Exercises

Exercise 1. (*Great Circle Distance*) Write a program called `GreatCircle.java` that accepts x_1 (double), y_1 (double), x_2 (double), and y_2 (double) as command-line arguments representing the latitude and longitude (in degrees) of two points on earth, and writes to standard output the great-circle distance (in km) between the two points, given by the formula

$$d = 6359.83 \arccos(\sin(x_1)\sin(x_2) + \cos(x_1)\cos(x_2)\cos(y_1 - y_2)).$$

```
>_ ~/workspace/project1
$ java GreatCircle 48.87 -2.33 37.8 -122.4
8701.387455462233
```

GreatCircle.java

```
import stdlib.StdOut;

public class GreatCircle {
    // Entry point.
    public static void main(String[] args) {
        // Accept x1 (double), y1 (double), x2 (double), and y2 (double) as command-line arguments.
        ...

        // Convert the angles to radians.
        ...

        // Calculate great-circle distance d.
        ...

        // Write d to standard output.
        ...
    }
}
```

Exercise 2. (*Counting Primes*) Implement the static method `isPrime()` in `PrimeCounter.java` that accepts an integer x and returns `true` if x is prime and `false` otherwise. Also implement the static method `primes()` that accepts an integer n and returns the number of primes less than or equal to n — a number x is prime if it is not divisible by any number $i \in [2, \sqrt{x}]$.

```
>_ ~/workspace/project1
$ java PrimeCounter 1000
168
```

PrimeCounter.java

```
import stdlib.StdOut;

public class PrimeCounter {
    // Entry point. [DO NOT EDIT]
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        StdOut.println(primes(n));
    }

    // Returns true if x is prime; and false otherwise.
    private static boolean isPrime(int x) {
        // For each 2 <= i <= x / i, if x is divisible by i, then x is not a prime. If no such i
        // exists, then x is a prime.
        ...
    }

    // Returns the number of primes <= n.
    private static int primes(int n) {
        // For each 2 <= i <= n, use isPrime() to test if i is prime, and if so increment a count.
        // At the end return the count.
        ...
    }
}
```

Exercise 3. (*Euclidean Distance*) Implement the static method `distance()` in `Distance.java` that accepts position vectors x and y — each represented as a 1D array of doubles — and returns the Euclidean distance between the two vectors, calculated as the square root of the sums of the squares of the differences between the corresponding entries.

>_ ~/workspace/project1

```
$ java Distance
5
-9 1 10 -1 1
5
-5 9 6 7 4
13.0
```

Distance.java

```
import stdlib.StdArrayIO;
import stdlib.StdOut;

public class Distance {
    // Entry point. [DO NOT EDIT]
    public static void main(String[] args) {
        double[] x = StdArrayIO.readDouble1D();
        double[] y = StdArrayIO.readDouble1D();
        StdOut.println(distance(x, y));
    }

    // Returns the Euclidean distance between the position vectors x and y.
    private static double distance(double[] x, double[] y) {
        // Sum up the squares of (x[i] - y[i]), where 0 <= i < x.length, and return the square
        // root of the sum.
        ...
    }
}
```

Exercise 4. (*Matrix Transpose*) Implement the static method `tranpose()` in `Transpose.java` that accepts a matrix x — represented as a 2D array of doubles — and returns a new matrix that is the transpose of x.

>_ ~/workspace/project1

```
$ Transpose
3 3
1 2 3
4 5 6
7 8 9
3 3
  1.00000   4.00000   7.00000
  2.00000   5.00000   8.00000
  3.00000   6.00000   9.00000
```

Distance.java

```
import stdlib.StdArrayIO;

public class Transpose {
    // Entry point. [DO NOT EDIT]
    public static void main(String[] args) {
        double[][] x = StdArrayIO.readDouble2D();
        StdArrayIO.print(transpose(x));
    }

    // Returns a new matrix that is the transpose of x.
    private static double[][] transpose(double[][] x) {
        // Create a new 2D matrix t (for transpose) with dimensions n x m, where m x n are the
        // dimensions of x.
        ...

        // For each 0 <= i < m and 0 <= j < n, set t[j][i] to x[i][j].
        ...

        // Return t.
        ...
    }
}
```

Exercise 5. (*Rational Number*) Implement an immutable data type called `Rational` that represents a rational number, ie, a number of the form a/b where a and $b \neq 0$ are integers. The data type must support the following API:

Rational	
`Rational(long x)`	constructs a rational number whose numerator is `x` and denominator is 1
`Rational(long x, long y)`	constructs a rational number given its numerator `x` and denominator `y` (†)
`Rational add(Rational other)`	returns the sum of this rational number and `other`
`Rational multiply(Rational other)`	returns the product of this rational number and `other`
`boolean equals(Object other)`	returns `true` if this rational number is equal to `other`, and false otherwise
`String toString()`	returns a string representation of this rational number

† Use the private method `gcd()` to ensure that the numerator and denominator never have any common factors. For example, the rational number 2/4 must be represented as 1/2.

>_ ~/workspace/project1

```
$ java Rational 10
a            = 1 + 1/2 + 1/4 + ... + 1/2^10 = 1023/512
b            = (2^10 - 1) / 2^(10 - 1) = 1023/512
a.equals(b) = true
```

Rational.java

```
import stdlib.StdOut;

public class Rational {
    private long x; // numerator
    private long y; // denominator

    // Constructs a rational number whose numerator is x and denominator is 1.
    public Rational(long x) {
        // Set this.x to x and this.y to 1.
        ...
    }

    // Constructs a rational number given its numerator x and denominator y.
    public Rational(long x, long y) {
        // Set this.x to x / gcd(x, y) and this.y to y / gcd(x, y).
        ...
    }

    // Returns the sum of this rational number and other.
    public Rational add(Rational other) {
        // Sum of rationals a/b and c/d is the rational (ad + bc) / bd.
        ...
    }

    // Returns the product of this rational number and other.
    public Rational multiply(Rational other) {
        // Product of rationals a/b and c/d is the rational ac / bd.
        ...
    }

    // Returns true if this rational number is equal to other, and false otherwise.
    public boolean equals(Object other) {
        if (other == null) {
            return false;
        }
        if (other == this) {
            return true;
        }
        if (other.getClass() != this.getClass()) {
            return false;
        }

        // Rationals a/b and c/d are equal iff a == c and b == d.
        ...
    }

    // Returns a string representation of this rational number.
    public String toString() {
        long a = x, b = y;
        if (a == 0 || b == 1) {
            return a + "";
```

```
        }
        if (b < 0) {
            a *= -1;
            b *= -1;
        }
        return a + "/" + b;
    }

    // Returns gcd(p, q), computed using Euclid's algorithm.
    private static long gcd(long p, long q) {
        return q == 0 ? p : gcd(q, p % q);
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        Rational total = new Rational(0);
        Rational term = new Rational(1);
        for (int i = 1; i <= n; i++) {
            total = total.add(term);
            term = term.multiply(new Rational(1, 2));
        }
        Rational expected = new Rational((long) Math.pow(2, n) - 1, (long) Math.pow(2, n - 1));
        StdOut.printf("a           = 1 + 1/2 + 1/4 + ... + 1/2^%d = %s\n", n, total);
        StdOut.printf("b           = (2^%d - 1) / 2^(%d - 1) = %s\n", n, n, expected);
        StdOut.printf("a.equals(b) = %b\n", total.equals(expected));
    }
}
```

Exercise 6. (*Harmonic Number*) Write a program called `Harmonic.java` that accepts n (int) as command-line argument, computes the nth harmonic number H_n as a rational number, and writes the value to standard output.

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} + \frac{1}{n}.$$

```
>_ ~/workspace/project1
$ java Harmonic 5
137/60
```

Harmonic.java

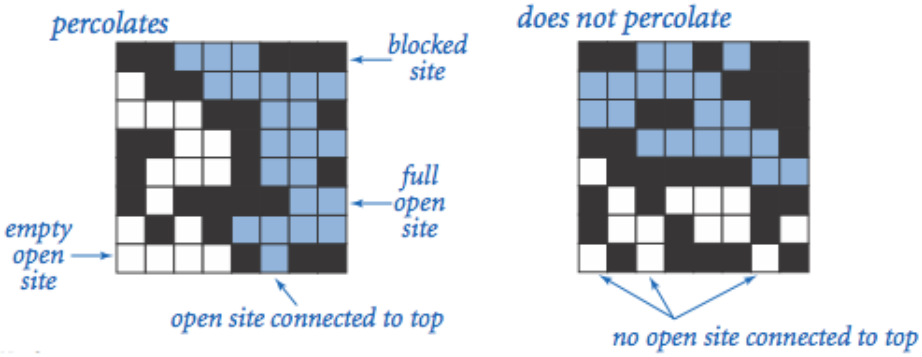
```
import stdlib.StdOut;

public class Harmonic {
    // Entry point.
    public static void main(String[] args) {
        // Accept n (int) as command-line argument.
        ...

        // Set total to the rational number 0.
        ...

        // For each 1 <= i <= n, add the rational term 1 / i to total.
        ...

        // Write total to standard output.
        ...
    }
}
```

Problems

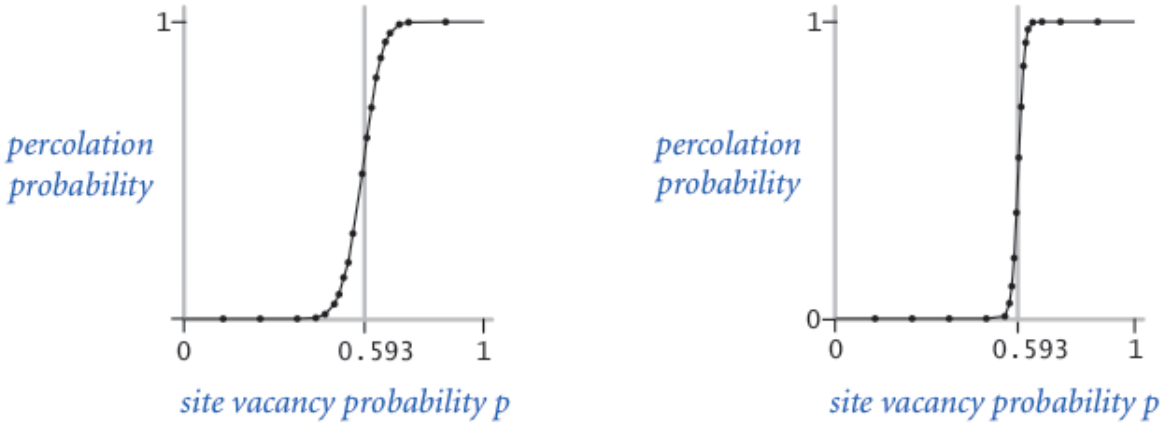
Goal Write a program to estimate the percolation threshold of a system.

Percolation Given a composite system comprising of randomly distributed insulating and metallic materials: what fraction of the system needs to be metallic so that the composite system is an electrical conductor? Given a porous landscape with water on the surface (or oil below), under what conditions will the water be able to drain through to the bottom (or the oil to gush through to the surface)? Scientists have defined an abstract process known as *percolation* to model such situations.

The Model We model a percolation system using an nxn grid of sites. Each site is either open or blocked. A full site is an open site that can be connected to an open site in the top row via a chain of neighboring (left, right, up, down) open sites. We say the system percolates if there is a full site in the bottom row. In other words, a system percolates if we fill all open sites connected to the top row and that process fills some open site on the bottom row. For the insulating/metallic materials example, the open sites correspond to metallic materials, so that a system that percolates has a metallic path from top to bottom, with full sites conducting. For the porous substance example, the open sites correspond to empty space through which water might flow, so that a system that percolates lets water fill open sites, flowing from top to bottom.



The Problem If sites are independently set to be open with probability p (and therefore blocked with probability $1 - p$), what is the probability that the system percolates? When p equals 0, the system does not percolate; when p equals 1, the system percolates. The plots below show the site vacancy probability p versus the percolation probability for 20×20 random grid (left) and 100×100 random grid (right).



When n is sufficiently large, there is a threshold value $p^\star$ such that when $p < p^\star$ a random $n \times n$ grid almost never percolates, and when $p > p^\star$, a random $n \times n$ grid almost always percolates. No mathematical solution for determining the percolation threshold $p^\star$ has yet been derived. Your task is to write a computer program to estimate $p^\star$.

Percolation API To model a percolation system, we define an interface called `Percolation`, supporting the following API:

Percolation	
`void open(int i, int j)`	opens site `(i, j)` if it is not already open
`boolean isOpen(int i, int j)`	returns `true` if site `(i, j)` is open, and `false` otherwise
`boolean isFull(int i, int j)`	returns `true` if site `(i, j)` is full, and `false` otherwise
`int numberOfOpenSites()`	returns the number of open sites
`boolean percolates()`	returns `true` if this system percolates, and `false` otherwise

Problem 1. (*Array Percolation*) Develop a data type called `ArrayPercolation` that implements the `Percolation` interface using a 2D array as the underlying data structure.

`ArrayPercolation implements UF`

`ArrayPercolation(int n)`	constructs an `n x n` percolation system, with all sites blocked

Corner cases:

- `ArrayPercolation()` should throw an `IllegalArgumentException("Illegal n")` if $n \leq 0$.
- `open()`, `isOpen()`, and `isFull()` should throw an `IndexOutOfBoundsException("Illegal i or j")` if i or j is outside the interval $[0, n-1]$.

Performance requirements:

- `open()`, `isOpen()`, and `numberOfOpenSites()` should run in time $T(n) \sim 1$.
- `percolates()` should run in time $T(n) \sim n$.
- `ArrayPercolation()` and `isFull()` should run in time $T(n) \sim n^2$.

`>_ ~/workspace/project1`

```
$ java ArrayPercolation data/input10.txt
10 x 10 system:
  Open sites = 56
  Percolates = true
$ java ArrayPercolation data/input10-no.txt
10 x 10 system:
  Open sites = 55
  Percolates = false
```

Directions:

- Instance variables:
  - Percolation system size, `int n`.
  - Percolation system, `boolean[][] open` (`true` $\implies$ open site and `false` $\implies$ blocked site).
  - Number of open sites, `int openSites`.
- `private void floodFill(boolean[][] full, int i, int j)`
  - Return if `i` or `j` is out of bounds; or site `(i, j)` is not open; or site `(i, j)` is full (ie, `full[i][j]` is `true`).
  - Fill site `(i, j)`.
  - Call `floodFill()` recursively on the sites to the north, east, west, and south of site `(i, j)`.
- `public ArrayPercolation(int n)`
  - Initialize instance variables.
- `void open(int i, int j)`
  - If site `(i, j)` is not open:
    - Open the site.
    - Increment `openSites` by one.
- `boolean isOpen(int i, int j)`
  - Return whether site `(i, j)` is open or not.
- `boolean isFull(int i, int j)`
  - Create an $n \times n$ array of booleans called `full`.

  - Call `floodFill()` on every site in the first row of the percolation system, passing `full` as the first argument.
  - Return `full[i][j]`.

- `int numberOfOpenSites()`
  - Return the number of open sites.
- `boolean percolates()`
  - Return whether the system percolates or not — a system percolates if the last row contains at least one full site.

Problem 2. (*Union Find Percolation*) Develop a data type called `UFPercolation` that implements the `Percolation` interface using a `WeightedQuickUnionUF` object as the underlying data structure.

`UFPercolation implements UF`

`UFPercolation(int n)`	constructs an `n x n` percolation system, with all sites blocked

Corner cases:

- `UFPercolation()` should throw an `IllegalArgumentException("Illegal n")` if $n \leq 0$.
- `open()`, `isOpen()`, and `isFull()` should throw an `IndexOutOfBoundsException("Illegal i or j")` if i or j is outside the interval $[0, n-1]$.

Performance requirements:

- `isOpen()` and `numberOfOpenSites()` should run in time $T(n) \sim 1$.
- `open()`, `isFull()`, and `percolates()` should run in time $T(n) \sim \log n$.
- `UFPercolation()` should run in time $T(n) \sim n^2$.

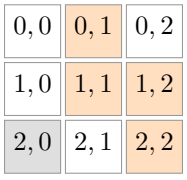
`~/workspace/project1`

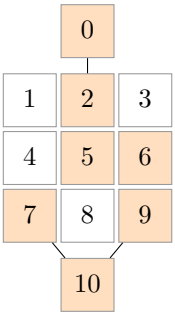
```
$ java UFPercolation data/input10.txt
10 x 10 system:
  Open sites = 56
  Percolates = true
$ java UFPercolation data/input10-no.txt
10 x 10 system:
  Open sites = 55
  Percolates = false
```

Directions:

- Model percolation system as an $n \times n$ array of booleans (`true` $\implies$ open site and `false` $\implies$ blocked site).
- Create an `UF` object with $n^2 + 2$ sites and use the private `encode()` method to translate sites $(0,0), (0,1), \ldots, (n-1, n-1)$ of the array to sites $1, 2, \ldots, n^2$ of the `UF` object; sites 0 (source) and $n^2 + 1$ (sink) are virtual, ie, not part of the percolation system.
- A 3×3 percolation system and its `UF` representation

- Instance variables:
  - Percolation system size, `int n`.
  - Percolation system, `boolean[][] open`.
  - Number of open sites, `int openSites`.
  - Union-find representation of the percolation system, `WeightedQuickUnionUF uf`.
- `private int encode(int i, int j)`
  - Return the `UF` site $(1, 2, \ldots, n^2)$ corresponding to the percolation system site `(i, j)`.
- `public UFPercolation(int n)`
  - Initialize instance variables.
- `void open(int i, int j)`
  - If site `(i, j)` is not open:
    - Open the site
    - Increment `openSites` by one.
    - If the site is in the first (or last) row, connect the corresponding `uf` site with the source (or sink).
    - If any of the neighbors to the north, east, west, and south of site `(i, j)` is open, connect the `uf` site corresponding to site `(i, j)` with the `uf` site corresponding to that neighbor.
- `boolean isOpen(int i, int j)`
  - Return whether site `(i, j)` is open or not.
- `boolean isFull(int i, int j)`
  - Return whether site `(i, j)` is full or not — a site is full if it is open and its corresponding `uf` site is connected to the source.
- `int numberOfOpenSites()`
  - Return the number of open sites.
- `boolean percolates()`
  - Return whether the system percolates or not — a system percolates if the sink is connected to the source.
- Using virtual source and sink sites introduces what is called the *back wash* problem.
- In the 3×3 system, consider opening the sites $(0, 1)$, $(1, 2)$,$(1, 1)$, $(2, 0)$, and $(2, 2)$, and in that order; the system percolates once $(2, 2)$ is opened.





