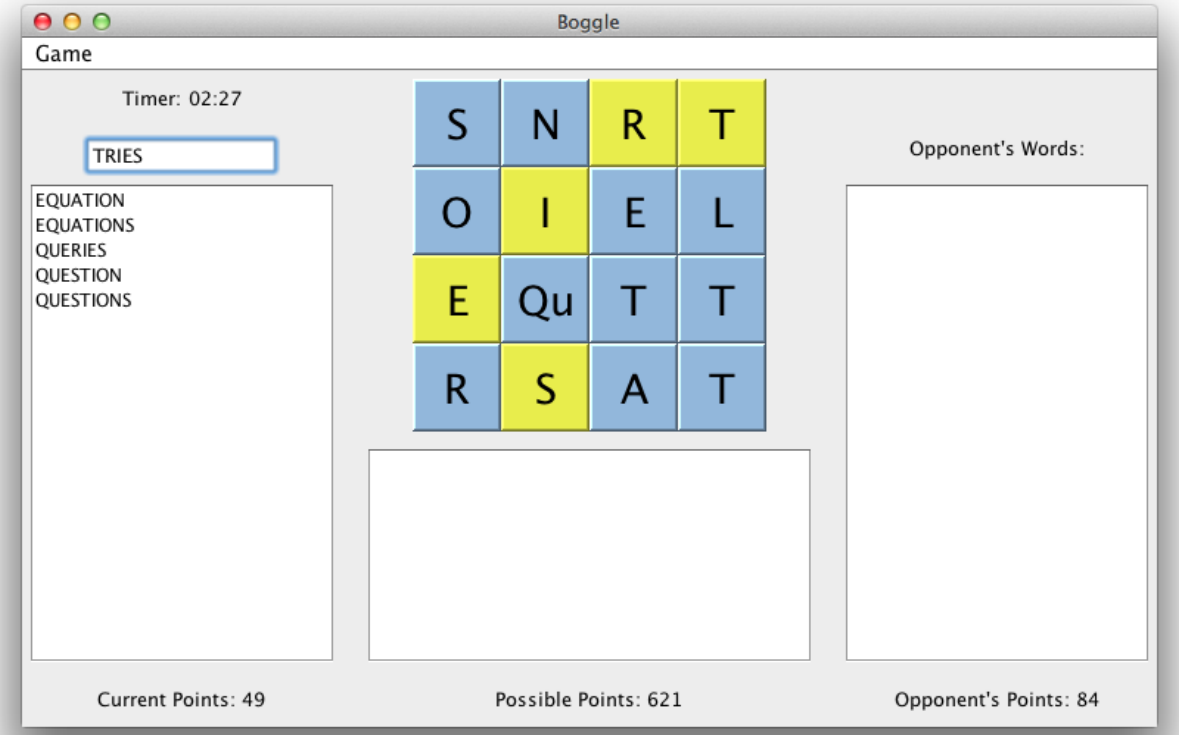
Here are two sample executions:

```
% java BoggleSolver dictionary-algs4.txt board4x4.txt
AID
EQUATION
DIE
EQUATIONS
END
ENDS
QUERIES
...
QUESTION
YOU
QUESTIONS
Score = 33
```

```
% java BoggleSolver dictionary-algs4.txt board-q.txt
...
...
TRIES
Score
= 84
```

Performance. If you choose your data structures and algorithms judiciously, your program can preprocess the dictionary and find all valid words in a random Hasbro board (or even a random 10-by-10 board) in a fraction of a second. To stress test the performance of your implementation, create one `BoggleSolver` object (from a given dictionary); then, repeatedly generate and solve random Hasbro boards. How many random Hasbro boards can you solve per second? *For full credit, your program must be able to solve thousands of random Hasbro boards per second.* The goal on this assignment is raw speed—for example, it's fine to use 10× more memory if the program is 10× faster.

