Comp 200 Fall 2017 Project 1
Due date: 16th October,2017

- **Warnings**
  - Please read the "Project 1" handout carefully
  - Download your template project1.scm from DriveF@VOL/COURSES/UGRADS/COMP200/SHARE/[username]/project1/.
  - Please do the necessary editing on your project1.scm and make sure your file loads without errors by using RUN (DrRacket) command. **Only the files that are error-free will be graded.**
  - Name of the submission file have to be **project1.scm**. If you need to make re-submission the format should be the following: **project1_X.scm** where X is the submission number. For example, if you want to re-submit your homework for a **second time**, the name of file should be **project1_2.scm** . **Files that are named other than that will not be graded.**
  - Please upload your homework to the folder DriveF@VOL/COURSES/UGRADS/COMP200/HOMEWORK/[username]/project1/.
  - In *DrRacket* please select the **R5RS** option from the language toolbar.
- **Reading** Chapter 1 and Section 2.1 in Structure and Interpretation of Computer Programs
- **Help** Please email comp200@ku.edu.tr if you have any questions.

Purpose

The purpose of this project is for you to gain experience with writing and testing relatively simple procedures. You should read the project submission instructions in the Assignments section of the course website to find out how to retrieve the project file and submit your solutions. For each problem below, include your code (with identification of the problem number being solved), as well as comments and explanations of your code, and demonstrate your code's functionality against a set of test cases. On occasion, we may provide some example test cases, but you should **always** create and include your own additional, meaningful test cases to ensure that your code works not only on typical inputs, but also on more difficult cases. Get in the habit of writing and running these test cases after **every** procedure you write — no matter how trivial the procedure may seem to you.

Scenario

Alyssa I. Hackerkesen has just arrived in Cambridge, England, about to embark on a semester in the Cambridge exchange program. Suffering from severe culture shock, she is delighted to find that her favorite

candy, *M&M's*, comes in a British variant, *Smarties*. Even better, she finds that *Smarties* come in a variety of colors (colours?), and that her favorite color, **orange**, is well represented. Thrilled by this unexpected turn of events, she decides that she needs to develop software in an effort to optimize the number of orange smarties she can consume over the next three months.

The key to this project will be to model the way that packets of Smarties are produced. Throughout the project, we'll assume that each Smarties packet is produced by the manufacturer, in the following random process. Each Smarties packet will contain some number, n, of smarties (e.g., $n = 100$). The value for n will be chosen by Alyssa or the manufacturer. The colors of these n Smarties are then chosen at random. More specifically, the probability that a randomly chosen Smartie is orange is p, where p is some value between 0 and 1. If p is high, there's a good chance that there will be a large number of orange Smarties in the packet. If p is low, there will a good chance that there will be very few Smarties that are orange, and Alyssa will be disappointed.

Problem 1: Some Simple Probability Theory

The first thing that Alyssa would like to do is model the probability of seeing exactly b orange Smarties in a Smarties packet. This probability will vary depending on the number of smarties in a packet, n, and the probability of any one Smartie being orange, p. In general, the probability of seeing exactly b smarties is given by the following formula:

$$bin(n, b, p) = \binom{n}{b} p^b (1-p)^{n-b}$$

where $\binom{n}{b}$ is defined as the binomial coefficient

$$\binom{n}{b} = \frac{n!}{(n-b)!b!}$$

(This result comes from the *binomial distribution*, if you're interested in reading more about this you can see `http://mathworld.wolfram.com/BinomialDistribution.html`.)

We want you to write a procedure that takes parameters n and b as input, and returns the binomial coefficient $\binom{n}{b}$ in **two different ways**.

For the first method, you should be able to directly encode the formula above (i.e., write a procedure to compute `factorial`, and then use it to implement the formula):

```
(define factorial
  (lambda (n)
    YOUR-CODE-HERE))

(define binomial
  (lambda (n b)
    YOUR-CODE-HERE))
```

Test your code for at least the following cases:

```
(binomial 5 1)  ; -> 5
(binomial 5 2)  ; -> 10
(binomial 10 5)  ; -> 252
```

As a second method, you can take advantage of properties of factorial. In particular,

$$\binom{n}{b} = \frac{n!}{(n-b)!b!} = \frac{n}{b}\frac{(n-1)}{(b-1)}\cdots\frac{(n-b+1)}{1}$$

Write a second version of `binomial` that uses this idea as its basis and does not use factorial:

```
(define binomial-2
  (lambda (n b)
    YOUR-CODE-HERE))
```

Test your code to show that binomial-2 in fact gives the same answers as binomial.