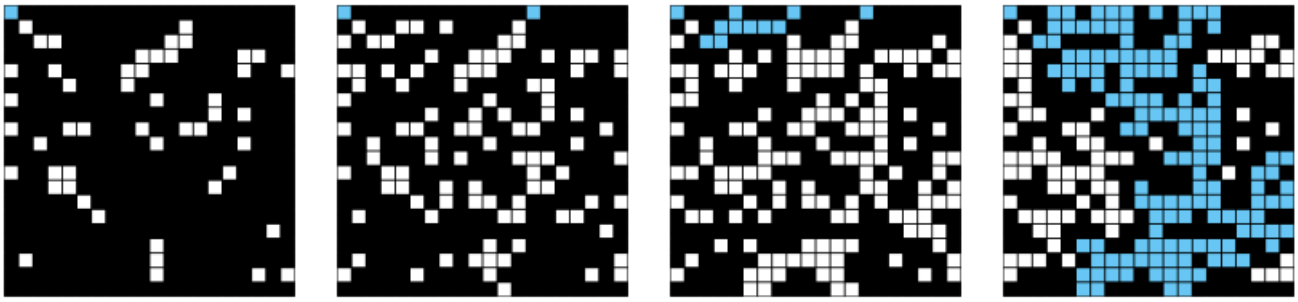
- The site $(2, 0)$ is technically not full since it is not connected to an open site in the top row via a path of neighboring (north, east, west, and south) open sites, but the corresponding `uf` site (7) is connected to the source, so is incorrectly reported as being full — this is the back wash problem.

- To receive full credit, you must resolve the back wash problem.

Problem 3. (*Estimation of Percolation Threshold*) To estimate the percolation threshold, consider the following computational (Monte Carlo simulation) experiment:

- Create an $n \times n$ percolation system (use the `UFPercolation` implementation) with all sites blocked.
- Repeat the following until the system percolates:
  - Choose a site (row i, column j) uniformly at random among all blocked sites.
  - Open the site (row i, column j).
- The fraction of sites that are open when the system percolates provides an estimate of the percolation threshold.

For example, if sites are opened in a 20×20 grid according to the snapshots below, then our estimate of the percolation threshold is $204/400 = 0.51$ because the system percolates when the 204th site is opened.



50 open sites *100 open sites* *150 open sites* *204 open sites*

By repeating this computational experiment m times and averaging the results, we obtain a more accurate estimate of the percolation threshold. Let $x_1, x_2, \ldots, x_m$ be the fractions of open sites in computational experiments $1, 2, \ldots, m$. The sample mean μ provides an estimate of the percolation threshold, and the sample standard deviation σ measures the sharpness of the threshold:

$$\mu = \frac{x_1 + x_2 + \cdots + x_m}{m}, \quad \sigma^2 = \frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_m - \mu)^2}{m - 1}.$$

Assuming m is sufficiently large (say, at least 30), the following interval provides a 95% confidence interval for the percolation threshold:

$$\left[\mu - \frac{1.96\sigma}{\sqrt{m}}, \mu + \frac{1.96\sigma}{\sqrt{m}}\right].$$

To perform a series of computational experiments, create an immutable data type called `PercolationStats` that supports the following API:

`PercolationStats`	
`PercolationStats(int n, int m)`	performs `m` independent experiments on an `n x n` percolation system
`double mean()`	returns sample mean of percolation threshold
`double stddev()`	returns sample standard deviation of percolation threshold
`double confidenceLow()`	returns low endpoint of 95% confidence interval
`double confidenceHigh()`	returns high endpoint of 95% confidence interval

The constructor perform m independent computational experiments (discussed above) on an $n \times n$ grid. Using this experimental data, it should calculate the mean, standard deviation, and the 95% confidence interval for the percolation threshold.

Corner cases:

- The constructor should throw an `IllegalArgumentException("Illegal n or m")` if either $n \leq 0$ or $m \leq 0$.

Performance requirements:

- `mean()`, `stddev()`, `confidenceLow()`, and `confidenceHigh()` should run in time $T(n, m) \sim m$.
- `PercolationStats()` should run in time $T(n, m) \sim mn^2$.

```
>_ ~/workspace/project1
$ java PercolationStats 100 1000
Percolation threshold for a 100 x 100 system:
  Mean                = 0.592
  Standard deviation  = 0.016
  Confidence interval = [0.591, 0.594]
```

Directions:

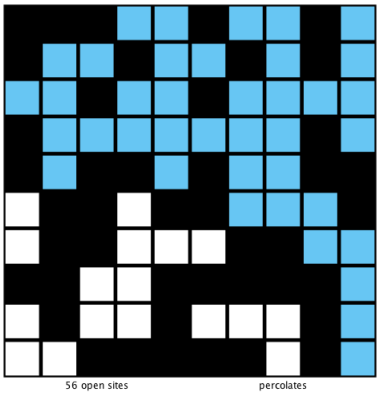
- Instance variables:
  - Number of independent experiments, `int m`.
  - Percolation thresholds for the `m` experiments, `double[] x`.
- `PercolationStats(int n, int m)`
  - Initialize instance variables.
  - Perform the following experiment `m` times:
    - Create an $n \times n$ percolation system (use the `UFPercolation` implementation).
    - Until the system percolates, choose a site (i, j) at random and open it if it is not already open.
    - Calculate percolation threshold as the fraction of sites opened, and store the value in `x[]`.
- `double mean()`
  - Return the mean μ of the values in `x[]`.
- `double stddev()`
  - Return the standard deviation σ of the values in `x[]`.
- `double confidenceLow()`
  - Return $\mu - \frac{1.96\sigma}{\sqrt{m}}$.
- `double confidenceHigh()`
  - Return $\mu + \frac{1.96\sigma}{\sqrt{m}}$.

Data The `data` directory contains some input `.txt` files for the percolation visualization programs, and associated with each file is an output `.png` file that shows the desired output. For example

```
>_ ~/workspace/project1
$ cat data/input10.txt
10
9 1
1 9
...
7 9
```

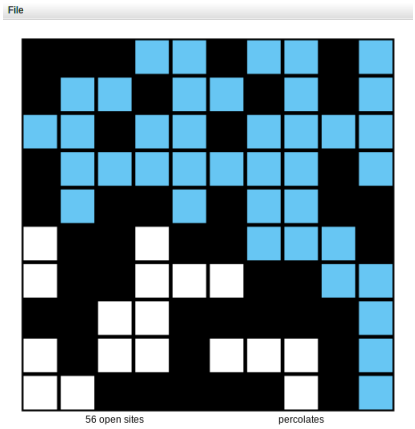
The first number specifies the size of the percolation system and the pairs of numbers that follow specify the sites to open.

```
>_ ~/workspace/project1
$ display data/input10.png
```



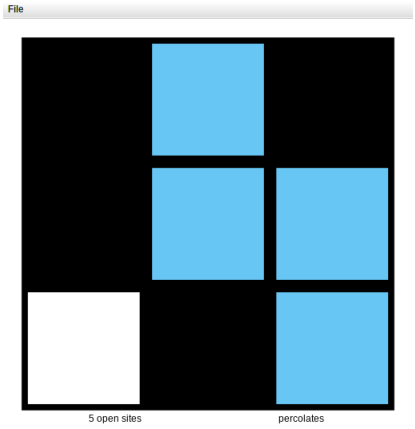
Visualization Programs The program `PercolationVisualizer` accepts *mode* (the String "array" or "UF") and *filename* (String) as command-line arguments, and uses `ArrayPercolation` or `UFPercolation` to determine and visually report if the system represented by the input file percolates or not.

```
>_ ~/workspace/project1
$ java PercolationVisualizer UF data/input10.txt
```



The program `InteractivePercolationVisualizer` accepts *mode* ("array" or "UF") and n (int) as command-line arguments, constructs an $n \times n$ percolation system using `ArrayPercolation` or `UFPercolation`, and allows you to interactively open sites in the system by clicking on them and visually inspect if the system percolates or not.

```
>_ ~/workspace/project1
$ java InteractivePercolationVisualizer UF 3
3
0 1
1 2
1 1
2 0
2 2
```



Acknowledgements This project is an adaptation of the Percolation assignment developed at Princeton University by Robert Sedgewick and Kevin Wayne.

Files to Submit

1. `GreatCircle.java`
2. `PrimeCounter.java`
3. `Distance.java`
4. `Transpose.java`
5. `Rational.java`
6. `Harmonic.java`
7. `ArrayPercolation.java`
8. `UFPercolation.java`
9. `PercolationStats.java`
10. `report.txt`

Before you submit your files, make sure:

- Your programs meet the style requirements by running the following command in the terminal.

```
>_ ~/workspace/project1
$ check_style src/*.java
```

- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You use the template file `report.txt` for your report.
- Your report meets the prescribed guidelines.

Exercises

Exercise 1. (*Iterable Binary Strings*) Implement an immutable, iterable data type called `BinaryStrings` to systematically iterate over binary strings of length n. The data type must support the following API:

BinaryStrings

`BinaryStrings(int n)`	constructs an iterable `BinaryStrings` object given the length of binary strings needed
`Iterator<String> iterator()`	returns an iterator to iterate over binary strings of length n

~/workspace/project2

```
$ java BinaryStrings 3
000
001
010
011
100
101
110
111
```

BinaryStrings.java

```
import java.util.Iterator;

import stdlib.StdOut;

// An immutable data type to systematically iterate over binary strings of length n.
public class BinaryStrings implements Iterable<String> {
    private int n; // need all binary strings of length n

    // Constructs a BinaryStrings object given the length of binary strings needed.
    public BinaryStrings(int n) {
        ...
    }

    // Returns an iterator to iterate over binary strings of length n.
    public Iterator<String> iterator() {
        ...
    }

    // Binary strings iterator.
    private class BinaryStringsIterator implements Iterator<String> {
        private int count; // number of binary strings returned so far
        private int p;     // current number in decimal

        // Constructs an iterator.
        public BinaryStringsIterator() {
            ...
        }

        // Returns true if there are anymore binary strings to be iterated, and false otherwise.
        public boolean hasNext() {
            ...
        }

        // Returns the next binary string.
        public String next() {
            ...
        }

        // Returns the n-bit binary representation of x.
        private String binary(int x) {
            String s = Integer.toBinaryString(x);
            int padding = n - s.length();
            for (int i = 1; i <= padding; i++) {
                s = "0" + s;
            }
            return s;
        }
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        for (String s : new BinaryStrings(n)) {
```

```
            StdOut.println(s);
        }
    }
}
```

Exercise 2. (*Iterable Primes*) Implement an immutable, iterable data type called `Primes` to systematically iterate over the first n primes. The data type must support the following API:

Primes

`Primes(int n)`	constructs a `Primes` object given the number of primes needed
`Iterator<Integer> iterator()`	returns an iterator to iterate over the first n primes

~/workspace/project2

```
$ java Primes 10
2
3
5
7
11
13
17
19
23
29
```

Primes.java

```
import java.util.Iterator;

import stdlib.StdOut;

// An immutable data type to systematically iterate over the first n primes.
public class Primes implements Iterable<Integer> {
    private int n; // need first n primes

    // Constructs a Primes object given the number of primes needed.
    public Primes(int n) {
        ...
    }

    // Returns an iterator to iterate over the first n primes.
    public Iterator<Integer> iterator() {
        ...
    }

    // Primes iterator.
    private class PrimesIterator implements Iterator<Integer> {
        private int count; // number of primes returned so far
        private int p;     // current prime

        // Constructs an iterator.
        public PrimesIterator() {
            ...
        }

        // Returns true if there are anymore primes to be iterated, and false otherwise.
        public boolean hasNext() {
            ...
        }

        // Returns the next prime.
        public Integer next() {
            // Increment count by 1.
            ...

            // As long as p is not prime, increment p by 1.
            ...

            // Return current value of p and increment it by 1.
            ...
        }

        // Returns true if x is a prime, and false otherwise.
        private boolean isPrime(int x) {
```

```
            for (int i = 2; i <= x / i; i++) {
                if (x % i == 0) {
                    return false;
                }
            }
            return true;
        }
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        for (int i : new Primes(n)) {
            StdOut.println(i);
        }
    }
}
```

Exercise 3. (*Min Max*) Implement a library called `MinMax` with static methods `min()` and `max()` that accept a reference `first` to the first node in a linked list of integer-valued items and return the minimum and the maximum values respectively.

>_ ~/workspace/project2

```
$ java MinMax
min(first) == StdStats.min(items)? true
max(first) == StdStats.max(items)? true
```

MinMax.java

```
import stdlib.StdOut;
import stdlib.StdRandom;
import stdlib.StdStats;

public class MinMax {
    // Returns the minimum value in the given linked list.
    public static int min(Node first) {
        // Set min to the largest integer.
        ...

        // Compare each element in linked list with min and if it is smaller, update min.
        ...

        // Return min.
        ...
    }

    // Returns the maximum value in the given linked list.
    public static int max(Node first) {
        // Set max to the smallest integer.
        ...

        // Compare each element in linked list with max and if it is larger, update max.
        ...

        // Return max.
        ...
    }

    // A data type to represent a linked list. Each node in the list stores an integer item and a
    // reference to the next node in the list.
    protected static class Node {
        protected int item;  // the item
        protected Node next; // the next node
    }

    // Unit tests the library. [DO NOT EDIT]
    public static void main(String[] args) {
        int[] items = new int[1000];
        for (int i = 0; i < 1000; i++) {
            items[i] = StdRandom.uniform(-10000, 10000);
        }
        Node first = null;
        for (int item : items) {
            Node oldfirst = first;
            first = new Node();
            first.item = item;
            first.next = oldfirst;
```

```
        }
        StdOut.println("min(first) == StdStats.min(items)? " + (min(first) == StdStats.min(items)));
        StdOut.println("max(first) == StdStats.max(items)? " + (max(first) == StdStats.max(items)));
    }
}
```

Exercise 4. (*Text Editor Buffer*) Implement a data type called `Buffer` to represent a buffer in a text editor. The data type must support the following API:

Buffer

`Buffer()`	creates an empty buffer
`void insert(char c)`	inserts `c` at the cursor position
`char delete()`	deletes and returns the character immediately ahead of the cursor
`void left(int k)`	moves the cursor `k` positions to the left
`void right(int k)`	moves the cursor `k` positions to the right
`int size()`	returns the number of characters in this buffer
`String toString()`	returns a string representation of this buffer with the `"|"` character (not part of the buffer) at the cursor position

~/workspace/project2

```
$ java Buffer
|There is grandeur in this view of life, with its several powers,having been originally breathed by the
Creator into a few forms or into one; and that, whilst this planet has gone cycling on according to the
fixed law of gravity, from so simple a beginning endless forms most beautiful and most wonderful have
been, and are being, evolved. -- Charles Darwin, The Origin of Species
```

Hint: Use two stacks `left` and `right` to store the characters to the left and right of the cursor, with the characters on top of the stacks being the ones immediately to its left and right.