Interactive game (optional, but fun and no extra work). Once you have a working version of `BoggleSolver.java`, download, compile, and run BoggleGame.java to play Boggle against a computer opponent. To enter a word, either type it in the text box or click the corresponding sequence of dice on the board. The computer opponent has various levels of difficulty, ranging from finding only words from popular nursery rhymes (easy) to words that appear in *Algorithms 4/e* (medium) to finding every valid word (humbling).



Challenge for the bored. Here are some challenges:

- Find a maximum scoring 4-by-4 Hasbro board.
- Find a maximum scoring 4-by-4 Hasbro board using the Zinga list of 584,983 Italian words.
- Find a minimum scoring 4-by-4 Hasbro board.
- Find a maximum scoring 5-by-5 Deluxe Boggle board.
- Find a maximum scoring *N*-by-*N* board (not necessarily using the Hasbro dice) for different values of *N*.
- Find a board with the most words (or the most words that are 8 letters or longer).
- Find a 4-by-4 Hasbro board that scores *exactly* 2,500, 3,000, 3,500, or 4,000 points.
- Design an algorithm to determine whether a given 4-by-4 board can be generated by rolling the 16 Hasbro dice.
- How many words in the dictionary appear in no 4-by-4 Hasbro boards?
- Add new features to BoggleGame.java.
- Extend your program to handle arbitrary Unicode letters and dictionaries. You may need to consider alternate algorithms and data structures.
- Extend your program to handle arbitrary strings on the faces of the dice, generalizing your hack for dealing with the two-letter sequence `Qu`.

Unless otherwise stated, use the dictionary-yawl.txt dictionary. If you discover interesting boards, you are encouraged to share and describe them in the Discussion Forums.

Deliverables. Submit `BoggleSolver.java` and any other accompanying files (excluding those in `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`.

Programming Assignment 5: Burrows-Wheeler Data Compression

Implement the Burrows-Wheeler data compression algorithm. This revolutionary algorithm outcompresses gzip and PKZIP, is relatively easy to implement, and is not protected by any patents. It forms the basis of the Unix compression utililty bzip2.

The Burrows-Wheeler compression algorithm consists of three algorithmic components, which are applied in succession:

1. *Burrows-Wheeler transform.* Given a typical English text file, transform it into a text file in which sequences of the same character occur near each other many times.
2. *Move-to-front encoding.* Given a text file in which sequences of the same character occur near each other many times, convert it into a text file in which certain characters appear more frequently than others.
3. *Huffman compression.* Given a text file in which certain characters appear more frequently than others, compress it by encoding freqently occuring characters with short codewords and rare ones with long codewords.

The final step is the one that compresses the message: it is particularly effective because the first two steps result in a text file in which certain characters appear much more frequently than others. To expand a message, apply the inverse operations in reverse order: first apply the Huffman expansion, then the move-to-front decoding, and finally the inverse Burrows-Wheeler transform. Your task is to implement Burrows-Wheeler and move-to-front components efficiently.

Binary input and binary output. To enable that your programs work with binary data, you will use the libraries BinaryStdIn.java and BinaryStdOut.java described in *Algorithms, 4th edition*. To display the binary output when debugging, you can useHexDump.java, which takes a command-line argument *N*, reads bytes from standard input and writes them to standard output in hexadecimal, *N* per line.

```
% more abra.txt
ABRACADABRA!

% java HexDump 16 < abra.txt
41 42 52 41 43 41 44 41 42 52 41 21
96 bits
```

Note that in ASCII, `'A'` is 41 (hex) and `'!'` is 21 (hex).

Huffman encoding and decoding. Huffman.java (Program 5.10 in *Algorithms, 4th edition*) implements the classic Huffman compression and expansion algorithms.

```
% java Huffman - < abra.txt | java HexDump 16
50 4a 22 43 43 54 a8 40 00 00 01 8f 96 8f 94
120 bits

% java Huffman - < abra.txt | java Huffman +
ABRACADABRA!
```

You will not write any code for this step.

