Computer Science E214

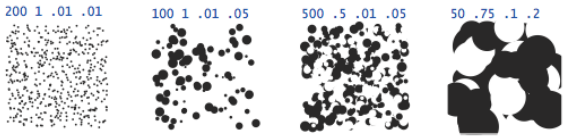
Tutorial 2

2015

1 Book 1.5.26

Write a program `Circles` that draws filled black and white circles of random size at random positions in the unit square, producing images like those below. Your program should take four command-line arguments: the number of circles, the probability that each circle is black, the minimum radius, and the maximum radius, so that the left-most image might be obtained by running:

```
$ java Circles 200 1 .01 .01
```



2 Not The Textbook's Peaks (Book 1.5.29)

Suppose that a terrain is represented by a two-dimensional grid of elevation values (in meters). A *peak* is a grid point whose four neighbouring cells are strictly lower. Grid points with fewer than four neighbours are on the edge of the grid and these points are not peaks. Write a program `Peaks` that reads a terrain from standard input (or an input file using redirection) and then finds and prints the row and column coordinate of each peak, as well as how high that peak is.

Input and output format: The first line of the input specifies the dimensions of a $N \times M$ matrix of double values, separated by whitespace. Each successive line of input is one row of the matrix. Thus the input format looks like:

```
N M
row1col1 row1col2 ... row1colM
row2col1 row2col2 ... row2colM
...
rowNcol1 rowNcol2 ... rowNcolM
```

For example, opening the input file called `easytwopeaks.txt` we see the following:

```
4 7
0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.3 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.4 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

In this example there are two peaks. The first is in row 2, column 2, and has a height of 0.3, while the second peak is at row 3, column 5, with a height of 0.4. For this input, your program should print:

```
2,2->0.3
3,5->0.4
```

Note that the peaks should be printed in the order in which the corresponding heights were inputted into your program.

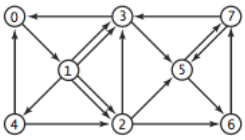
Other testing files have been included: `onepeak.txt`, `twopeak.txt`, and `edgecheck.txt`[1]. The corresponding output for each has been placed in other text files with names ending in `_result.txt`, allowing you to check your program.

Additional challenge: Modify your program to optionally take the maximum dimension N of a square terrain and a number of simulations T for each size as command-line arguments. If these are provided, no further input is provided: instead, your program should print a table with estimates of (i) the average number of peaks and (ii) the average maximum peak height in a random $i \times i$ terrain (obtained using simulation) for each i between 1 and N. (Assume height values are uniform random values between 0 and 1.) An example of using this program:

```
$  java Peaks 10 100
  1    0.0000 0.000000
  2    0.0000 0.000000
  3    0.1800 0.147909
  4    0.7400 0.552123
  5    1.8600 0.814055
  6    3.1400 0.936190
  7    5.0400 0.965826
  8    6.9500 0.970957
  9    9.8300 0.980337
 10   12.6500 0.982754
```

3 Discerning Surfing (Book 1.6.2, 1.6.11, and 1.6.12)

Encode the eight-page example shown below for use as input to `Transition.java` in a file called `eightpage.txt`. Then implement and use `Transition.java` to obtain a transition matrix for the example. Use this matrix to calculate the page ranks by implementing and using both `RandomSurfer.java` and `Markov.java`.



Eight-page example

Now modify `Transition` to ignore the effect of multiple links. That is, if there are multiple links from one page to another, count them as one link. Call your modified class `SingleTransition`. Repeat the analysis above on the example, but using the modified transition matrix obtained with `SingleTransition`.

Repeat the analysis described above on the three-page example provided in `threepage.txt` in the tutorial resources.

Compare your results in each case — which do you think is a better approach to ranking web pages?

For this problem, submit your program `SingleTransition.java`, but include the following in comments at the bottom of the file: (i) the contents of your file `eightpage.txt`, (ii) the page ranks you obtained for each method on each data set, and (iii) your thoughts on any differences observed between the approaches and their results on the different examples..

[1] `edgecheck.txt` tests if your program handles edges correctly.

4 The binomial distribution (Book 2.1.34)

Write a function

```
public static double binomial(int N, int k, double p)
```

in a class `Binomial` to compute the probability of obtaining exactly k heads in N flips of a biased coin (heads with probability p) using the formula

$$f(N, k, p) = p^k (1-p)^{N-k} N!/(k!(N-k)!)$$

Hint: To stave off overflow, compute $x = \ln f(N, k, p)$ and then return e^x.

Write a `main` method for the class that obtains N and p from the command line and prints out

$$\sum_{k=0}^{N} f(N, k, p) \ ,$$

which should be equal to one (up to numerical accuracy).

Also use the class `BinomialTest` provided in the tutorial resources to unit test your `binomial` method. To use this class to test in DrJava, open `Binomial.java` and `BinomialTest.java`, compile them both, and then click on test while you have either of them open on the editing page. For the command prompt, you should be able run `javat.bat BinomialTest`. If your tests all pass, temporarily "break" your code so that you can see how failed unit tests work.

Take a look at the contents of `BinomialTest.java` and ensure you understand how the tests are structured. Try to add your own method — see http://www.drjava.org/docs/user/ch07.html for instructions on creating tests.

5 Histograms (Book 2.1.19)

Write a method `histogram` in a library called `Histograms` that takes an integer M and an array `a[]` of `int` values as parameters and returns an array of length M whose ith entry ($0 \leq i < M$) is the number of times the integer i appeared in the argument array. Any input values which are negative, or are M or larger, should simply be ignored.

As one example, in the DrJava interactions pane, you should get the following behaviour:

```
> Histograms.histogram(5, new int[]{0, 0, 0, 4, 4, 3})
{ 3, 0, 0, 1, 2 }
```

There are 5 elements in the final results, because of the 5 passed as the first parameter to `Histograms.histogram`. The numbers in the result array are as they are because 0, 1, 2, 3 and 4 appear 3, 0, 0, 1 and 2 times respectively in the original parameter array {`0, 0, 0, 4, 4, 3`}. Some more examples follow:

```
> Histograms.histogram(3, new int[]{})
{ 0, 0, 0 }
> Histograms.histogram(4, new int[]{1,2,3,1,2,3,1,1,1,2})
{ 0, 5, 3, 2 }
> Histograms.histogram(4, new int[]{1,2,-3,3,1,2,3,7,1,1,4,1,2,0})
{ 1, 5, 3, 2 }
```

Write a test client `TestHistogram` for your method which allows the user to specify M on standard input, followed by N, the number of values used to construct the histogram. The N values that should populate `a[]` should then follow. See some example files with the correct outputs in the resources file for this tut. Your program should be able to process any file in the given format. For the first input example your command and output should look like this:

```
$java TestHistogram < histogram_example1.txt
2 3 1 4
```

6 The Erlang loss formula

Consider a model of a telephone network switch where λ is the arrival rate of telephone calls (measured in calls per second) to a switch that can carry maximally N calls, and $1/\mu$ is the average duration (measured in seconds) of a call. In telephony jargon, λ is called the *offered traffic* per unit time and $\rho = \lambda/\mu$ the *offered load*. The load ρ is measured in units of "Erlangs".

It is known that (under some reasonable assumptions we don't go into here) the probability that an incoming call is lost because all N circuits are busy is a function of the load, ρ, and N. This *blocking probability* is given by the *Erlang loss formula* (also known as the *Erlang B-formula*):

$$B(N, \rho) = \frac{\rho^N / N!}{\sum_{n=0}^{N} \rho^n / n!} \tag{1}$$

Eqn. (1) looks easy to evaluate, but think of the situation where $\rho = 100$ and $N = 200$. Then there is a term of the form $(100)^{200}/200!$ that needs to be evaluated, and this is difficult to do exactly, even for a computer. To get around this we use a recursive formulation for $B(N, \rho)$ which calculates $B(N, \rho)$ in terms of $B(N-1, \rho)$