Buffer.java

```
import dsa.LinkedStack;
import stdlib.StdOut;

// A data type to represent a text editor buffer.
public class Buffer {
    protected LinkedStack<Character> left;  // chars left of cursor
    protected LinkedStack<Character> right; // chars right of cursor

    // Creates an empty buffer.
    public Buffer() {
        ...
    }

    // Inserts c at the cursor position.
    public void insert(char c) {
        ...
    }

    // Deletes and returns the character immediately ahead of the cursor.
    public char delete() {
        ...
    }

    // Moves the cursor k positions to the left.
    public void left(int k) {
        ...
    }

    // Moves the cursor k positions to the right.
    public void right(int k) {
        ...
    }

    // Returns the number of characters in this buffer.
    public int size() {
        ...
    }

    // Returns a string representation of the buffer with the "|" character (not part of the buffer)
```

```
    // at the cursor position.
    public String toString() {
        ...

        // Push chars from left into a temporary stack.
        ...

        // Append chars from temporary stack to sb.
        ...

        // Append "|" to sb.
        ...

        // Append chars from right to sb.
        ...

        // Return the string from sb.
        ...
    }

    // Unit tests the data type (DO NOT EDIT).
    public static void main(String[] args) {
        Buffer buf = new Buffer();
        String s = "There is grandeur in this view of life, with its several powers, having been " +
                "originally breathed into a few forms or into one; and that, whilst this planet " +
                "has gone cycling on according to the fixed law of gravity, from so simple a " +
                "beginning endless forms most beautiful and most wonderful have been, and are " +
                "being, evolved. ~ Charles Darwin, The Origin of Species";
        for (int i = 0; i < s.length(); i++) {
            buf.insert(s.charAt(i));
        }
        buf.left(buf.size());
        buf.right(97);
        s = "by the Creator ";
        for (int i = 0; i < s.length(); i++) {
            buf.insert(s.charAt(i));
        }
        buf.right(228);
        buf.delete();
        buf.insert('-');
        buf.insert('-');
        buf.left(342);
        StdOut.println(buf);
    }
}
```

Exercise 5. (*Josephus Problem*) In the Josephus problem from antiquity, n people are in dire straits and agree to the following strategy to reduce the population. They arrange themselves in a circle (at positions numbered from 1 to n) and proceed around the circle, eliminating every mth person until only one person is left. Legend has it that Josephus figured out where to sit to avoid being eliminated. Implement a program `Josephus.java` that accepts n (int) and m (int) as command-line arguments, and writes to standard output the order in which people are eliminated (and thus would show Josephus where to sit in the circle).

```
>_ ~/workspace/project2
$ java Josephus 7 2
2
4
6
1
5
3
7
```

```
Josephus.java
import dsa.LinkedQueue;
import stdlib.StdOut;

public class Josephus {
    // Entry point.
    public static void main(String[] args) {
        // Accept n (int) and m (int) as command-line arguments.
        ...
```

```
        // Create a queue q and enqueue integers 1, 2, ..., n.
        ...

        // Set i to 0. As long as q is not empty: increment i; dequeue an element (say pos); if m
        // divides i, write pos to standard output, otherwise enqueue pos to q.
        ...
    }
}
```

Problems

Goal The purpose of this project is to implement elementary data structures using arrays and linked lists, and to introduce you to generics and iterators.

Problem 1. (*Deque*) A double-ended queue or deque (pronounced "deck") is a generalization of a stack and a queue that supports adding and removing items from either the front or the back of the data structure. Create a generic, iterable data type called `LinkedDeque` that uses a doubly-linked list to implement the following deque API:

LinkedDeque

`LinkedDeque()`	constructs an empty deque
`boolean isEmpty()`	returns `true` if this deque empty, and `false` otherwise
`int size()`	returns the number of items on this deque
`void addFirst(Item item)`	adds `item` to the front of this deque
`void addLast(Item item)`	adds `item` to the back of this deque
`Item peekFirst()`	returns the item at the front of this deque
`Item removeFirst()`	removes and returns the item at the front of this deque
`Item peekLast()`	returns the item at the back of this deque
`Item removeLast()`	removes and returns the item at the back of this deque
`Iterator<Item> iterator()`	returns an iterator to iterate over the items in this deque from front to back
`String toString()`	returns a string representation of this deque

Corner Cases

- The `add*()` methods should throw a `NullPointerException("item is null")` if *item* is `null`.
- The `peek*()` and `remove*()` methods should throw a `NoSuchElementException("Deque is empty")` if the deque is empty.
- The `next()` method in the deque iterator shoud throw a `NoSuchElementException("Iterator is empty")` if there are no more items to iterate.

Performance Requirements

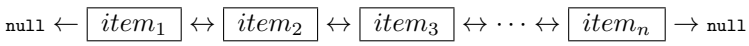
- The constructor and methods in `LinkedDeque` and `DequeIterator` should run in time $T(n) \sim 1$.

>_ ~/workspace/project2

```
$ java LinkedDeque
Filling the deque...
The deque (364 characters): There is grandeur in this view of life, with its several powers, having been originally
breathed into a few forms or into one; and that, whilst this planet has gone cycling on according to the fixed law
of gravity, from so simple a beginning endless forms most beautiful and most wonderful have been, and are being,
evolved. ~ Charles Darwin, The Origin of Species
Emptying the deque...
deque.isEmpty()? true
```

Directions:

- Use a doubly-linked list `Node` to implement the API — each node in the list stores a generic `item`, and references `next` and `prev` to the next and previous nodes in the list



- Instance variables:
  - Reference to the front of the deque, `Node first`.
  - Reference to the back of the deque, `Node last`.
  - Size of the deque, `int n`.
- `LinkedDeque()`
  - Initialize instance variables to appropriate values.
- `boolean isEmpty()`
  - Return whether the deque is empty or not.
- `int size()`
  - Return the size of the deque.
- `void addFirst(Item item)`
  - Add the given item to the front of the deque.
  - Increment `n` by one.
- `void addLast(Item item)`
  - Add the given item to the back of the deque.
  - Increment `n` by one.
- `Item peekFirst()`
  - Return the item at the front of the deque.
- `Item removeFirst()`
  - Remove and return the item at the front of the deque.
  - Decrement `n` by one.
- `Item peekLast()`
  - Return the item at the back of the deque.
- `Item removeLast()`
  - Remove and return the item at the back of the deque.
  - Decrement `n` by one.
- `Iterator<Item> iterator()`
  - Return an object of type `DequeIterator`.
- `LinkedDeque` :: `DequeIterator`.
  - Instance variable:
    - Reference to current node in the iterator, `Node current`.
  - `DequeIterator()`
    - Initialize instance variable appropriately.

- `boolean hasNext()`
  - Return whether the iterator has more items to iterate or not.
- `Item next()`
  - Return the item in `current` and advance `current` to the next node.

Problem 2. (*Sorting Strings*) Implement a program called `Sort.java` that accepts strings from standard input, stores them in a `LinkedDeque` data structure, sorts the deque, and writes the sorted strings to standard output.

Performance Requirements

- The program should run in time $T(n) \sim n^2$, where n is the number of input strings.

```
>_ ~/workspace/project2
$ java Sort
A B R A C A D A B R A
<ctrl-d>
A
A
A
A
A
B
B
C
D
R
R
```

Directions:

- Create a queue `d`.
- For each word `w` read from standard input
  - Add `w` to the front of `d` if it is less† than the first word in `d`.
  - Add `w` to the back of `d` if it is greater† than the last word in `d`.
  - Otherwise, remove words that are less than `w` from the front of `d` and store them in a temporary stack `s`; add `w` to the front of `d`; and add words from `s` also to the front of `d`.
- Write the words from `d` to standard output.

† Use the helper method `boolean less(String v, String w)` to test if a string `v` is less than a string `w`.

Problem 3. (*Random Queue*) A random queue is similar to a stack or queue, except that the item removed is chosen uniformly at random from items in the data structure. Create a generic, iterable data type called `ResizingArrayRandomQueue` that uses a resizing array to implement the following random queue API:

ResizingArrayRandomQueue	
`ResizingArrayRandomQueue()`	constructs an empty random queue
`boolean isEmpty()`	returns `true` if this queue is empty, and `false` otherwise
`int size()`	returns the number of items in this queue
`void enqueue(Item item)`	adds *item* to the end of this queue
`Item sample()`	returns a random item from this queue
`Item dequeue()`	removes and returns a random item from this queue
`Iterator<Item> iterator()`	returns an independent† iterator to iterate over the items in this queue in random order
`String toString()`	returns a string representation of this queue

† The order of two or more iterators on the same randomized queue must be mutually independent, ie, each iterator must maintain its own random order.

Corner Cases

- The `enqueue()` method should throw a `NullPointerException("item is null")` if *item* is `null`.
- The `sample()` and `dequeue()` methods should throw a `NoSuchElementException("Random queue is empty")` if the random queue is empty.
- The `next()` method in the random queue iterator shoud throw a `NoSuchElementException("Iterator is empty")` if there are no more items to iterate.

Performance Requirements

- The constructor and methods in `ResizingArrayRandomQueue` should run in time $T(n) \sim 1$.
- The constructor in `RandomQueueIterator` should run in time $T(n) \sim n$.
- The methods in `RandomQueueIterator` should run in time $T(n) \sim 1$.

```
>_ ~/workspace/project2
$ java ResizingArrayRandomQueue
sum       = 5081434
iterSumQ  = 5081434
dequeSumQ = 5081434
iterSumQ + dequeSumQ == 2 * sum? true
```

Directions:

- Use a resizing array to implement the API
- Instance variables:
  - Array to store the items of queue, `Item[] q`.
  - Size of the queue, `int n`.
- `ResizingArrayRandomQueue()`
  - Initialize instance variables appropriately — create `q` with an initial capacity of 2.
- `boolean isEmpty()`
  - Return whether the queue is empty or not.
- `int size()`
  - Return the size of the queue.
- `void enqueue(Item item)`
  - If `q` is at full capacity, resize it to twice its current capacity.
  - Insert the given item in `q` at index `n`.
  - Increment `n` by one.
- `Item sample()`
  - Return `q[r]`, where `r` is a random integer from the interval `[0, n)`.
- `Item dequeue()`
  - Save `q[r]` in `item`, where `r` is a random integer from the interval `[0, n)`.
  - Set `q[r]` to `q[n - 1]` and `q[n - 1]` to `null`.

  - If `q` is at quarter capacity, resize it to half its current capacity.
  - Decrement `n` by one.
  - Return `item`.

- `Iterator<Item> iterator()`
  - Return an object of type `RandomQueueIterator`.
- `ResizingArrayRandomQueue :: RandomQueueIterator()`
  - Instance variables:
    * Array to store the items of `q`, `Item[] items`.
    * Index of the current item in `items`, `int current`.
  - `RandomQueueIterator()`
    * Create `items` with capacity `n`.
    * Copy the `n` items from `q` into `items`.
    * Shuffle `items`.
    * Initialize `current` appropriately.
  - `boolean hasNext()`
    * Return whether the iterator has more items to iterate or not.
  - `Item next()`
    * Return the item in `items` at index `current` and advance `current` by one.

Problem 4. (*Sampling Integers*) Implement a program called `Sample.java` that accepts *lo* (int), *hi* (int), *k* (int), and *mode* (String) as command-line arguments, uses a random queue to sample *k* integers from the interval $[lo, hi]$, and writes the samples to standard output. The sampling must be done with replacement if *mode* is "+", and without replacement if *mode* is "-". You may assume that $k \leq hi - lo + 1$.

Corner Cases

- The program should throw an `IllegalArgumentException("Illegal mode")` if *mode* is different from "+" or "-".

Performance Requirements

- The program should run in time $T(k, n) \sim kn$ in the worst case (sampling without replacement), where k is the sample size and n is the length of the sampling interval.

```
>_ ~/workspace/project2
$ java Sample 1 5 5 +
3
3
5
4
1
$ java Sample 1 5 5 -
2
3
1
4
5
```

Directions:

- Accept *lo* (int), *hi* (int), *k* (int), and *mode* (String) as command-line arguments.
- Create a random queue *q* containing integers from the interval $[lo, hi]$.

- If *mode* is "+" (sampling with replacement), sample and write k integers from q to standard output.
- If *mode* is "-" (sampling without replacement), dequeue and write k integers from q to standard. output

Acknowledgements This project is an adaptation of the Deques and Randomized Queues assignment developed at Princeton University by Kevin Wayne.

Files to Submit

1. `BinaryStrings.java`
2. `Primes.java`
3. `MinMax.java`
4. `Buffer.java`
5. `Josephus.java`
6. `LinkedDeque.java`
7. `Sort.java`
8. `ResizingArrayRandomQueue.java`
9. `Sample.java`
10. `report.txt`

Before you submit your files, make sure:

- Your programs meet the style requirements by running the following command in the terminal.

```
>_ ~/workspace/project2
$ check_style src/*.java
```

- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You use the template file `report.txt` for your report.
- Your report meets the prescribed guidelines.

Exercises

Exercise 1. (*Comparable Six-sided Die*) Implement a comparable data type called `Die` that represents a six-sided die and supports the following API:

Die

`Die()`	constructs a die
`void roll()`	rolls this die
`int value()`	returns the face value of this die
`boolean equals(Die other)`	returns `true` if this die is the same as `other`, and `false` otherwise
`int compareTo(Die other)`	returns a comparison of this die with `other`, by their face values
`String toString()`	returns a string representation of this die

>_ ~/workspace/project3

```
$ java Die 5 3 4
Dice a, b, and c:
*   *
  *
*   *
*
  *
    *
*   *

*   *
a.equals(b)    = false
b.equals(c)    = false
a.compareTo(b) = 2
b.compareTo(c) = -1
```

Die.java

```
import stdlib.StdOut;
import stdlib.StdRandom;

public class Die implements Comparable<Die> {
    private int value; // the face value

    // Constructs a die.
    public Die() {
        ...
    }

    // Rolls this die.
    public void roll() {
        ...
    }

    // Returns the face value of this die.
    public int value() {
        ...
    }

    // Returns true if this die is the same as other, and false otherwise.
    public boolean equals(Object other) {
        if (other == this) {
            return true;
        }
        if (other == null) {
            return false;
        }
        if (other.getClass() != this.getClass()) {
            return false;
        }
        ...
    }

    // Returns a comparison of this die with other, by their face values.
    public int compareTo(Die that) {
        ...
    }

    // Returns a string representation of this die.
```

```
    public String toString() {
        ...
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);
        int z = Integer.parseInt(args[2]);
        Die a = new Die();
        a.roll();
        while (a.value() != x) {
            a.roll();
        }
        Die b = new Die();
        b.roll();
        while (b.value() != y) {
            b.roll();
        }
        Die c = new Die();
        c.roll();
        while (c.value() != z) {
            c.roll();
        }
        StdOut.println("Dice a, b, and c:");
        StdOut.println(a);
        StdOut.println(b);
        StdOut.println(c);
        StdOut.println("a.equals(b)    = " + a.equals(b));
        StdOut.println("b.equals(c)    = " + b.equals(c));
        StdOut.println("a.compareTo(b) = " + a.compareTo(b));
        StdOut.println("b.compareTo(c) = " + b.compareTo(c));
    }
}
```

Exercise 2. (*Comparable Geo Location*) Implement an immutable data type called `Location` that represents a location on Earth and supports the following API:

Location	
`Location(String name, double lat, double lon)`	constructs a new location given its name, latitude, and longitude
`double distanceTo(Location other)`	returns the great-circle distance† between this location and `other`
`boolean equals(Object other)`	returns `true` if this location is the same as `other`, and `false` otherwise
`String toString()`	returns a string representation of this location
`int compareTo(Location other)`	returns a comparison of this location with `other` based on their respective distances to the origin, Parthenon (Greece) @ 37.971525, 23.726726

† See Exercise 1 of Project 1 for formula.

```
>_ ~/workspace/project3
$ java Location 2 XYZ 27.1750 78.0419
Seven wonders, in the order of their distance to Parthenon (Greece):
  The Colosseum (Italy) (41.8902, 12.4923)
  Petra (Jordan) (30.3286, 35.4419)
  Taj Mahal (India) (27.175, 78.0419)
  Christ the Redeemer (Brazil) (22.9519, -43.2106)
  The Great Wall of China (China) (40.6769, 117.2319)
  Chichen Itza (Mexico) (20.6829, -88.5686)
  Machu Picchu (Peru) (-13.1633, -72.5456)
wonders[2] == XYZ (27.175, 78.0419)? true
```

Location.java

```
import java.util.Arrays;

import stdlib.StdOut;

public class Location implements Comparable<Location> {
    private String name; // location name
    private double lat;  // latitude
    private double lon;  // longitude
```

```
    // Constructs a new location given its name, latitude, and longitude.
    public Location(String name, double lat, double lon) {
        ...
    }

    // Returns the great-circle distance between this location and other.
    public double distanceTo(Location other) {
        ...
    }

    // Returns true if this location is the same as other, and false otherwise.
    public boolean equals(Object other) {
        if (other == null) {
            return false;
        }
        if (other == this) {
            return true;
        }
        if (other.getClass() != this.getClass()) {
            return false;
        }
        ...
    }

    // Returns a string representation of this location.
    public String toString() {
        ...
    }

    // Returns a comparison of this location with other based on their respective distances to
    // the origin, Parthenon (Greece) @ 37.971525, 23.726726.
    public int compareTo(Location that) {
        ...
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        int rank = Integer.parseInt(args[0]);
        String name = args[1];
        double lat = Double.parseDouble(args[2]);
        double lon = Double.parseDouble(args[3]);
        Location[] wonders = new Location[7];
        wonders[0] = new Location("The Great Wall of China (China)", 40.6769, 117.2319);
        wonders[1] = new Location("Petra (Jordan)", 30.3286, 35.4419);
        wonders[2] = new Location("The Colosseum (Italy)", 41.8902, 12.4923);
        wonders[3] = new Location("Chichen Itza (Mexico)", 20.6829, -88.5686);
        wonders[4] = new Location("Machu Picchu (Peru)", -13.1633, -72.5456);
        wonders[5] = new Location("Taj Mahal (India)", 27.1750, 78.0419);
        wonders[6] = new Location("Christ the Redeemer (Brazil)", 22.9519, -43.2106);
        Arrays.sort(wonders);
        StdOut.println("Seven wonders, in the order of their distance to Parthenon (Greece):");
        for (Location wonder : wonders) {
            StdOut.println("  " + wonder);
        }
        Location loc = new Location(name, lat, lon);
        StdOut.print("wonders[" + rank + "] == " + loc + "? ");
        StdOut.println(wonders[rank].equals(loc));
    }
}
```

Exercise 3. (*Comparable 3D Point*) Implement an immutable data type called `Point3D` that represents a point in 3D and supports the following API:

Point3D	
`Point3D(double x, double y, double z)`	constructs a point in 3D given its `x`, `y`, and `z` coordinates
`double distance(Point3D other)`	returns the Euclidean distance† between this point and `other`
`String toString()`	returns a string representation of this point
`int compareTo(Point3D other)`	returns a comparison of this point with `other` based on their respective distances to the origin $(0, 0, 0)$
`static Comparator<Point3D> xOrder()`	returns a comparator to compare two points by their x-coordinate
`static Comparator<Point3D> yOrder()`	returns a comparator to compare two points by their x-coordinate
`static Comparator<Point3D> zOrder()`	returns a comparator to compare two points by their x-coordinate

† The Euclidean distance between the points (x_1, y_1, z_1) and (x_2, y_2, z_2) is given by $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$.