Move-to-front encoding and decoding. The main idea of *move-to-front* encoding is to maintain an ordered sequence of all of the characters in the alphabet, and repeatedly read in a character

from the input message, print out the position in which that character appears, and move that character to the front of the sequence. As a simple example, if the initial ordering over a 6-character alphabet is `A B C D E F`, and we want to encode the input `CAAABCCCACCF`, then we would update the move-to-front sequences as follows:

```
move-to-front    in   out
-------------    ---  ---
 A B C D E F      C    2
 C A B D E F      A    1
 A C B D E F      A    0
 A C B D E F      A    0
 A C B D E F      B    2
 B A C D E F      C    2
 C B A D E F      C    0
 C B A D E F      C    0
 C B A D E F      A    2
 A C B D E F      C    1
 C A B D E F      C    0
 C A B D E F      F    5
 F C A B D E
```

If the same character occurs next to each other many times in the input, then many of the output values will be small integers, such as 0, 1, and 2. The extremely high frequency of certain characters makes an ideal scenario for Huffman coding.

- *Move-to-front encoding.* Your task is to maintain an ordered sequence of the 256 extended ASCII characters. Initialize the sequence by making the *i*th character in the sequence equal to the *i*th extended ASCII character. Now, read in each 8-bit character `c` from standard input one at a time, output the 8-bit index in the sequence where `c` appears, and move `c` to the front.
- `% java MoveToFront - < abra.txt | java HexDump 16`
- `41 42 52 02 44 01 45 01 04 04 02 26`
- `96 bits`
- *Move-to-front decoding.* Initialize an ordered sequence of 256 characters, where extended ASCII character *i* appears *i*th in the sequence. Now, read in each 8-bit character *i* (but treat it as an integer between 0 and 255) from standard input one at a time, write the *i*th character in the sequence, and move that character to the front. Check that the decoder recovers any encoded message.
- `% java MoveToFront - < abra.txt | java MoveToFront +`
- `ABRACADABRA!`

Name your program `MoveToFront.java` and organize it using the following API:

```
public class MoveToFront {
    // apply move-to-front encoding, reading from standard input
and writing to standard output
    public static void encode()

    // apply move-to-front decoding, reading from standard input
and writing to standard output
    public static void decode()

    // if args[0] is '-', apply move-to-front encoding
    // if args[0] is '+', apply move-to-front decoding
    public static void main(String[] args)
```

```
}
```

The running time of move-to-front encoding and decoding should be proportional to $R\,N$ in the worst case and proportional to N in practice on inputs that arise when compressing typical English text, where N is the number of characters in the input and R is the alphabet size.

Circular suffix array. To efficiently implement the key component in the Burrows-Wheeler transform, you will use a fundamental data structure known as the *circular suffix array*, which describes the abstraction of a sorted array of the N circular suffixes of a string of length N. As an example, consider the string "ABRACADABRA!" of length 12. The table below shows its 12 circular suffixes and the result of sorting them.

```
 i        Original Suffixes           Sorted Suffixes
index[i]
--    -----------------------     -----------------------    ---
-----
 0    A B R A C A D A B R A !     ! A B R A C A D A B R A    11
 1    B R A C A D A B R A ! A     A ! A B R A C A D A B R    10
 2    R A C A D A B R A ! A B     A B R A ! A B R A C A D    7
 3    A C A D A B R A ! A B R     A B R A C A D A B R A !    0
 4    C A D A B R A ! A B R A     A C A D A B R A ! A B R    3
 5    A D A B R A ! A B R A C     A D A B R A ! A B R A C    5
 6    D A B R A ! A B R A C A     B R A ! A B R A C A D A    8
 7    A B R A ! A B R A C A D     B R A C A D A B R A ! A    1
 8    B R A ! A B R A C A D A     C A D A B R A ! A B R A    4
 9    R A ! A B R A C A D A B     D A B R A ! A B R A C A    6
10    A ! A B R A C A D A B R     R A ! A B R A C A D A B    9
11    ! A B R A C A D A B R A     R A C A D A B R A ! A B    2
```

We define `index[i]` to be the index of the original suffix that appears *ith* in the sorted array. For example, `index[11] = 2` means that the 2*nd* original suffix appears 11*th* in the sorted order (i.e., last alphabetically).