$$B(N, \rho) = \frac{\rho\, B(N-1, \rho)}{N + \rho B(N-1, \rho)} \tag{2}$$

where $B(0, \rho) = 1$.

Create a class `Erlang`, and use it to perform the following:

1. Write a recursive method `recErlang` which takes two parameters N and ρ and computes and returns the blocking probability $B(N, \rho)$ using Eqn. (2).
2. Determine the number of recursive calls that are used in total by this method to calculate the blocking probability — put your answer in the documentation for the method. (Note: If you are making more than one recursive call in your function, you are being inefficient, and you will run into trouble with large values of N — find a way to avoid it!)
3. Develop a non-recursive method `nonrecErlang` with the same parameters which also computes $B(N, \rho)$.
4. Implement a `main` method for testing that accepts N and ρ as command-line arguments, and then prints the results of the recursive and the non-recursive methods, as below:

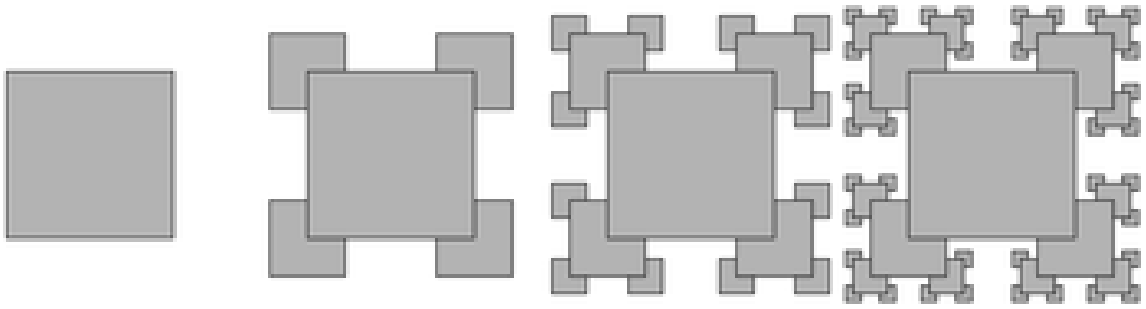
```
$java Erlang 10 20
Recursive: B(10, 20.000000) = 0.537963
Non-recursive: B(10, 20.000000) = 0.537963
```

7 Recursive Squares (Book 2.3.22)

Write a program `RecursiveSquares` to produce each of the following recursive patterns shown in Figures 1 - 4, using `StdDraw` to draw the squares. The ratio of the sizes of the squares is 2.2 : 1. To draw a shaded square, draw a filled gray square, then an unfilled black square. Give the program two command-line parameters: the first should choose between the different recursive patterns: 1 for *Front*, 2 for *Order*, 3 for *Right*, or 4 for *Shaded*; the second value N should specify the depth of the recursion. Figures 1 - 4 show the results for recursion depths of *one* through *four* (`(a)` - `(d)`). An example for running the program:

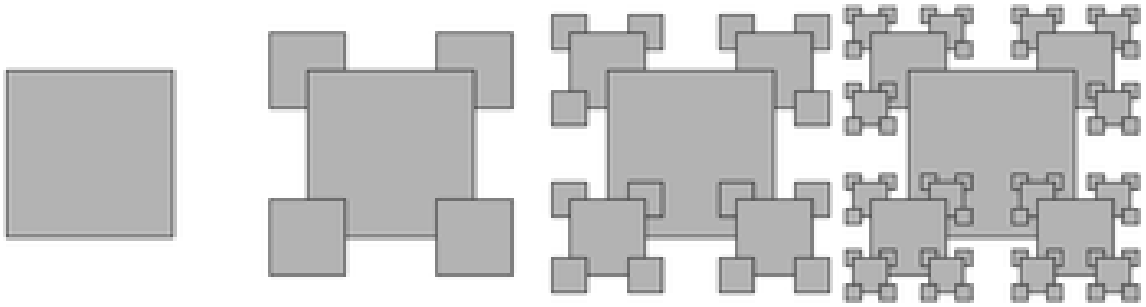
```
$ java RecursiveSquares 1 4
```

should render Figure 1d.



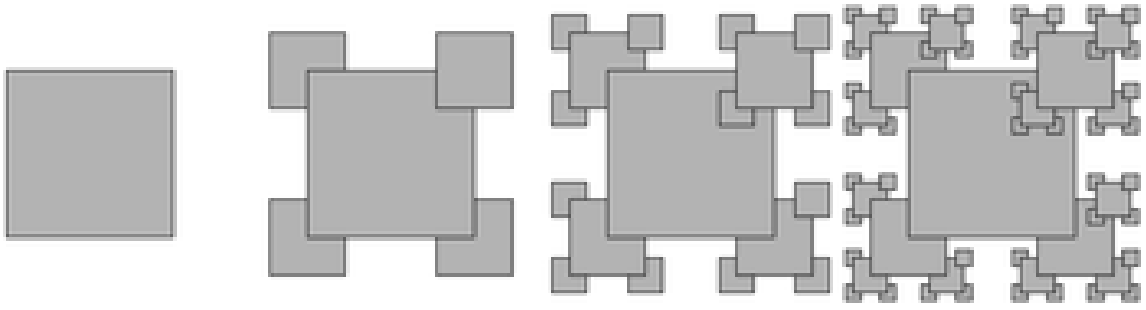
(a) (b) (c) (d)

Figure 1: Front



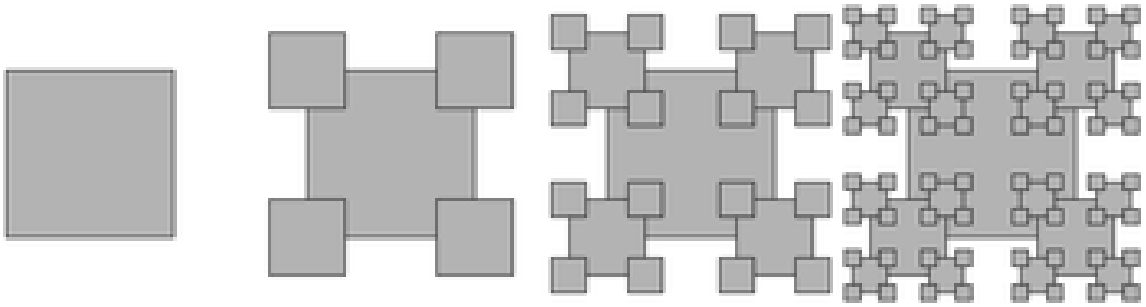
(a) (b) (c) (d)

Figure 2: Order



(a) (b) (c) (d)

Figure 3: Right



(a) (b) (c) (d)

Figure 4: Shaded