>_ ~/workspace/project3

```
$ java Point3D
How many points? 3
Enter 9 doubles, separated by whitespace: -3 1 6 0 5 8 -5 -7 -3
Here are the points in the order entered:
  (-3.0, 1.0, 6.0)
  (0.0, 5.0, 8.0)
  (-5.0, -7.0, -3.0)
Sorted by their natural ordering (compareTo)
  (-3.0, 1.0, 6.0)
  (-5.0, -7.0, -3.0)
  (0.0, 5.0, 8.0)
Sorted by their x coordinate (xOrder)
  (-5.0, -7.0, -3.0)
  (-3.0, 1.0, 6.0)
  (0.0, 5.0, 8.0)
Sorted by their y coordinate (yOrder)
  (-5.0, -7.0, -3.0)
  (-3.0, 1.0, 6.0)
  (0.0, 5.0, 8.0)
Sorted by their z coordinate (zOrder)
  (-5.0, -7.0, -3.0)
  (-3.0, 1.0, 6.0)
  (0.0, 5.0, 8.0)
```

Point3D.java

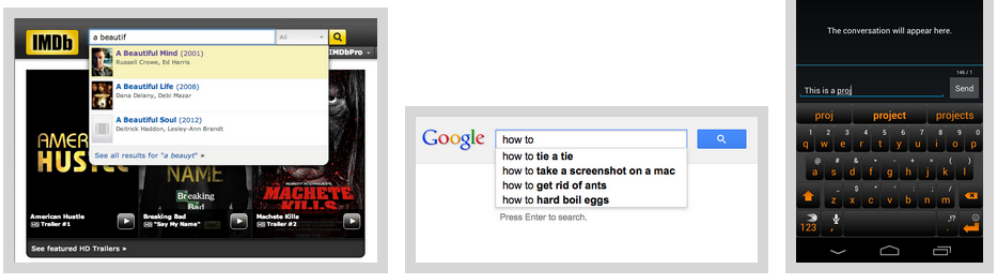
```
import java.util.Arrays;
import java.util.Comparator;

import stdlib.StdIn;
import stdlib.StdOut;

public class Point3D implements Comparable<Point3D> {
    private double x; // x coordinate
    private double y; // y coordinate
    private double z; // z coordinate

    // Constructs a point in 3D given its x, y, and z coordinates.
    public Point3D(double x, double y, double z) {
        ...
    }

    // Returns the Euclidean distance between this point and other.
    public double distance(Point3D other) {
        ...
    }

    // Returns a string representation of this point.
    public String toString() {
        ...
    }

    // Returns a comparison of this point with other based on their respective distances to the
    // origin (0, 0, 0).
    public int compareTo(Point3D other) {
        ...
    }

    // Returns a comparator to compare two points by their x-coordinate.
    public static Comparator<Point3D> xOrder() {
        ...
    }

    // Returns a comparator to compare two points by their y-coordinate.
    public static Comparator<Point3D> yOrder() {
        ...
    }

    // Returns a comparator to compare two points by their z-coordinate.
    public static Comparator<Point3D> zOrder() {
        ...
    }

    // A comparator for comparing two points by their x-coordinate.
    private static class XOrder implements Comparator<Point3D> {
        // Returns a comparison of p1 and p2 by their x-coordinate.
```

```
        public int compare(Point3D p1, Point3D p2) {
            ...
        }
    }

    // A comparator for comparing two points by their y-coordinate.
    private static class YOrder implements Comparator<Point3D> {
        // Returns a comparison of p1 and p2 by their y-coordinate.
        public int compare(Point3D p1, Point3D p2) {
            ...
        }
    }

    // A comparator for comparing two points by their z-coordinate.
    private static class ZOrder implements Comparator<Point3D> {
        // Returns a comparison of p1 and p2 by their z-coordinate.
        public int compare(Point3D p1, Point3D p2) {
            ...
        }
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        StdOut.print("How many points? ");
        int n = StdIn.readInt();
        Point3D[] points = new Point3D[n];
        StdOut.printf("Enter %d doubles, separated by whitespace: ", n * 3);
        for (int i = 0; i < n; i++) {
            double x = StdIn.readDouble();
            double y = StdIn.readDouble();
            double z = StdIn.readDouble();
            points[i] = new Point3D(x, y, z);
        }
        StdOut.println("Here are the points in the order entered:");
        for (Point3D point : points) {
            StdOut.println("  " + point);
        }
        Arrays.sort(points);
        StdOut.println("Sorted by their natural ordering (compareTo)");
        for (Point3D point : points) {
            StdOut.println("  " + point);
        }
        Arrays.sort(points, Point3D.xOrder());
        StdOut.println("Sorted by their x coordinate (xOrder)");
        for (Point3D point : points) {
            StdOut.println("  " + point);
        }
        Arrays.sort(points, Point3D.yOrder());
        StdOut.println("Sorted by their y coordinate (yOrder)");
        for (Point3D point : points) {
            StdOut.println("  " + point);
        }
        Arrays.sort(points, Point3D.zOrder());
        StdOut.println("Sorted by their z coordinate (zOrder)");
        for (Point3D point : points) {
            StdOut.println("  " + point);
        }
    }
}
```

Problems

Goal The purpose of this assignment is to write a program to implement *autocomplete* for a given set of n strings and nonnegative weights. That is, given a prefix, find all strings in the set that start with the prefix, in descending order of weight.

Autocomplete is an important feature of many modern applications. As the user types, the program predicts the complete *query* (typically a word or phrase) that the user intends to type. Autocomplete is most effective when there are a limited number of likely queries. For example, the Internet Movie Database uses it to display the names of movies as the user types; search engines use it to display suggestions as the user enters web search queries; cell phones use it to speed up text input.



In these examples, the application predicts how likely it is that the user is typing each query and presents to the user a list of the top-matching queries, in descending order of weight. These weights are determined by historical data, such as box office revenue for movies, frequencies of search queries from other Google users, or the typing history of a cell phone user. For the purposes of this assignment, you will have access to a set of all possible queries and associated weights (and these queries and weights will not change).

The performance of autocomplete functionality is critical in many systems. For example, consider a search engine which runs an autocomplete application on a server farm. According to one study, the application has only about 50ms to return a list of suggestions for it to be useful to the user. Moreover, in principle, it must perform this computation *for every keystroke typed into the search bar* and *for every user*!

In this assignment, you will implement autocomplete by sorting the queries in lexicographic order; using binary search to find the set of queries that start with a given prefix; and sorting the matching queries in descending order by weight.

Problem 1. (*Autocomplete Term*) Implement an immutable comparable data type called `Term` that represents an autocomplete term: a string query and an associated real-valued weight. You must implement the following API, which supports comparing terms by three different orders: lexicographic order by query; in descending order by weight; and lexicographic order by query but using only the first r characters. The last order may seem a bit odd, but you will use it in Problem 3 to find all terms that start with a given prefix (of length r).

Term	
`Term(String query)`	constructs a term given the associated query string, having weight 0
`Term(String query, long weight)`	constructs a term given the associated query string and weight
`String toString()`	returns a string representation of this term
`int compareTo(Term that)`	returns a comparison of this term and `other` by query
`static Comparator<Term> byReverseWeightOrder()`	returns a comparator for comparing two terms in reverse order of their weights
`static Comparator<Term> byPrefixOrder(int r)`	returns a comparator for comparing two terms by their prefixes of length `r`

Corner Cases

- The constructor should throw a `NullPointerException("query is null")` if *query* is `null` and an `IllegalArgumentException("Illegal weight")` if $weight < 0$.
- The `byPrefixOrder()` method should throw an `IllegalArgumentException("Illegal r")` if $r < 0$.

Performance Requirements

- The string comparison methods should run in time $T(n) \sim n$, where n is number of characters needed to resolve the comparison.

```
>_ ~/workspace/project3
$ java Term data/baby-names.txt 5
Top 5 by lexicographic order:
11      Aaban
5       Aabha
11      Aadam
11      Aadan
12      Aadarsh
Top 5 by reverse-weight order:
22175   Sophia
20811   Emma
18949   Isabella
18936   Mason
18925   Jacob
```

Directions:

- Instance variables:
  - Query string, `String query`.
  - Query weight, `long weight`.
- `Term(String query)` and `Term(String query, long weight)`
  - Initialize instance variables to appropriate values.
- `String toString()`
  - Return a string containing the weight and query separated by a tab.
- `int compareTo(Term other)`
  - Return a negative, zero, or positive integer based on whether `this.query` is less than, equal to, or greater than `other.query`.
- `static Comparator<Term> byReverseWeightOrder()`
  - Return an object of type `ReverseWeightOrder`.
- `static Comparator<Term> byPrefixOrder(int r)`
  - Return an object of type `PrefixOrder`.
- `Term :: ReverseWeightOrder`
  - `int compare(Term v, Term w)`
    - Return a negative, zero, or positive integer based on whether `v.weight` is less than, equal to, or greater than `w.weight`.
- `Term :: PrefixOrder`
  - Instance variable:
    - Prefix length, `int r`.
  - `PrefixOrder(int r)`
    - Initialize instance variable appropriately.
  - `int compare(Term v, Term w)`
    - Return a negative, zero, or positive integer based on whether `a` is less than, equal to, or greater than `b`, where `a` is a substring of `v` of length `min(r, v.query.length())` and `b` is a substring of `w` of length `min(r, w.query.length())`.

Problem 2. (*Binary Search Deluxe*) When binary searching a sorted array that contains more than one key equal to the search key, the client may want to know the index of either the first or the last such key. Accordingly, implement a library called `BinarySearchDeluxe` with the following API:

BinarySearchDeluxe	
`static int firstIndexOf(Key[] a, Key key, Comparator<Key> c)`	returns the index of the first key in `a` that equals the search `key`, or -1, according to the order induced by the comparator `c`
`static int lastIndexOf(Key[] a, Key key, Comparator<Key> c)`	returns the index of the last key in `a` that equals the search `key`, or -1, according to the order induced by the comparator `c`

Corner Cases

- Each method should throw a `NullPointerException("a, key, or c is null")` if any of the arguments is `null`. You may assume that the array `a` is sorted (with respect to the comparator `c`).

Performance Requirements

- Each method should should run in time $T(n) \sim \log n$, where n is the length of the array `a`.

```
>_ ~/workspace/project3
$ java BinarySearchDeluxe data/wiktionary.txt love
firstIndexOf(love) = 5318
lastIndexOf(love)  = 5324
frequency(love)    = 7
$ java BinarySearchDeluxe data/wiktionary.txt coffee
firstIndexOf(coffee) = 1628
lastIndexOf(coffee)  = 1628
frequency(coffee)    = 1
$ java BinarySearchDeluxe data/wiktionary.txt java
firstIndexOf(java) = -1
lastIndexOf(java)  = -1
frequency(java)    = 0
```

Directions:

- `static int firstIndexOf(Key[] a, Key key, Comparator<Key> c)`
  - Modify the standard binary search such that when `a[mid]` matches `key`, instead of returning `mid`, remember it in, say `index` (initialized to -1), and adjust `hi` appropriately.
  - Return `index`.
- `static int lastIndexOf(Key[] a, Key key, Comparator<Key> c)` can be implemented similarly.

Problem 3. (*Autocomplete*) In this part, you will implement a data type that provides autocomplete functionality for a given set of string and weights, using `Term` and `BinarySearchDeluxe`. To do so, *sort* the terms in lexicographic order; use *binary search* to find the set of terms that start with a given prefix; and sort the matching terms in descending order by weight. Organize your program by creating an immutable data type called `Autocomplete` with the following API:

Autocomplete	
`Autocomplete(Term[] terms)`	constructs an autocomplete data structure from an array of `terms`
`Term[] allMatches(String prefix)`	returns all terms that start with `prefix`, in descending order of their weights.
`int numberOfMatches(String prefix)`	returns the number of terms that start with `prefix`

Corner Cases

- The constructor should throw a `NullPointerException("terms is null")` if *terms* is `null`.
- Each method should throw a `NullPointerException("prefix is null)"` if $prefix$ is `null`.

Performance Requirements

- The constructor should run in time $T(n) \sim n \log n$, where n is the number of terms.
- The `allMatches()` method should run in time $T(n) \sim \log n + m \log m$, where m is the number of matching terms.
- The `numberOfMatches()` method should run in time $T(n) \sim \log n$.

```
>_ ~/workspace/project3
$ java Autocomplete data/wiktionary.txt 5
Enter a prefix (or ctrl-d to quit): love
First 5 matches for "love", in descending order by weight:
  49649600      love
  12014500      loved
  5367370       lovely
  4406690       lover
  3641430       loves
Enter a prefix (or ctrl-d to quit): coffee
All matches for "coffee", in descending order by weight:
  2979170       coffee
Enter a prefix (or ctrl-d to quit):
First 5 matches for "", in descending order by weight:
  5627187200    the
  3395006400    of
  2994418400    and
  2595609600    to
  1742063600    in
Enter a prefix (or ctrl-d to quit): <ctrl-d>
```

Directions:

- Instance variable:
  - Array of terms, `Term[] terms`.
- `Autocomplete(Term[] terms)`
  - Initialize `this.terms` to a defensive copy (ie, a fresh copy and not an alias) of. `terms`
  - Sort `this.terms` in lexicographic order.
- `Term[] allMatches(String prefix)`
  - Find the index `i` of the first term in `terms` that starts with `prefix`.
  - Find the number of terms (say `n`) in `terms` that start with `prefix`.
  - Construct an array `matches` containing `n` elements from `terms`, starting at. index `i`
  - Sort `matches` in reverse order of weight and return the sorted array.
- `int numberOfMatches(String prefix)`
  - Find the indices `i` and `j` of the first and last term in `terms` that start with `prefix`.
  - Using the indices, compute the number of terms that start with `prefix`, and return that value.

Data The `data` directory contains sample input files for testing. For example

```
>_ ~/workspace/project3
$ more data/wiktionary.txt
10000
    5627187200  the
    3395006400  of
           ...  ...
        392402  wench
        392323  calves
```

The first line specifies the number of terms and the following lines specify the weight and query string for each of those terms.

Visualization Program The program `AutocompleteVisualizer` accepts the name of a file and an integer k as command-line arguments, provides a GUI for the user to enter queries, and presents the top k matching terms in real time.

```
>_ ~/workspace/project3
$ java AutocompleteVisualizer data/wiktionary.txt 5
```

Acknowledgements This project is an adaptation of the Autocomplete Me assignment developed at Princeton University by Matthew Drabick and Kevin Wayne.

Files to Submit

1. `Die.java`
2. `Location.java`
3. `Point3D.java`
4. `Term.java`
5. `BinarySearchDeluxe.java`
6. `Autocomplete.java`
7. `report.txt`

Before you submit your files, make sure:

- Your programs meet the style requirements by running the following command in the terminal.

```
>_ ~/workspace/project3
$ check_style src/*.java
```

- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You use the template file `report.txt` for your report.
- Your report meets the prescribed guidelines.

Exercises

Exercise 1. (*Certify Heap*) Implement the static method `isMaxHeap()` in `CertifyHeap.java` that takes an array `a` of `Comparable` objects (excluding `a[0] = *`) and returns `true` if `a` represents a max-heap, and `false` otherwise.

>_ ~/workspace/project4

```
$ java CertifyHeap
* M A X H E A P
<ctrl-d>
false
$ java CertifyHeap
<ctrl-d>
* A A E H M P X
false
$ java CertifyHeap
<ctrl-d>
* X P M H E A A
true
```

CertifyHeap.java

```
import stdlib.StdIn;
import stdlib.StdOut;

public class CertifyHeap {
    // Returns true if a[] represents a max-heap, and false otherwise.
    public static boolean isMaxHeap(Comparable[] a) {
        // Set n to the number of elements in a.
        ...

        // For each node 1 <= i <= n / 2, if a[i] is less than either of its children, return
        // false, meaning a[] does not represent a max-heap. If no such i exists, return true.
        ...
    }

    // Returns true of v is less than w, and false otherwise.
    private static boolean less(Comparable v, Comparable w) {
        return (v.compareTo(w) < 0);
    }

    // Unit tests the library. [DO NOT EDIT]
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        StdOut.println(isMaxHeap(a));
    }
}
```

Exercise 2. (*Ramanujan's Taxi*) Srinivasa Ramanujan was an Indian mathematician who became famous for his intuition for numbers. When the English mathematician G. H. Hardy came to visit him one day, Hardy remarked that the number of his taxi was 1729, a rather dull number. To which Ramanujan replied, "No, Hardy! It is a very interesting number. It is the smallest number expressible as the sum of two cubes in two different ways." Verify this claim by writing a program `Ramanujan1.java` that accepts n (int) as command-line argument and writes to standard output all integers less than or equal to n that can be expressed as the sum of two cubes in two different ways. In other words, find distinct positive integers a, b, c, and d such that $a^3 + b^3 = c^3 + d^3 \leq n$.