Your job is to implement the following circular suffix array API, which provides the client access to the `index[]` values:

```
public class CircularSuffixArray {
    public CircularSuffixArray(String s)  // circular suffix
array of s
    public int length()                   // length of s
    public int index(int i)               // returns index of
ith sorted suffix
}
```

Your data type must use linear space; the methods `length()` and `index()` must take constant time. Prior to Java 7, Update 6, the array of circular suffixes could be constructed in linear space using the `substring()` method. It now takes quadratic space—*do not explicitly generate the N suffixes*. Instead, precompute and store the `index[]` array, which requires only linear space.

Burrows-Wheeler transform. The goal of the Burrows-Wheeler transform is not to compress a message, but rather to transform it into a form that is more amenable to compression. The transform rearranges the characters in the input so that there are lots of clusters with repeated characters, but in such a way that it is still possible to recover the original input. It relies on the

following intuition: if you see the letters `hen` in English text, then most of the time the letter preceding it is `t` or `w`. If you could somehow group all such preceding letters together (mostly `t`'s and some `w`'s), then you would have an easy opportunity for data compression.

- *Burrows-Wheeler encoding.* The Burrows-Wheeler transform of a string *s* of length *N* is defined as follows: Consider the result of sorting the *N* circular suffixes of *s*. The Burrows-Wheeler transform is the last column in the sorted suffixes array `t[]`, preceded by the row number `first` in which the original string ends up. Continuing with the "`ABRACADABRA!`" example above, we highlight the two components of the Burrows-Wheeler transform in the table below.

```
 i     Original Suffixes          Sorted Suffixes       t
index[i]
--    -----------------------    -----------------------
--------
 0    A B R A C A D A B R A !    ! A B R A C A D A B R A
11
 1    B R A C A D A B R A ! A    A ! A B R A C A D A B R
10
 2    R A C A D A B R A ! A B    A B R A ! A B R A C A D
7
*3    A C A D A B R A ! A B R    A B R A C A D A B R A !
*0
 4    C A D A B R A ! A B R A    A C A D A B R A ! A B R
3
 5    A D A B R A ! A B R A C    A D A B R A ! A B R A C
5
 6    D A B R A ! A B R A C A    B R A ! A B R A C A D A
8
 7    A B R A ! A B R A C A D    B R A C A D A B R A ! A
1
 8    B R A ! A B R A C A D A    C A D A B R A ! A B R A
4
 9    R A ! A B R A C A D A B    D A B R A ! A B R A C A
6
10    A ! A B R A C A D A B R    R A ! A B R A C A D A B
9
11    ! A B R A C A D A B R A    R A C A D A B R A ! A B
2
```

  Since the original string `ABRACADABRA!` ends up in row 3, we have `first` = `3`. Thus, the Burrows-Wheeler transform is

```
3
ARD!RCAAAABB
```

  Notice how there are 4 consecutive `A`s and 2 consecutive `B`s—these clusters make the message easier to compress.

```
% java BurrowsWheeler - < abra.txt | java HexDump 16
00 00 00 03 41 52 44 21 52 43 41 41 41 41 42 42
128 bits
```

Also, note that the integer 3 is represented using 4 bytes (`00 00 00 03`). The character `'A'` is represented by hex `41`, the character `'R'` by `52`, and so forth.

- *Burrows-Wheeler decoder.* Now, we describe how to invert the Burrows-Wheeler transform and recover the original input string. If the *jth* original suffix (original string, shifted *j* characters to the left) is the *ith* row in the sorted order, we define `next[i]` to be the row in the sorted order where the (*j* + 1)*st* original suffix appears. For example, if `first` is the row in which the original input string appears, then `next[first]` is the row in the sorted order where the 1*st* original suffix (the original string left-shifted by 1) appears; `next[next[first]]` is the row in the sorted order where the 2*nd* original suffix appears; `next[next[next[first]]]` is the row where the 3*rd* original suffix appears; and so forth.
    - *Decoding the message given t[], first, and the next[] array.* The input to the Burrows-Wheeler decoder is the last column `t[]` of the sorted suffixes along with `first`. From `t[]`, we can deduce the first column of the sorted suffixes because it consists of precisely the same characters, but in sorted order.

```
  i       Sorted Suffixes      t       next
--    ----------------------      ----
 0    ! ? ? ? ? ? ? ? ? ? A         3
 1    A ? ? ? ? ? ? ? ? ? R         0
 2    A ? ? ? ? ? ? ? ? ? D         6
*3    A ? ? ? ? ? ? ? ? ? !         7
 4    A ? ? ? ? ? ? ? ? ? R         8
 5    A ? ? ? ? ? ? ? ? ? C         9
 6    B ? ? ? ? ? ? ? ? ? A        10
 7    B ? ? ? ? ? ? ? ? ? A        11
 8    C ? ? ? ? ? ? ? ? ? A         5
 9    D ? ? ? ? ? ? ? ? ? A         2
10    R ? ? ? ? ? ? ? ? ? B         1
11    R ? ? ? ? ? ? ? ? ? B         4
```

      Now, given the `next[]` array and `first`, we can reconstruct the original input string because the first character of the *i*th original suffix is the *i*th character in the input string. In the example above, since `first` = 3, we know that the original input string appears in row 3; thus, the original input string starts with `'A'` (and ends with `'!'`). Since `next[first]` = 7, the next original suffix appears in row 7; thus, the next character in the original input string is `'B'`.
      Since `next[next[first]]` =11, the next original suffix appears in row 11; thus, the next character in the original input string is `'R'`.

    - *Constructing the next[] array from t[] and first.* Amazingly, the information contained in the Burrows-Wheeler transform suffices to reconstruct the `next[]` array, and, hence, the original message! Here's how. It is easy to deduce a `next[]` value for a character that appears exactly once in the input string. For example, consider the suffix that starts with `'C'`. By inspecting the first column, it appears 8*th* in the sorted order. The next original suffix after this one will have the character `'C'` as its last character. By inspecting the last column, the next original appears 5*th* in the sorted order. Thus, `next[8]` = 5. Similarly, `'D'` and `'!'` each occur only once, so we can deduce that `next[9]` = 2 and `next[0]` = 3.

```
  i       Sorted Suffixes      t       next
```

```
o  --    ----------------------      ----
o   0    ! ? ? ? ? ? ? ? ? ? A         3
o   1    A ? ? ? ? ? ? ? ? ? R
o   2    A ? ? ? ? ? ? ? ? ? D
o  *3    A ? ? ? ? ? ? ? ? ? !
o   4    A ? ? ? ? ? ? ? ? ? R
o   5    A ? ? ? ? ? ? ? ? ? C
o   6    B ? ? ? ? ? ? ? ? ? A
o   7    B ? ? ? ? ? ? ? ? ? A
o   8    C ? ? ? ? ? ? ? ? ? A         5
o   9    D ? ? ? ? ? ? ? ? ? A         2
o  10    R ? ? ? ? ? ? ? ? ? B
o  11    R ? ? ? ? ? ? ? ? ? B
```

However, since `'R'` appears twice, it may seem ambiguous whether `next[10] = 1` and `next[11] = 4`, or whether `next[10] = 4` and `next[11] = 1`. Here's the key rule that resolves the apparent ambiguity:

If sorted row i and j both start with the same character and i < j, then next[i] < next[j].