>_ ~/workspace/project4

```
$ java Ramanujan1 10000
1729 = 1^3 + 12^3 = 9^3 + 10^3
4104 = 2^3 + 16^3 = 9^3 + 15^3
```

Ramanujan1.java

```
import stdlib.StdOut;

public class Ramanujan1 {
    // Entry point.
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        ...
    }
}
```

Directions:

- Use four nested `for` loops, with these bounds on the loop variables: $0 < a \leq \sqrt[3]{n}$, $a < b \leq \sqrt[3]{n - a^3}$, $a < c \leq \sqrt[3]{n}$, and $c < d \leq \sqrt[3]{n - c^3}$

Do not explicitly compute cube roots, and instead use `x * x * x < y` in place of `x < Math.cbrt(y)`.

Exercise 3. (*Ramanujan's Taxi Redux*) Write a program `Ramanujan2.java` that uses a minimum-oriented priority queue to solve the problem from Exercise 2.

```
>_ ~/workspace/project4
$ java Ramanujan2 10000
1729 = 1^3 + 12^3 = 9^3 + 10^3
4104 = 9^3 + 15^3 = 2^3 + 16^3
```

Ramanujan2.java

```
import dsa.MinPQ;
import stdlib.StdOut;

public class Ramanujan2 {
    // Entry point.
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        ...
    }

    // A data type that encapsulates a pair of numbers (i, j) and the sum of their cubes.
    private static class Pair implements Comparable<Pair> {
        private int i;          // first number in the pair
        private int j;          // second number in the pair
        private int sumOfCubes; // i^3 + j^3

        // Constructs a pair (i, j).
        public Pair(int i, int j) {
            this.i = i;
            this.j = j;
            sumOfCubes = i * i * i + j * j * j;
        }

        // Returns a comparison of pairs this and other based on their sum-of-cubes values.
        public int compareTo(Pair other) {
            return sumOfCubes - other.sumOfCubes;
        }
    }
}
```

Directions:

- Initialize a min-PQ `pq` with pairs $(1, 2), (2, 3), (3, 4), \ldots, (i, i + 1)$, where $i < \sqrt[3]{n}$
- While i is not empty:
  - Remove the smallest pair (call it *current*) from `pq`.
  - Print the previous pair (k, l) and current pair (i, j) if $k^3 + l^3 = i^3 + j^3 \leq n$.
  - If $i < \sqrt[3]{n}$, insert the pair $(i, j + 1)$ into `pq`.

Again, do not explicitly compute cube roots, and instead use `x * x * x < y` in place of `x < Math.cbrt(y)`.

Problems

Goal The purpose of this project is to write a program to solve the 8-puzzle problem (and its natural generalizations) using the $A^\star$ 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. 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 position (left) to the goal position (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                                                     goal
```

Best-First Search Now, we describe a solution to the problem that illustrates a general artificial intelligence methodology known as the $A^\star$ 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 sum of the Hamming distance (number of tiles in the wrong position), plus the number of moves made so far to get to the search node. Intuitively, a search node with a small number of tiles 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 distance (sum of the vertical and horizontal distance) from the tiles 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
```

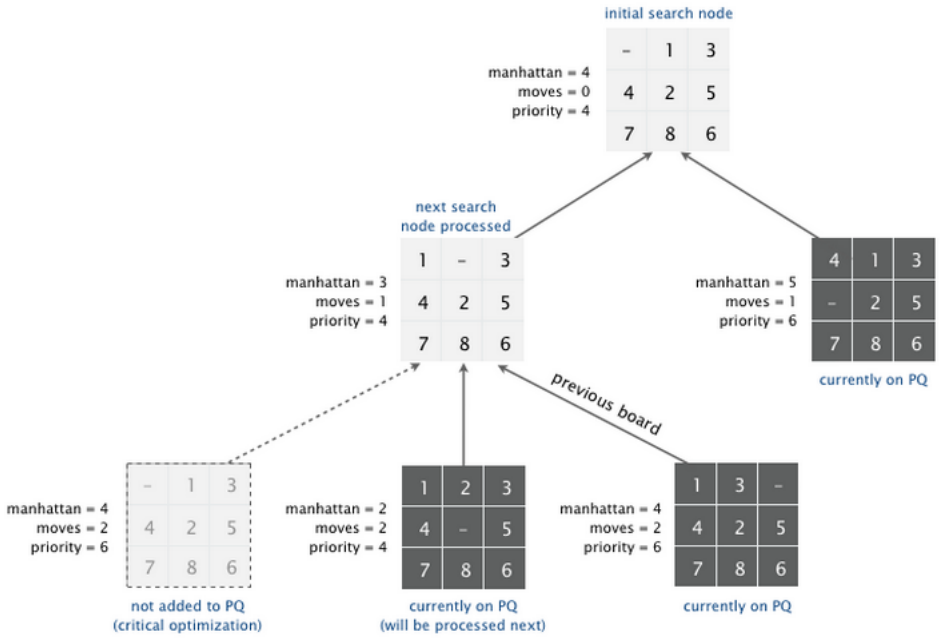
We make a key observation: To solve the puzzle from a given search node on the priority queue, the total number of moves we need to make (including those already made) is at least its priority, using either the Hamming or Manhattan priority function. (For Hamming priority, this is true because each tile that is out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each tile 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)
```

A Second Optimization To avoid recomputing the Hamming/Manhattan distance of a board (or, alternatively, the Hamming/Manhattan priority of a solver node) from scratch each time during various priority queue operations, compute it at most once per object; save its value in an instance variable; and return the saved value as needed. This caching technique is broadly applicable: consider using it in any situation where you are recomputing the same quantity many times and for which computing that quantity is a bottleneck operation.

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 Unsolvable Puzzles Not all initial boards can lead to the goal board by a sequence of legal moves, including the two below:

```
 1  2  3          1  2  3  4
 4  5  6          5  6  7  8
 8  7             9 10 11 12
                 13 15 14
```

To detect such situations, use the fact that boards are divided into two equivalence classes with respect to reachability: those that lead to the goal board; and those that cannot lead to the goal board. Moreover, we can identify in which equivalence class a board belongs without attempting to solve it.

- *Odd board size.* Given a board, an *inversion* is any pair of tiles i and j where $i < j$ but i appears after j when considering the board in row-major order (row 0, followed by row 1, and so forth).

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

row-major order: 1 2 3 4 5 6 8 7  1 2 3 4 5 6 8 7  1 2 3 4 6 8 5 7  1 2 3 4 6 8 5 7  1 2 3 8 4 6 5 7

                 inversions = 1   inversions = 1    inversions = 3    inversions = 3    inversions = 5
                 (8-7)            (8-7)            (6-5 8-5 8-7)     (6-5 8-5 8-7)     (8-4 8-6 8-5 8-7 6-5)
```

If the board size n is an odd integer, then each legal move changes the number of inversions by an even number. Thus, if a board has an odd number of inversions, then it cannot lead to the goal board by a sequence of legal moves because the goal board has an even number of inversions (zero).

The converse is also true: if a board has an even number of inversions, then it can lead to the goal board by a sequence of legal moves.

```
                    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

row-major order: 1 3 4 2 5 7 8 6    1 3 4 2 5 7 8 6    1 2 3 4 5 7 8 6    1 2 3 4 5 7 8 6    1 2 3 4 5 6 7 8

                 inversions = 4     inversions = 4     inversions = 2     inversions = 2     inversions = 0
                 (3-2 4-2 7-6 8-6)  (3-2 4-2 7-6 8-6)  (7-6 8-6)          (7-6 8-6)
```

- *Even board size.* If the board size n is an even integer, then the parity of the number of inversions is not invariant. However, the parity of the number of inversions plus the row of the blank square is invariant: each legal move changes this sum by an even number. If this sum is even, then it cannot lead to the goal board by a sequence of legal moves; if this sum is odd, then it can lead to the goal board by a sequence of legal moves.

```
    1  2  3  4          1  2  3  4          1  2  3  4          1  2  3  4          1  2  3  4
    5     6  8    =>    5  6     8    =>    5  6  7  8    =>    5  6  7  8    =>    5  6  7  8
    9 10  7 11          9 10  7 11          9 10    11          9 10 11             9 10 11 12
   13 14 15 12         13 14 15 12         13 14 15 12         13 14 15 12         13 14 15

  blank row = 1       blank row  = 1       blank row  = 2       blank row  = 2       blank row  = 3
 inversions = 6       inversions = 6       inversions = 3       inversions = 3       inversions = 0
 -------------        -------------        -------------        -------------        -------------
        sum = 7              sum = 7              sum = 5              sum = 5              sum = 3
```

Problem 1. (*Board Data Type*) Implement an immutable data type called `Board` to represent a board in an n-puzzle, supporting the following API:

Board	
`Board(int[][] tiles)`	constructs a board from an $n \times n$ array; `tiles[i][j]` is the tile at row i and column j, with 0 denoting the blank tile
`int size()`	returns the size of this board size
`int tileAt(int i, int j)`	returns the tile at row `i` and column `j`
`int hamming()`	returns Hamming distance between this board and the goal board
`int manhattan()`	returns the Manhattan distance between this board and the goal board
`boolean isGoal()`	returns `true` if this board is the goal board, and `false` otherwise
`boolean isSolvable()`	returns `true` if this board solvable, and `false` otherwise
`Iterable<Board> neighbors()`	returns an iterable object containing the neighboring boards of this board
`boolean equals(Object other)`	returns `true` if this board is the same as `other`, and `false` otherwise
`String toString()`	returns a string representation of this board

Performance Requirements

- The constructor should run in time $T(n) \sim n^2$, where n is the board size.
- The `size()`, `tileAt()`, `hamming()`, `manhattan()`, and `isGoal()` methods should run in time $T(n) \sim 1$.
- The `isSolvable()` method should run in time $T(n) \sim n^2 \log n^2$.
- The `neighbors()` and `equals()` methods should run in time $T(n) \sim n^2$.

```
>_ ~/workspace/project4
$ java Board data/puzzle05.txt
The board (3-puzzle):
 4  1  3
    2  6
 7  5  8
Hamming = 5, Manhattan = 5, Goal? false, Solvable? true
Neighboring boards:
 4  1  3
 7  2  6
    5  8
----------
```

```
    1  3
 4  2  6
 7  5  8
----------
 4  1  3
 2     6
 7  5  8
----------
$ java Board data/puzzle4x4-unsolvable1.txt
The board (4-puzzle):
 3  2  4  8
 1  6    12
 5 10  7 11
 9 13 14 15
Hamming = 12, Manhattan = 13, Goal? false, Solvable? false
Neighboring boards:
 3  2  4  8
 1  6  7 12
 5 10    11
 9 13 14 15
----------
 3  2     8
 1  6  4 12
 5 10  7 11
 9 13 14 15
----------
 3  2  4  8
 1  6 12
 5 10  7 11
 9 13 14 15
----------
 3  2  4  8
 1     6 12
 5 10  7 11
 9 13 14 15
----------
```

Before you write any code, make sure you thoroughly understand the concepts that are central to solving the 8-puzzle and its generalizations using the $A^\star$ algorithm. Compute the following for the two initial boards A and B shown below:

A

4	1	3
	2	6
7	5	8

B

1	2	3
4	6	5
7	8	

1. Hamming distance of the board to the goal board
2. Manhattan distance of the board to the goal board
3. Neighboring boards of the board
4. Row-major order of the board
5. Position of the blank tile (in row-major order) in the board
6. Number of inversions (excluding the blank tile) for the board
7. Is the board solvable? Explain why or why not
8. A shortest solution for the board, if one exists

Directions:

- Instance variables:
  - Tiles in the board, `int[][] tiles`.
  - Board size, `int n`.

  - Hamming distance to the goal board, `int hamming`.
  - Manhattan distance to the goal board, `int manhattan`.
  - Position of the blank tile in row-major order, `int blankPos`.

- `private int[][] cloneTiles()`
  - Return a defensive copy of the tiles of the board.
- `Board(int[][] tiles)`
  - Initialize the instance variables `this.tiles` and `n` to `tiles` and the number of rows in `tiles` respectively.
  - Compute the Hamming/Manhattan distances to the goal board and the position of the blank tile in row-major order, and store the values in the instance variables `hamming`, `manhattan`, and `blankPos` respectively.
- `int size()`
  - Return the board size.
- `int tileAt(int i, int j)`
  - Return the tile at row `i` and column `j`.
- `int hamming()`
  - Return the Hamming distance to the goal board.
- `int manhattan()`
  - Return the Manhattan distance to the goal board.
- `boolean isGoal()`
  - Return `true` if the board is the goal board, and `false` otherwise.
- `boolean isSolvable()`
  - Create an array of size $n^2 - 1$ containing the tiles (excluding the blank tile) of the board in row-major order.
  - Use `Inversions.count()` to compute the number of inversions in the array.
  - From the number of inversions, compute and return whether the board is solvable.
- `Iterable<Board> neighbors()`
  - Create a queue `q` of `Board` objects.
  - For each possible neighbor of the board (determined by the blank tile position):
    * Clone the tiles of the board.
    * Exchange an appropriate tile with the blank tile in the clone.
    * Construct a `Board` object from the clone, and enqueue it into `q`.
  - Return `q`.
- `boolean equals(Board other)`
  - Return `true` if the board is the same as `other`, and `false` otherwise.

Problem 2. (*Solver Data Type*) Implement an immutable data type called `Solver` that uses the $A^\star$ algorithm to solve the 8-puzzle and its generalizations. The data type should support the following API:

Solver	
`Solver(Board board)`	finds a solution to the initial `board` using the $A^\star$ algorithm
`int moves()`	returns the minimum number of moves needed to solve the initial board
`Iterable<Board> solution()`	returns a sequence of boards in a shortest solution of the initial board

Corner Cases

- The constructor should throw a `NullPointerException("board is null")` if *board* is `null` and an `IllegalArgumentException("board is unsolvable")` if *board* is unsolvable.

```
>_ ~/workspace/project4
$ java Solver data/puzzle05.txt
Solution (5 moves):
 4  1  3
    2  6
 7  5  8
----------
    1  3
 4  2  6
 7  5  8
----------
 1     3
 4  2  6
 7  5  8
----------
 1  2  3
 4     6
 7  5  8
----------
 1  2  3
 4  5  6
 7     8
----------
 1  2  3
 4  5  6
 7  8
----------
$ java Solver data/puzzle4x4-unsolvable1.txt
Unsolvable puzzle
```

Directions:

- Instance variables:
  - Minimum number of moves needed to solve the initial board, `int moves`.
  - Sequence of boards in a shortest solution of the initial board, `LinkedStack<Board> solution`.
- `Solver :: SearchNode` (represents a node in the game tree)
  - Instsance variables:
    - The board represented by this node, `Board board`.
    - Number of moves it took to get to this node from the initial node, `int moves`.
    - The previous search node, `SearchNode previous`.
  - `SearchNode(Board board, int moves, SearchNode previous)`
    - Initialize instance variables appropriately.
  - `int compareTo(SearchNode other)`
    - Return a comparison of the search node with other , based on the sum: Manhattan distance of the board in the node plus the number of moves to the node (from the initial search node).
- `Solver(Board initial)`
  - Create a `MinPQ<SearchNode>` object `pq` and insert the initial search node into it
  - As long as `pq` is not empty:

  - Remove the smallest node (call it `node`) from `pq`.
  - If the board in `node` is the goal board, extract from the node the number of moves in the solution and the solution and store the values in the instance variables `moves` and `solution` respectively, and break.
  - Otherwise, iterate over the neighboring boards of `node.board`, and for each neighbor that is different from `node.previous.board`, insert a new `SearchNode` object into `pq`, constructed using appropriate values.

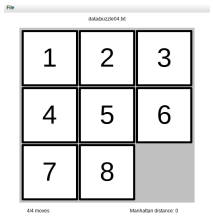
- `int moves()`
  - Return the minimum number of moves needed to solve the initial board.
- `Iterable<Board> solution()`
  - Return the sequence of boards in a shortest solution of the initial board.