This rule implies `next[10] = 1` and `next[11] = 4`. Why is this rule valid? The rows are sorted so row 10 is lexicographically less than row 11. Thus the ten unknown characters in row 10 must be less than the ten unknown characters in row 11 (since both start with `'R'`). We also know that between the two rows that end with `'R'`, row 1 is less than row 4. But, the ten unknown characters in row 10 and 11 are precisely the first ten characters in rows 1 and 4.
Thus, `next[10] = 1` and`next[11] = 4` or this would contradict the fact that the suffixes are sorted.

- Check that the decoder recovers any encoded message.
  - `% **java BurrowsWheeler - < abra.txt | java BurrowsWheeler +**`
  - `ABRACADABRA!`

Name your program `BurrowsWheeler.java` and organize it using the following API:

```
public class BurrowsWheeler {
    // apply Burrows-Wheeler encoding, reading from standard
input and writing to standard output
    public static void encode()

    // apply Burrows-Wheeler decoding, reading from standard
input and writing to standard output
    public static void decode()

    // if args[0] is '-', apply Burrows-Wheeler encoding
    // if args[0] is '+', apply Burrows-Wheeler decoding
    public static void main(String[] args)
}
```

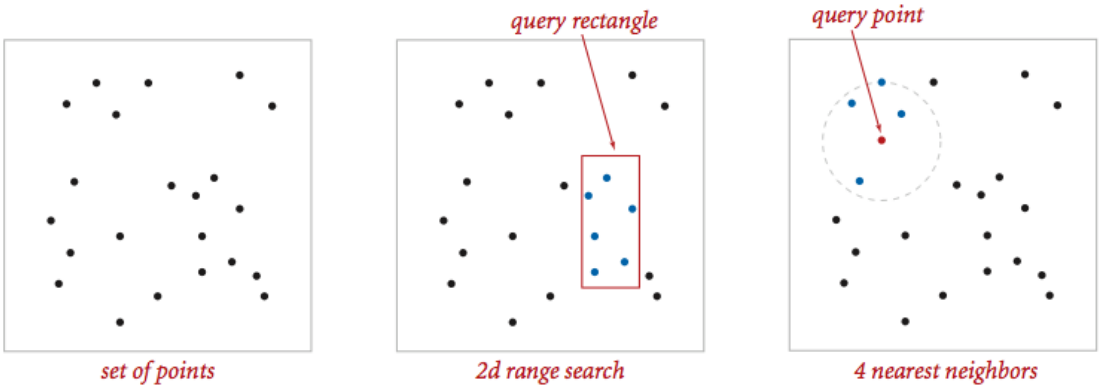
The running time of your Burrows-Wheeler encoder should be proportional to $N + R$ in the worst case, excluding the time to construct the circular suffix array. The running time of your Burrows-Wheeler decoder should be proportional to $N + R$ in the worst case.

Analysis (optional). Once you have `MoveToFront.java` and `BurrowsWheeler.java` working, compress some of these text files; then, test it on some binary files. Calculate the compression ratio achieved for each file and report the time to compress and expand each file. (Here, compression and expansion consists of applying `BurrowsWheeler`, `MoveToFront`, and `Huffman` in succession.) Finally, determine the order of growth of the running time of each of your encoders and decoders, both in the worst case and on typical Englist text inputs.