Data and Test Programs The `data` directory contains a number of sample input files representing boards of different sizes; for example

```
>_ ~/workspace/project4
$ more data/puzzle04.txt
3
 0  1  3
 4  2  5
 7  8  6
```

The program `SolverVisualizer` accepts the name of an input file as command-line argument, and using your `Board` and `Solver` data types graphically solves the sliding block puzzle defined by the file

```
>_ ~/workspace/project4
$ java SolverVisualizer data/puzzle04.txt
```



The program `PuzzleChecker` accepts the names of an input files as command-line arguments, creates an initial board from each file, and writes to standard output: the filename, minimum number of moves to reach the goal board from the initial board, and the time (in secs) taken; if the initial board is unsolvable, a "–" is written for the number of moves and time taken

```
>_ ~/workspace/project4
$ java PuzzleChecker data/puzzle*.txt
filename                                moves     time
---------------------------------------------------------
data/puzzle00.txt                           0     0.00
data/puzzle01.txt                           1     0.00
data/puzzle02.txt                           2     0.00
data/puzzle03.txt                           3     0.00
data/puzzle04.txt                           4     0.00
data/puzzle05.txt                           5     0.00
...
data/puzzle47.txt                          47     9.41
data/puzzle48.txt                          48     2.13
data/puzzle49.txt                          49    19.63
data/puzzle4x4-unsolvable1.txt             --       --
data/puzzle50.txt                          50    12.31
```

Acknowledgements This project is an adaptation of the 8 Puzzle assignment developed at Princeton University by Robert Sedgewick and Kevin Wayne.

Files to Submit

1. `CertifyHeap.java`
2. `Ramanujan1.java`
3. `Ramanujan2.java`
4. `Board.java`
5. `Solver.java`
6. `report.txt`

Before you submit your files, make sure:

- Your programs meet the style requirements by running the following command in the terminal.

```
>_ ~/workspace/project4
$ check_style src/*.java
```

- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You use the template file `report.txt` for your report.
- Your report meets the prescribed guidelines.

Exercises

Exercise 1. (*Array-based Symbol Table*) Implement a data type called `ArrayST` that uses an unordered array as the underlying data structure to implement the basic symbol table API.

>_ ~/workspace/project5

```
$ java ArrayST < data/tinyST.txt
S 0
E 12
A 8
R 3
C 4
H 5
X 7
M 9
P 10
L 11
```

ArrayST.java

```
import dsa.BasicST;
import dsa.LinkedQueue;
import stdlib.StdIn;
import stdlib.StdOut;

public class ArrayST<Key, Value> implements BasicST<Key, Value> {
    private Key[] keys;     // keys in the symbol table
    private Value[] values; // the corresponding values
    private int n;          // number of key-value pairs

    // Constructs an empty symbol table.
    public ArrayST() {
        ...
    }

    // Returns true if this symbol table is empty, and false otherwise.
    public boolean isEmpty() {
        ...
    }

    // Returns the number of key-value pairs in this symbol table.
    public int size() {
        ...
    }

    // Inserts the key and value pair into this symbol table.
    public void put(Key key, Value value) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        if (value == null) {
            throw new IllegalArgumentException("value is null");
        }
        ...
    }

    // Returns the value associated with key in this symbol table, or null.
    public Value get(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        ...
    }

    // Returns true if this symbol table contains key, and false otherwise.
    public boolean contains(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        ...
    }

    // Deletes key and the associated value from this symbol table.
    public void delete(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        ...
```

```
    }

    // Returns all the keys in this symbol table.
    public Iterable<Key> keys() {
        ...
    }

    // Resizes the underlying arrays to capacity.
    private void resize(int capacity) {
        Key[] tempk = (Key[]) new Object[capacity];
        Value[] tempv = (Value[]) new Object[capacity];
        for (int i = 0; i < n; i++) {
            tempk[i] = keys[i];
            tempv[i] = values[i];
        }
        values = tempv;
        keys = tempk;
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        ArrayST<String, Integer> st = new ArrayST<>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys()) {
            StdOut.println(s + " " + st.get(s));
        }
    }
}
```

Exercise 2. (*Spell Checker*) Implement a program called `Spell` that accepts *filename* (String) as command-line argument, which is the name of a file containing common misspellings (a line-oriented file with each comma-separated line containing a misspelled word and the correct spelling); reads text from standard input; and writes to standard output the misspelled words in the text, the line numbers where they occurred, and their corrections.

>_ ~/workspace/project5

```
$ java Spell data/misspellings.txt < data/war_and_peace.txt
wont:5370 -> won't
unconciousness:16122 -> unconsciousness
accidently:18948 -> accidentally
leaded:21907 -> led
wont:22062 -> won't
aquaintance:30601 -> acquaintance
wont:39087 -> won't
wont:50591 -> won't
planed:53591 -> planned
wont:53960 -> won't
Ukranian:58064 -> Ukrainian
wont:59650 -> won't
conciousness:59835 -> consciousness
occuring:59928 -> occurring
```

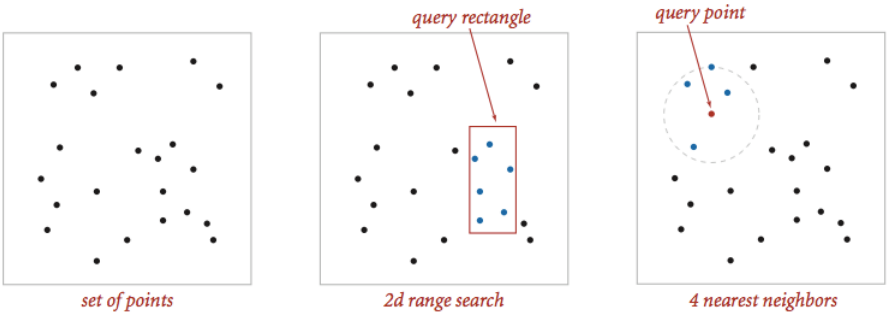
Spell.java

```
import stdlib.In;
import stdlib.StdIn;
import stdlib.StdOut;

public class Spell {
    // Entry point.
    public static void main(String[] args) {
        In in = new In(args[0]);
        String[] lines = in.readAllLines();
        in.close();

        // Create an ArrayST<String, String> object called st.
        ...

        // For each line in lines, split it into two tokens using "," as delimiter; insert into
        // st the key-value pair (token 1, token 2).
        ...

        // Read from standard input one line at a time; increment a line number counter; split
```

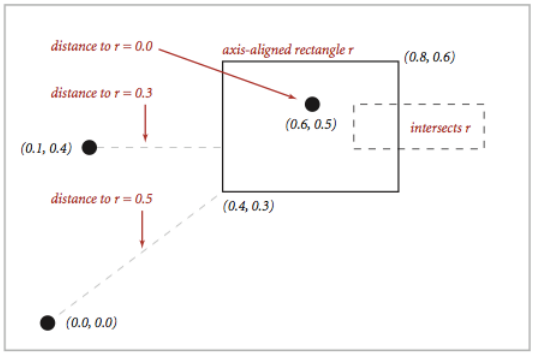
```
        // the line into words using "\\b" as the delimiter; for each word in words, if it
        // exists in st, write the (misspelled) word, its line number, and corresponding value
        // (correct spelling) from st.
        ...
    }
}
```

Problems

The purpose of this project is to create a symbol table data type whose keys are two-dimensional points. We'll use a 2dTree to support efficient range search (find all the points contained in a query rectangle) and k-nearest neighbor search (find k points that are closest to a query point). 2dTrees have numerous applications, ranging from classifying astronomical objects to computer animation to speeding up neural networks to mining data to image retrieval.



Geometric Primitives We will use the data types `dsa.Point2D` and `dsa.RectHV` to represent points and axis-aligned rectangles in the plane.



Symbol Table API Here is a Java interface `PointST<Value>` specifying the API for a symbol table data type whose keys are `Point2D` objects and values are generic objects:

PointST<Value>	
`boolean isEmpty()`	returns `true` if this symbol table is empty, and `false` otherwise
`int size()`	returns the number of key-value pairs in this symbol table
`void put(Point2D p, Value value)`	inserts the given point and value into this symbol table
`Value get(Point2D p)`	returns the value associated with the given point in this symbol table, or `null`
`boolean contains(Point2D p)`	returns `true` if this symbol table contains the given point, and `false` otherwise
`Iterable<Point2D> points()`	returns all the points in this symbol table
`Iterable<Point2D> range(RectHV rect)`	returns all the points in this symbol table that are inside the given rectangle
`Point2D nearest(Point2D p)`	returns the point in this symbol table that is different from and closest to the given point, or `null`
`Iterable<Point2D> nearest(Point2D p, int k)`	returns up to `k` points from this symbol table that are different from and closest to the given point

Problem 1. (*Brute-force Implementation*) Develop a data type called `BrutePointST` that implements the above API using a red-black BST (`RedBlackBST`) as the underlying data structure.

`BrutePointST<Value> implements PointST<Value>`	
`BrutePointST()`	constructs an empty symbol table

Corner Cases

- The `put()` method should throw a `NullPointerException()` with the message `"p is null"` if p is `null` and the message `"value is null"` if *value* is `null`.
- The `get()`, `contains()`, and `nearest()` methods should throw a `NullPointerException()` with the message `"p is null"` if p is `null`.
- The `rect()` method should throw a `NullPointerException()` with the message `"rect is null"` if *rect* is `null`.

Performance Requirements

- The `isEmpty()` and `size()` methods should run in time $T(n) \sim 1$, where n is the number of key-value pairs in the symbol table.
- The `put()`, `get()`, and `contains()` methods should run in time $T(n) \sim \log n$.
- The `points()`, `range()`, and `nearest()` methods should run in time $T(n) \sim n$.

```
>_ ~/workspace/project5
$ java BrutePointST 0.975528 0.345492 5 < data/circle10.txt
st.size() = 10
st.contains((0.975528, 0.345492))? true
st.range([-1.0, -1.0] x [1.0, 1.0]):
  (0.5, 0.0)
  (0.206107, 0.095492)
  (0.793893, 0.095492)
  (0.024472, 0.345492)
  (0.975528, 0.345492)
  (0.024472, 0.654508)
  (0.975528, 0.654508)
  (0.206107, 0.904508)
  (0.793893, 0.904508)
  (0.5, 1.0)
st.nearest((0.975528, 0.345492)) = (0.975528, 0.654508)
st.nearest((0.975528, 0.345492), 5):
  (0.975528, 0.654508)
  (0.793893, 0.095492)
  (0.793893, 0.904508)
  (0.5, 0.0)
  (0.5, 1.0)
```

Directions:

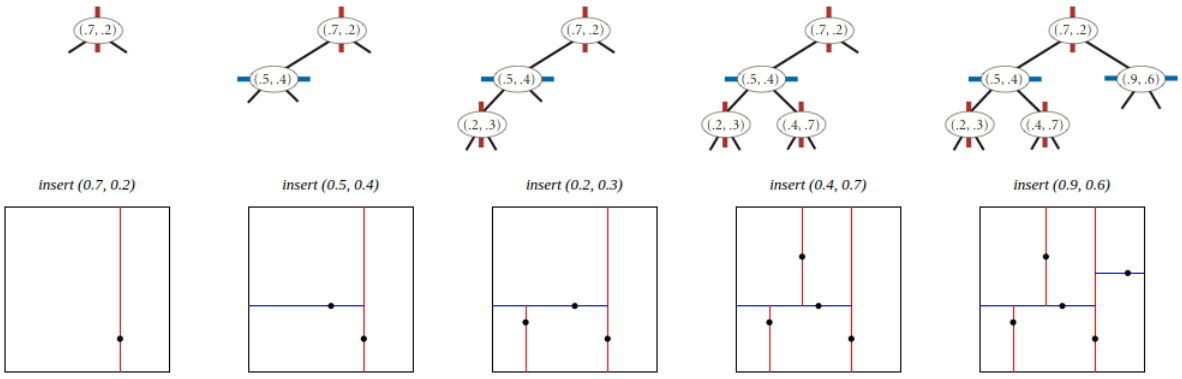
- Instance variable:
  - An underlying data structure to store the 2d points (keys) and their corresponding values, `RedBlackBST<Point2D, Value> bst`.
- `BrutePointST()`
  - Initialize the instance variable `bst` appropriately.
- `int size()`
  - Return the size of `bst`.
- `boolean isEmpty()`
  - Return whether `bst` is empty.
- `void put(Point2D p, Value value)`
  - Insert the given point and value into `bst`.
- `Value get(Point2D p)`
  - Return the value associated with the given point in `bst`, or `null`.
- `boolean contains(Point2D p)`
  - Return whether `bst` contains the given point.
- `Iterable<Point2D> points()`
  - Return an iterable object containing all the points in `bst`.
- `Iterable<Point2D> range(RectHV rect)`
  - Return an iterable object containing all the points in `bst` that are inside the given rectangle.
- `Point2D nearest(Point2D p)`
  - Return a point from `bst` that is different from and closest to the given point, or `null`.
- `Iterable<Point2D> nearest(Point2D p, int k)`
  - Return up to `k` points from `bst` that are different from and closest to the given point.

Problem 2. (*2dTree Implementation*) Develop a data type called `KdTreePointST` that uses a 2dTree to implement the above symbol table API.

`KdTreePointST<Value> implements PointST<Value>`	
`KdTreePointST()`	constructs an empty symbol table

A 2dTree 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, starting with the x-coordinates.

- *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.



- *Level-order traversal.* The `points()` method should return the points in level-order: first the root, then all children of the root (from left/bottom to right/top), then all grandchildren of the root (from left to right), and so forth. The level-order traversal of the 2dTree above is (.7, .2), (.5, .4), (.9, .6), (.2, .3), (.4, .7).

The prime advantage of a 2dTree over a BST is that it supports efficient implementation of range search, nearest neighbor, and k-nearest neighbor search. Each node corresponds to an axis-aligned rectangle, which encloses all of the points in its subtree. The root corresponds to the infinitely large square from $[(-\infty, -\infty), (+\infty, +\infty)]$; the left and right children of the root correspond 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). That is, you should search a subtree 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, you should search a node 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 choose first the subtree that is on the same side of the splitting line as the query point; the closest point found while exploring the first subtree may enable pruning of the second subtree.
- *k-nearest neighbor search.* Use the nearest neighbor search described above.

Corner Cases

- The `put()` method should throw a `NullPointerException()` with the message `"p is null"` if p is `null` and the message `"value is null"` if *value* is `null`.
- The `get()`, `contains()`, and `nearest()` methods should throw a `NullPointerException()` with the message `"p is null"` if p is `null`.
- The `rect()` method should throw a `NullPointerException()` with the message `"rect is null"` if *rect* is `null`.

Performance Requirements

- The `isEmpty()` and `size()` methods should run in time $T(n) \sim 1$, where n is the number of key-value pairs in the symbol table.
- The `put()`, `get()`, `contains()`, `range()`, and `nearest()` methods should run in time $T(n) \sim \log n$.
- The `points()` method should run in time $T(n) \sim n$.

```
>_ ~/workspace/project5
$ java KdTreePointST 0.975528 0.345492 5 < data/circle10.txt
st.empty()? false
st.size() = 10
st.contains((0.975528, 0.345492))? true
st.range([-1.0, -1.0] x [1.0, 1.0]):
  (0.206107, 0.095492)
  (0.024472, 0.345492)
  (0.024472, 0.654508)
  (0.975528, 0.654508)
  (0.793893, 0.095492)
  (0.5, 0.0)
  (0.975528, 0.345492)
  (0.793893, 0.904508)
  (0.206107, 0.904508)
  (0.5, 1.0)
st.nearest((0.975528, 0.345492)) = (0.975528, 0.654508)
st.nearest((0.975528, 0.345492), 5):
  (0.5, 1.0)
  (0.5, 0.0)
  (0.793893, 0.904508)
  (0.793893, 0.095492)
  (0.975528, 0.654508)
```

Directions:

- Instance variables:
  - Reference to the root of a 2dTree, `Node root`.
  - Number of nodes in the tree, `int n`.
- `KdTreePointST()`
  - Initialize instance variables `root` and `n` appropriately.
- `int size()`
  - Return the number of nodes in the 2dTree.
- `boolean isEmpty()`
  - Return whether the 2dTree is empty.
- `void put(Point2D p, Value value)`
  - Call the private `put()` method with appropriate arguments to insert the given point and value into the 2dTree; the parameter `lr` in this and other helper methods represents if the currrent node is x-aligned (`lr = true`) or y-aligned (`lr = false`).
- `Node put(Node x, Point2D p, Value value, RectHV rect, boolean lr)`
  - If `x = null`, return a new `Node` object built appropriately.
  - If the point in `x` is the same as the given point, update the value in `x` to the given value.
  - Otherwise, make a recursive call to `put()` with appropriate arguments to insert the given point and value into the left subtree `x.lb` or the right subtree `x.rt` depending on how `x.p` and `p` compare (use `lr` to decide which coordinate to consider).
  - Return `x`.
- `Value get(Point2D p)`
  - Call the private `get()` method with appropriate arguments to find the value corresponding to the given point.
- `Value get(Node x, Point2D p, boolean lr)`
  - If `x = null`, return `null`.

  - If the point in `x` is the same as the given point, return the value in `x`.
  - Make a recursive call to `get()` with appropriate arguments to find the value corresponding to the given point in the left subtree `x.lb` or the right subtree `x.rt` depending on how `x.p` and `p` compare.

- `boolean contains(Point2D p)`
  - Return whether the given point is in the 2dTree.

- `Iterable<Point2D> points()`
  - Return all the points in the 2dTree, collected using a level-order traversal of the tree; use two queues, one to aid in the traversal and the other to collect the points.

- `Iterable<Point2D> range(RectHV rect)`
  - Call the private `range()` method with appropriate arguments, the last one being an empty queue of `Point2D` objects, and return the queue.

- `void range(Node x, RectHV rect, Queue<Point2D> q)`
  - If `x = null`, simply return.
  - If `rect` contains the point in `x`, enqueue the point into `q`
  - Make recursive calls to `range()` on the left subtree `x.lb` and on the right subtree `x.rt`.
  - Incorporate the *range search* pruning rule mentioned in the project writeup.

- `Point2D nearest(Point2D p)`
  - Return a point from the 2dTree that is different from and closest to the given point by calling the private method `nearest()` with appropriate arguments.

- `Point2D nearest(Node x, Point2D p, Point2D nearest, boolean lr)`
  - If `x = null`, return `nearest`.
  - If the point `x.p` is different from the given point `p` and the squared distance between the two is smaller than the squared distance between `nearest` and `p`, update `nearest` to `x.p`.
  - Make a recursive call to `nearest()` on the left subtree `x.lb`.
  - Make a recursive call to `nearest()` on the right subtree `x.rt`, using the value returned by the first call in an appropriate manner.
  - Incorporate the *nearest neighbor* pruning rules mentioned in the project writeup.

- `Iterable<Point2D> nearest(Point2D p, int k)`
  - Call the private `nearest()` method passing it an empty maxPQ of `Point2D` objects (built with a suitable comparator from `Point2D`) as one of the arguments, and return the PQ.

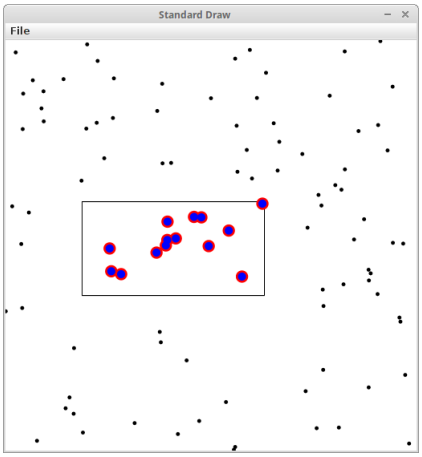
- `void nearest(Node x, Point2D p, int k, MaxPQ<Point2D> pq, boolean lr)`
  - If `x = null` or if the size of `pq` is greater than `k`, simply return.
  - If the point in `x` is different from the given point, insert it into `pq`.
  - If the size of `pq` exceeds `k`, remove the maximum point from the `pq`.
  - Make recursive calls to `nearest()` on the left subtree `x.lb` and on the right subtree `x.rt`.
  - Incorporate the *nearest neighbor* pruning rules mentioned in the project writeup.

Data The `data` directory contains a number of sample input files, each with n points (x, y), where $x, y \in (0, 1)$; for example

```
>_ ~/workspace/project5
$ cat data/input100.txt
0.042191 0.783317
0.390296 0.499816
0.666260 0.752352
...
0.263965 0.906869
0.564479 0.679364
0.772950 0.196867
```

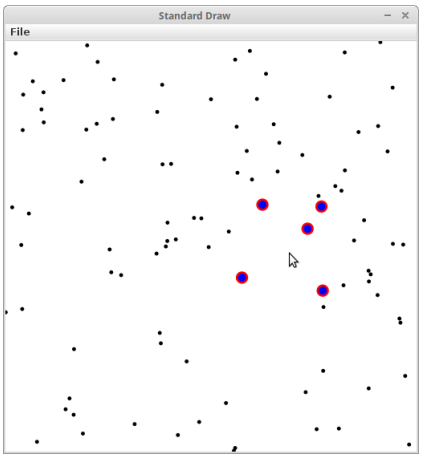
Visualization Programs The program `RangeSearchVisualizer` reads a sequence of points from a file (specified as a command-line argument), inserts those points into `BrutePointST` (red) and `KdTreePointST` (blue) based symbol tables, performs range searches on the axis-aligned rectangles dragged by the user, and displays the points obtained from the symbol tables in red and blue.

```
>_ ~/workspace/project5
$ java RangeSearchVisualizer data/input100.txt
```



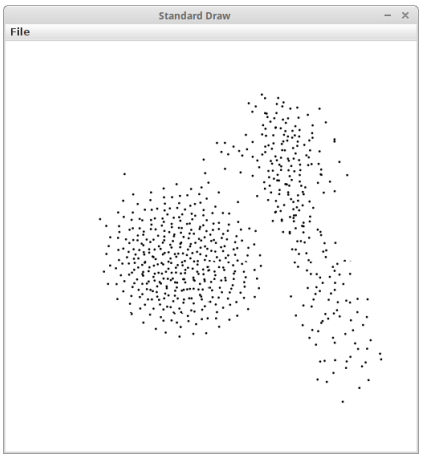
The program `NearestNeighborVisualizer` reads a sequence of points from a file (specified as the first command-line argument), inserts those points into `BrutePointST` (red) and `KdTreeSPointT` (blue) based symbol tables, performs k (specified as the second command-line argument) nearest-neighbor queries on the point corresponding to the location of the mouse, and displays the neighbors obtained from the symbol tables in red and blue.

```
>_ ~/workspace/project5
$ java NearestNeighborVisualizer data/input100.txt 5
```



The program `BoidSimulator` simulates the flocking behavior of birds, using a `BrutePointST` or `KdTreePointST` data type; the first command-line argument specifies which data type to use (`brute` or `kdtree`), the second argument specifies the number of boids, and the third argument specifies the number of friends each boid has.

```
>_ ~/workspace/project5
$ java BoidSimulator kdtree 1000 10
```



Acknowledgements This project is an adaptation of the KdTrees assignment developed at Princeton University by Kevin Wayne, with boid simulation by Josh Hug.

Files to Submit

1. `ArrayST.java`
2. `Spell.java`
3. `BrutePointST.java`
4. `KdTreePointST.java`
5. `report.txt`

Before you submit your files, make sure:

- Your programs meet the style requirements by running the following command in the terminal.

  ```
  >_ ~/workspace/project5
  $ check_style src/*.java
  ```

- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You use the template file `report.txt` for your report.
- Your report meets the prescribed guidelines.

Exercises

Exercise 1. (*Graph Properties*) Consider an undirected graph G with V vertices and E edges.

- The *degree distribution* of G is a function mapping each degree value in G to the number of vertices with that value.
- The *average degree* of G is $\frac{2E}{V}$.
- The *average path length* of G is the average length of all the paths in G.
- The local clustering coefficient C_i for a vertex v_i is the number of edges that actually exist between the vertices in its neighbourhood divided by the number of edges that could possibly exist between them, which is $\frac{V(V-1)}{2}$. The *global clustering coefficient* of G is $\frac{1}{V}\sum_i^V C_i$.

Implement a data type called `GraphProperties` with the following API to compute the aforementioned graph properties:

GraphProperties

`GraphProperties(Graph G)`	computes graph properties for the undirected graph `G`
`RedBlackBinarySearchTreeST<Integer, Integer> degreeDistribution()`	returns the degree distribution of the graph
`double averageDegree()`	returns the average degree of the graph
`double averagePathLength()`	returns the average path length of the graph
`double clusteringCoefficient()`	returns the global clustering coefficient of the graph

>_ ~/workspace/project6

```
$ java GraphProperties data/tinyG.txt
Degree distribution:
  1: 3
  2: 4
  3: 5
  4: 1
Average degree         =   2.308
Average path length    =   3.090
Clustering coefficient =   0.256
```

GraphProperties.java

```
import dsa.BFSPaths;
import dsa.Graph;
import dsa.RedBlackBinarySearchTreeST;
import stdlib.In;
import stdlib.StdOut;

public class GraphProperties {
    private RedBlackBinarySearchTreeST<Integer, Integer> st; // degree -> frequency
    private double avgDegree;                                // average degree of the graph
    private double avgPathLength;                            // average path length of the graph
    private double clusteringCoefficient;                    // clustering coefficient of the graph

    // Computes graph properties for the undirected graph G.
    public GraphProperties(Graph G) {
        ...
    }

    // Returns the degree distribution of the graph (a symbol table mapping each degree value to
    // the number of vertices with that value).
    public RedBlackBinarySearchTreeST<Integer, Integer> degreeDistribution() {
        ...
    }

    // Returns the average degree of the graph.
    public double averageDegree() {
        ...
    }

    // Returns the average path length of the graph.
    public double averagePathLength() {
        ...
    }

    // Returns the global clustering coefficient of the graph.
```

```
    public double clusteringCoefficient() {
        ...
    }

    // Returns true if G has an edge between vertices v and w, and false otherwise.
    private static boolean hasEdge(Graph G, int v, int w) {
        for (int u : G.adj(v)) {
            if (u == w) {
                return true;
            }
        }
        return false;
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        In in = new In(args[0]);
        Graph G = new Graph(in);
        GraphProperties gp = new GraphProperties(G);
        RedBlackBinarySearchTreeST<Integer, Integer> st = gp.degreeDistribution();
        StdOut.println("Degree distribution:");
        for (int degree : st.keys()) {
            StdOut.println("   " + degree + ": " + st.get(degree));
        }
        StdOut.printf("Average degree          = %7.3f\n", gp.averageDegree());
        StdOut.printf("Average path length     = %7.3f\n", gp.averagePathLength());
        StdOut.printf("Clustering coefficient = %7.3f\n", gp.clusteringCoefficient());
    }
}
```

Exercise 2. (*DiGraph Properties*) Consider a digraph G with V vertices.

- G is a *directed acyclic graph (DAG)* if it does not contain any directed cycles.
- G is a *map* if every vertex has an outdegree of 1.
- A vertex v is a *source* if its indegree is 0.
- A vertex v is a *sink* if its outdegree is 0.

Implement a data type called `DiGraphProperties` with the following API to compute the aforementioned digraph properties:

DiGraphProperties

`DiGraphProperties(DiGraph G)`	computes graph properties for the digraph `G`
`boolean isDAG()`	returns `true` if the digraph is a DAG, and `false` otherwise
`boolean isMap()`	returns `true` if the digraph is a map, and `false` otherwise
`Iterable<Integer> sources()`	returns all the sources in the digraph
`Iterable<Integer> sinks()`	returns all the sinks in the digraph

~/workspace/project6

```
$ java DiGraphProperties data/tinyDG.txt
Sources: 7
Sinks: 1
Is DAG? false
Is Map? false
```

DiGraphProperties.java

```
import dsa.DiCycle;
import dsa.DiGraph;
import dsa.LinkedBag;
import stdlib.In;
import stdlib.StdOut;

public class DiGraphProperties {
    private boolean isDAG;              // is the digraph a DAG?
    private boolean isMap;              // is the digraph a map?
    private LinkedBag<Integer> sources; // the sources in the digraph
    private LinkedBag<Integer> sinks;   // the sinks in the digraph
```

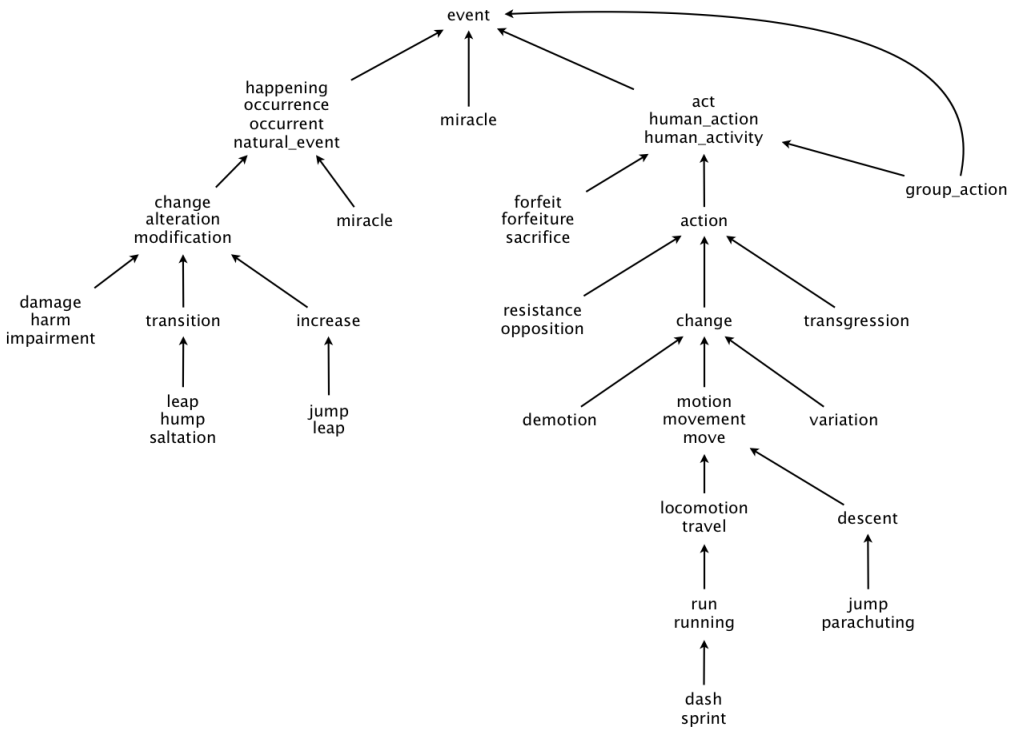
```
    // Computes graph properties for the digraph G.
    public DiGraphProperties(DiGraph G) {
        ...
    }

    // Returns true if the digraph is a directed acyclic graph (DAG), and false otherwise.
    public boolean isDAG() {
        ...
    }

    // Returns true if the digraph is a map, and false otherwise.
    public boolean isMap() {
        ...
    }

    // Returns all the sources (ie, vertices without any incoming edges) in the digraph.
    public Iterable<Integer> sources() {
        ...
    }

    // Returns all the sinks (ie, vertices without any outgoing edges) in the digraph.
    public Iterable<Integer> sinks() {
        ...
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        In in = new In(args[0]);
        DiGraph G = new DiGraph(in);
        DiGraphProperties gp = new DiGraphProperties(G);
        StdOut.print("Sources: ");
        for (int v : gp.sources()) {
            StdOut.print(v + " ");
        }
        StdOut.println();
        StdOut.print("Sinks: ");
        for (int v : gp.sinks()) {
            StdOut.print(v + " ");
        }
        StdOut.println();
        StdOut.println("Is DAG? " + gp.isDAG());
        StdOut.println("Is Map? " + gp.isMap());
    }
}
```

Problems

Goal Find the shortest common ancestor of a digraph in WordNet, a semantic lexicon for the English language that computational linguists and cognitive scientists use extensively. For example, WordNet was a key component in IBM's Jeopardy-playing Watson computer system.

WordNet groups words into sets of synonyms called synsets. For example, {*AND circuit, AND gate*} is a synset that represents a logical gate that fires only when all of its inputs fire. WordNet also describes semantic relationships between synsets. One such relationship is the *is-a* relationship, which connects a *hyponym* (more specific synset) to a *hypernym* (more general synset). For example, the synset {*gate, logic gate*} is a hypernym of {*AND circuit, AND gate*} because an AND gate is a kind of logic gate.

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 $v \to w$ denotes that w is a hypernym of v. The WordNet digraph is a *rooted DAG*: it is acyclic and has one vertex — the root — 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 shown 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 *comma-separated values* (CSV) format: each line contains a sequence of fields, separated by commas.

- *List of synsets.* The file `synsets.txt` contains all noun synsets in WordNet, one per line. Line i of the file (counting from 0) contains the information for synset i. The first field is the *synset id*, which is always the integer i; the second field is the synonym set (or synset); and the third field is its dictionary definition (or *gloss*), which is not relevant to this assignment.

```
% more synsets.txt
 ⋮
34,AIDS acquired_immune_deficiency_syndrome,a serious (often fatal) disease of the immune system
35,ALGOL,a programming language used to express computer programs as algorithms
36,AND_circuit AND_gate,a circuit in a computer that fires only when all of its inputs fire
37,APC,a drug combination found in some over-the-counter headache remedies
38,ASCII_character,any member of the standard code for representing characters by binary numbers
39,ASCII_character_set,(computer science) 128 characters that make up the ASCII coding scheme
40,ASCII_text_file,a text file that contains only ASCII characters without special formatting
41,ASL American_sign_language,the sign language used in the United States
42,AWOL,one who is away or absent without leave
 ⋮
```

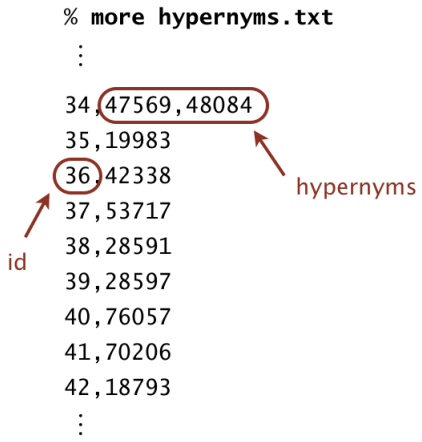
synset (points to "AIDS acquired_immune_deficiency_syndrome")

id (points to "36")

gloss (points to "one who is away or absent without leave")

For example, line 36 implies 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 constitute a synset are separated by spaces. If a noun contains more than one word, the words are connected by the underscore character.

- *List of hypernyms.* The file `hypernyms.txt` contains the hypernym relationships. Line i of the file contains the hypernyms of synset i. The first field is the synset id, which is always the integer i; subsequent fields are the id numbers of the synset's hypernyms.



For example, line 36 implies that synset 36 (`AND_circuit AND_Gate`) has 42338 (`gate logic_gate`) as it only hypernym. Line 34 implies that synset 34 (`AIDS acquired_immune_deficiency_syndrome`) has two hypernyms: 47569 (`immunodeficiency`) and 48084 (`infectious_disease`).