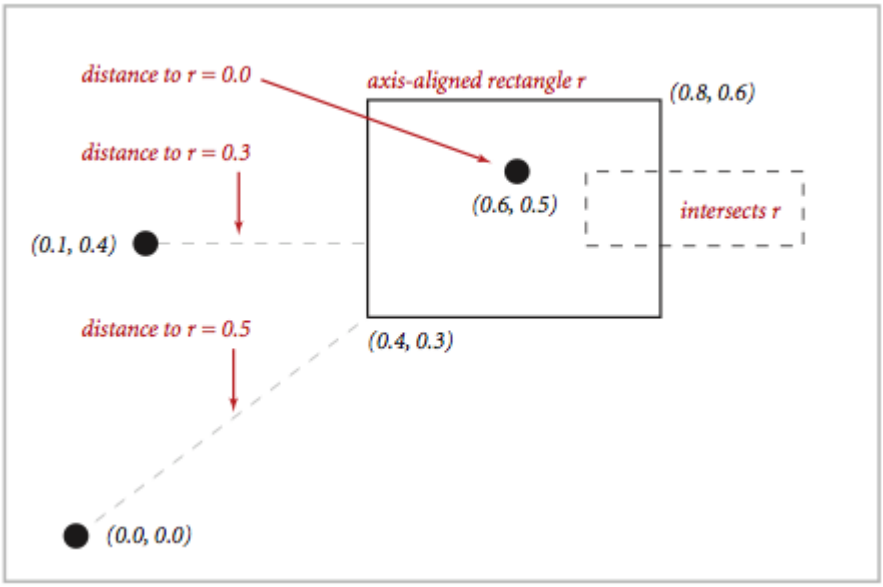
Deliverables. Submit `MoveToFront.java`, `BurrowsWheeler.java`, and `CircularSuffixArray.java` along with any other helper files needed to run your program (excluding those in `stdlib.jar` and `algs4.jar`). Your may not call any library functions other than those in `java.lang`, `java.util`, `stdlib.jar`, and `algs4.jar`.

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.

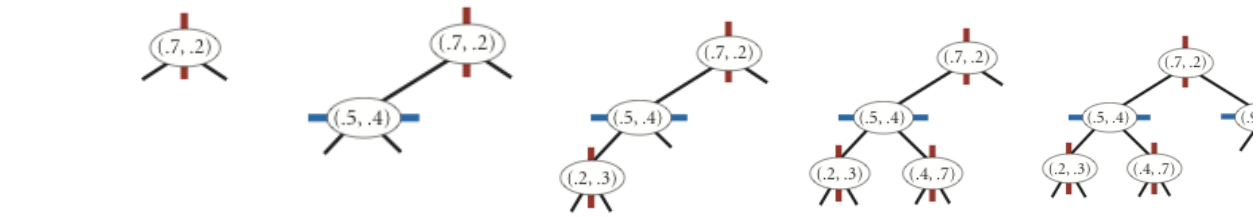
Brute-force implementation. Write a mutable data type PointSET.java that represents a set of points in the unit square. Implement the following API by using a red-black BST (using either SET from algs4.jar or java.util.TreeSet).

```
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
}
```

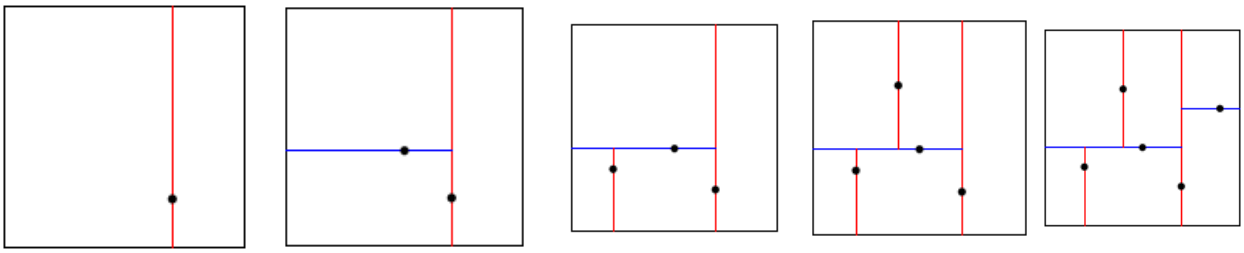
Your implementation should support insert() and contains() in time proportional to the logarithm of the number of points in the set in the worst case; it should support nearest() and range() in time proportional to the number of points in the set.

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.