Problem 1. (*`WordNet` Data Type*) Implement an immutable data type called `WordNet` with the following API:

`WordNet`	
`WordNet(String synsets, String hypernyms)`	constructs a `WordNet` object given the names of the input (synset and hypernym) files
`Iterable<String> nouns()`	returns all WordNet nouns
`boolean isNoun(String word)`	returns `true` if the given word is a WordNet noun, and `false` otherwise
`String sca(String noun1, String noun2)`	returns a synset that is a shortest common ancestor of `noun1` and `noun2`
`int distance(String noun1, String noun2)`	returns the length of the shortest ancestral path between `noun1` and `noun2`

Corner Cases

- The constructor should throw a `NullPointerException()` with the message `"synsets is null"` if *synsets* is `null` and the message `"hypernyms is null"` if *hypernyms* is `null`.
- The `isNoun()` method should throw a `NullPointerException("word is null")` if *word* is `null`.
- The `sca()` and `distance()` methods should throw a `NullPointerException()` with the message `"noun1 is null"` or `"noun2 is null"` if *noun*1 or *noun*2 is `null`. The methods should throw an `IllegalArgumentException()` with the message `"noun1 is not a noun"` or `"noun2 is not a noun"` if *noun*1 or *noun*2 is not a noun.

Performance Requirements

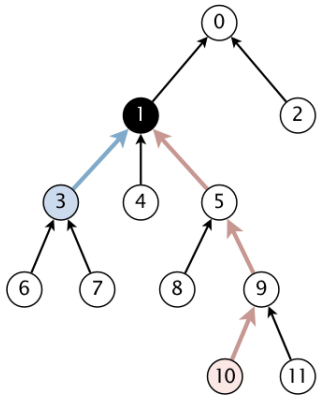
- The constructor and the `nouns()` method should run in time $T(n) \sim n$, where n is the size of the WordNet lexicon.
- The `isNoun()` method should run in time $T(n) \sim 1$.
- The `sca()` and `distance()` methods should make exactly one call to the `ancestor()` and `length()` methods in `ShortestCommonAncestor`, respectively.

```
>_ ~/workspace/project6
$ java WordNet data/synsets.txt data/hypernyms.txt worm bird
# of nouns = 119188
isNoun(worm)? true
isNoun(bird)? true
isNoun(worm bird)? false
sca(worm, bird) = animal animate_being beast brute creature fauna
distance(worm, bird) = 5
```

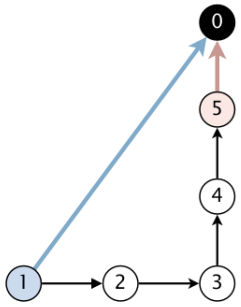
Directions:

- Instance variables:
  - A symbol table that maps a synset noun to a set of synset IDs (a synset noun can belong to multiple synsets), `RedBlackBST<String, SET<Integer>> st` .
  - A symbol table that maps a synset ID to the corresponding synset string, `RedBlackBST<Integer, String> rst`.
  - For shortest common ancestor computations, `ShortestCommonAncestor sca`.
- `WordNet(String synsets, String hypernyms)`
  - Initialize instance variables `st` and `rst` appropriately using the synset file.
  - Construct a `DiGraph` object `G` (representing a rooted DAG) with V vertices (equal to the number of entries in the synset file), and add edges to it, read in from the hypernyms file.
  - Initialize `sca` using `G`.
- `Iterable<String> nouns()`
  - Return all WordNet nouns.
- `boolean isNoun(String word)`
  - Return `true` if the given word is a synset noun, and `false` otherwise.
- `String sca(String noun1, String noun2)`
  - Use `sca` to compute and return a synset that is a shortest common ancestor of the given nouns.
- `int distance(String noun1, String noun2)`
  - Use `sca` to compute and return the length of the shortest ancestral path between the given nouns.

Shortest Common Ancestor An *ancestral path* between two vertices v and w in a rooted DAG 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. We refer to the common ancestor in a shortest ancestral path as a *shortest common ancestor*. Note that a shortest common ancestor always exists because the root is an ancestor of every vertex. Note also that an ancestral path is a path, but not a directed path.

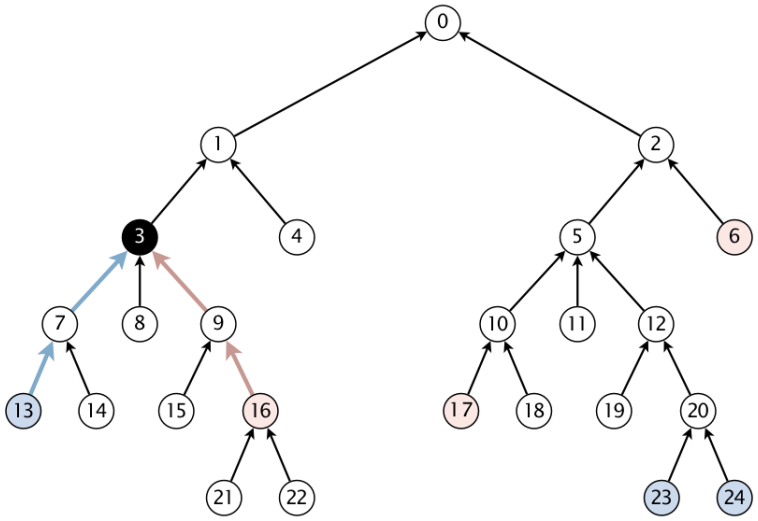


v = 3, w = 10
shortest ancestral path: 3−1−5−9−10
associated length: 4
shortest common ancestor: 1



v = 1, w = 5
ancestral path: 1−2−3−4−5
shortest ancestral path: 1−0−5
associated length: 2
shortest common ancestor: 0

We generalize the notion of shortest common ancestor to subsets of vertices. A shortest ancestral path of two subsets of vertices A and B is a shortest ancestral path over all pairs of vertices v and w, with v in A and w in B.



A = { 13, 23, 24 }, B = { 6, 16, 17 }
ancestral path: 13−7−3−1−0−2−6
ancestral path: 23−20−12−5−10−17
ancestral path: 23−20−12−5−2−6

shortest ancestral path: 13−7−3−9−16
associated length: 4
shortest common ancestor: 3

Problem 2. (*`ShortestCommonAncestor` Data Type*) Implement an immutable data type called `ShortestCommonAncestor` with the following API:

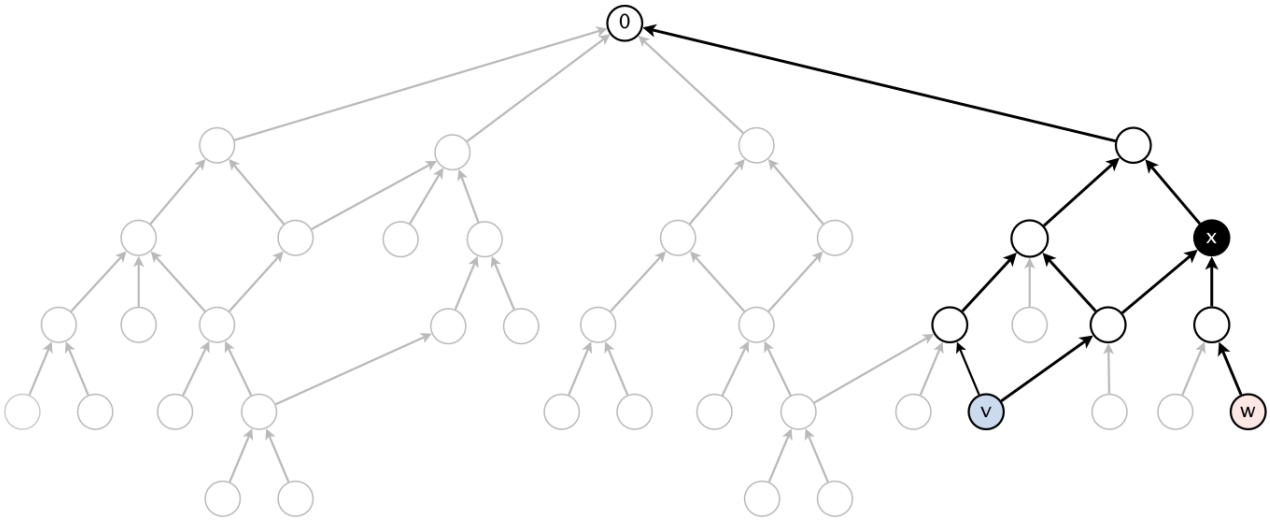
ShortestCommonAncestor	
`ShortestCommonAncestor(Digraph G)`	constructs a `ShortestCommonAncestor` object given a rooted DAG
`int length(int v, int w)`	returns length of the shortest ancestral path between vertices `v` and `w`
`int ancestor(int v, int w)`	returns a shortest common ancestor of vertices `v` and `w`
`int length(Iterable<Integer> A, Iterable<Integer> B)`	returns length of the shortest ancestral path of vertex subsets `A` and `B`
`int ancestor(Iterable<Integer> A, Iterable<Integer> B)`	returns a shortest common ancestor of vertex subsets `A` and `B`

Corner Cases

- The constructor should throw a `NullPointerException("G is null")` if G is `null`.
- The `length()` and `ancestor()` methods should throw an `IndesOutOfBoundsException()` with the message `"v is invalid"` or `"w is invalid"` if $v, w < 0$ or $v, w \geq V$, the number of vertices in G.
- The overloaded `length()` and `ancestor()` methods should throw a `NullPointerException()` with the message `"A is null"` or `"B is null"` if the vertex subset A or B is `null`. The methods should throw an `IllegalArgumentException()` with the message `"A is empty"` or `"B is empty"` if either A or B is empty.

Performance Requirements

- The constructor run in time $T(E, V) \sim 1$, where E and V are the number of edges and vertices in the digraph G, respectively.
- The methods `length()` and `ancestor()` should run in time $T(E, V) \sim E + V$. To be precise, they should run in time proportional to the number of vertices and edges reachable from the argument vertices. For example, to compute the shortest common ancestor of v and w in the digraph below, your algorithm can only examine the highlighted vertices and edges and it should not initialize any vertex-indexed arrays.




```
>_ ~/workspace/project6
$ java ShortestCommonAncestor data/digraph1.txt
3 10 8 11 6 2
<ctrl-d>
length = 4, ancestor = 1
length = 3, ancestor = 5
length = 4, ancestor = 0
```

Directions:

- Instance variable:

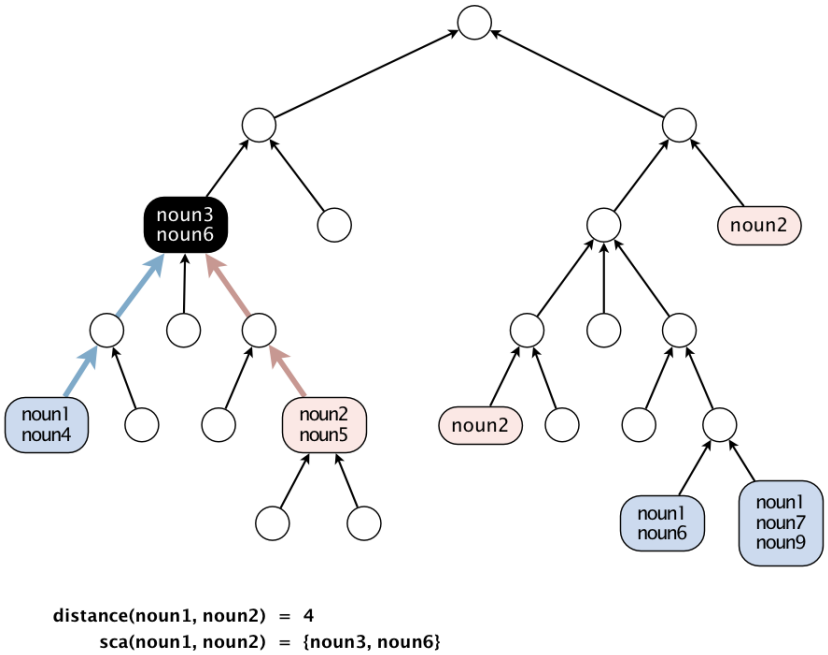
  - A rooted DAG, `DiGraph G`.

- `ShortestCommonAncestor(DiGraph G)`
  - Initialize instance variable appropriately.
- `private SeparateChainingHashST<Integer, Integer> distFrom(int v)`
  - Return a map of vertices reachable from `v` and their respective shortest distances from `v`, computed using BFS starting at `v`.
- `int length(int v, int w)`
  - Return the length of the shortest ancestral path between `v` and `w`; use `ancestor(int v, int w)` and `distFrom(int v)` methods to implement this method.
- `int ancestor(int v, int w)`
  - Return the shortest common ancestor of vertices `v` and `w`; to compute this, enumerate the vertices in `distFrom(v)` to find a vertex `x` that is also in `distFrom(w)` and has the minimum value for `distFrom(v)[x] + distFrom(w)[x]`.
- `private int[] triad(Iterable<Integer> A, Iterable<Integer> B)`
  - Return a 3-element array consisting of a shortest common ancestor `a` of vertex subsets `A` and `B`, a vertex `v` from `A`, and a vertex `w` from `B` such that the path `v-a-w` is the shortest ancestral path of `A` and `B`; use `length(int v, int w)` and `ancestor(int v, int w)` methods to implement this method.
- `int length(Iterable<Integer> A, Iterable<Integer> B)`
  - Return the length of the shortest ancestral path of vertex subsets `A` and `B`; use `triad((Iterable<Integer> A, Iterable<Integer> B)` and `distFrom(int v)` methods to implement. this method
- `int ancestor(Iterable<Integer> A, Iterable<Integer> B)`
  - Return a shortest common ancestor of vertex subsets `A` and `B`; use `triad((Iterable<Integer> A, Iterable<Integer> B)` to implement this method.

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, you consider *George W. Bush* and *John F. Kennedy* (two U.S. presidents) to be more closely related than *George W. Bush* and *chimpanzee* (two primates). It might not be clear whether *George W. Bush* and *Eric Arthur Blair* are more related than two arbitrary people. However, both *George W. Bush* and *Eric Arthur Blair* (aka George Orwell) are famous communicators and, therefore, closely related. We define the semantic relatedness of two WordNet nouns x and y as follows:

- A is set of synsets in which x appears;
- B is set of synsets in which y appears;
- $sca(x, y)$ a shortest common ancestor of A and B; and
- $distance(x, y)$ is length of shortest ancestral path of A and B.

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



Outcast Detection Given a list of WordNet nouns $x_1, x_2, \ldots, x_n$, 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:

$$d_i = distance(x_i, x_1) + distance(x_i, x_2) + \cdots + distance(x_i, x_n)$$

and return a noun x_i for which d_i is maximum. Note that because $distance(x_i, x_i) = 0$, it will not contribute to the sum.

Problem 3. (*`Outcast` Data Type*) Implement an immutable data type called `Outcast` with the following API:

`Outcast`	
`Outcast(WordNet wordnet)`	constructs an `Outcast` object given the WordNet semantic lexicon
`String outcast(String[] nouns)`	returns the outcast noun from `nouns`

You may assume that argument to `outcast()` contains only valid WordNet nouns (and that it contains at least two such nouns).

```
>_ ~/workspace/project6
$ java Outcast data/synsets.txt data/hypernyms.txt < data/outcast10.txt
cat cheetah dog wolf *albatross* horse zebra lemur orangutan chimpanzee
$ java Outcast data/synsets.txt data/hypernyms.txt < data/outcast11.txt
apple pear peach banana lime lemon blueberry strawberry mango watermelon *potato*
$ java Outcast data/synsets.txt data/hypernyms.txt < data/outcast12.txt
competition cup event fielding football level practice prestige team tournament world *mongoose*
```

Directions:

- Instance variable:
  - The WordNet semantic lexicon, `WordNet wordnet`.
- `Outcast(WordNet wordnet)`
  - Initialize instance variable appropriately.
- `String outcast(String[] nouns)`

– Compute the sum of the distances (using `wordnet`) between each noun in `nouns` and every other, and return the noun with the largest distance.

Data The `data` directory has a number of sample input files for testing. See project writeup for the format of the synset (`synset*.txt`) and hypernym (`hypernym*.txt`) files. The `digraph*.txt` files representing digraphs can be used as inputs for `ShortestCommonAncestor`.

```
>_ ~/workspace/project6
$ cat data/digraph1.txt
12
11
 6  3
 7  3
 3  1
 4  1
 5  1
 8  5
 9  5
10  9
11  9
 1  0
 2  0
```

The `outcast*.txt` files, each containing a list of nouns, can be used as inputs for `Outcast`

```
>_ ~/workspace/project6
$ cat data/outcast5a.txt
horse
zebra
cat
bear
table
```

Acknowledgements This project is an adaptation of the WordNet assignment developed at Princeton University by Alina Ene and Kevin Wayne.

Files to Submit

1. `GraphProperties.java`
2. `DiGraphProperties.java`
3. `WordNet.java`
4. `ShortestCommonAncestor.java`
5. `Outcast.java`
6. `report.txt`

Before you submit your files, make sure:

- Your programs meet the style requirements by running the following command in the terminal.

  ```
  >_ ~/workspace/project6
  $ check_style src/*.java
  ```

- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You use the template file `report.txt` for your report.
- Your report meets the prescribed guidelines.