Building on this code, write a procedure that takes parameters n, b, and p as input, and returns the probability that a packet of n smarties has exactly b orange smarties, given that p is the probability of any single Smartie being orange:

```
(define exactly-b-smarties
  (lambda (n b p)
    YOUR-CODE-HERE))
```

Test your code for at least the following cases (you should be able to calculate the later cases directly using a calculator, to verify that your code is producing the correct answer):

```
(exactly-b-smarties 1 1 0.5)   ; -> 0.5
(exactly-b-smarties 2 1 0.5)   ; -> 0.5
(exactly-b-smarties 2 2 0.5)   ; -> 0.25
(exactly-b-smarties 2 1 0.3)   ; ->
(exactly-b-smarties 10 2 0.3)  ; ->
```

Note that you might think about what checks you want to put into your code to ensure that it handles the right set of parameters. For example, what if `b` is greater than `n`?

Problem 2: More Probability Theory

Alyssa would now like to calculate the probability that there will be *at least* b orange smarties in a bag, given that it is generated with underlying parameters n and p. Note that the probability of seeing *at least* b orange smarties in a bag can be calculated as the probability of seeing exactly b orange smarties, plus the

probability of seeing exactly $b+1$ orange smarties, and so on, up to seeing exactly n smarties. This can be captured mathematically as:

$$\sum_{r=b}^{n} \binom{n}{r} p^r (1-p)^{n-r}$$

which uses the summation notation

$$\sum_{r=b}^{n} f(r) = f(b) + f(b+1) + f(b+2) + \ldots + f(n)$$

which applies to any function $f(r)$.

In other words, we've made use of the identity

$$\begin{aligned} Probability(\text{at least } b \text{ smarties}) \quad = \quad & Probability(\text{exactly } b \text{ smarties}) + \\ & Probability(\text{exactly } b+1 \text{ smarties}) + \\ & Probability(\text{exactly } b+2 \text{ smarties}) + \\ & \ldots \\ & Probability(\text{exactly } n \text{ smarties}) \end{aligned}$$

Making use of your code for `exactly-b-smarties`, write code which takes as input parameters n, b, and p, and returns the probability that at least b orange Smarties are found in a bag:

```
(define atleast-b-smarties
  (lambda (n b p)
    YOUR-CODE-HERE))
```

Write your code in two different ways. In the first method, use a recursive process (i.e., rely on the fact that getting at least `b` smarties is the same as either getting exactly `b` smarties or getting at least `b+1` smarties). In the second method, use an iterative process, while relying on your code to get exactly `b` smarties.

Test your code for **at least** the following cases:

```
(atleast-b-smarties 9 5 0.5)        ; -> 0.5
(atleast-b-smarties 19 10 0.5)      ; -> 0.5
(atleast-b-smarties 10 5 0.6)       ; ->
(atleast-b-smarties 15 5 0.3)       ; ->
```

Problem 3: Choosing a Bag

Alyssa is now almost ready to go shopping for Smarties. To her delight, she finds that each Smarties bag has the parameters n and p printed on it; the values for n and p may vary bag by bag. A bag costs one pound (about 2 dollars), and she decides that a bag will be worth buying as long as there's at least a 50% chance that it will have 8 or more orange Smarties.

Write a function `good-bag` which takes parameters n and p as input, and returns `#t` if the bag is worth buying, `#f` if it is not worth buying. Be careful of a special case, that if $n < 8$ there is no chance that the bag will have 8 or more Smarties.

```
(define good-bag
  (lambda (n p)
    YOUR-CODE-HERE))
```

Test your code for at least the following cases:

```
(good-bag 7 1)        ; -> #f
(good-bag 8 1)       ; -> #t
(good-bag 8 0.5)     ; -> #f
(good-bag 8 0.99)    ; -> #t
(good-bag 16 0.5)     ; ->
(good-bag 16 0.7)     ; ->
(good-bag 16 0.4)     ; ->
```

Problem 4: Choosing a Value for p

After a couple of months in England, Alyssa has purchased so many bags of Smarties that she has the power to directly negotiate with the manufacturer. Whenever she wants a bag of Smarties, she can call up the company, which will produce a customized bag for her. They will tell her the number of Smarties, n, that will appear in the bag. You can assume that n will be at least 8. Alyssa then gets to choose a minimum value for the probability p. In particular, she wants to choose the minimum value of p such that the bag is a "good bag", as defined in the previous problem.

Write a function `minimum-p` which takes a parameter n as input, and returns p, the minimum probability under which the bag will be acceptable to Alyssa. `minimum-p` should be a recursive procedure that tries different values for p ranging from 0 to 1, sampled at some specific increment size `inc` (i.e., tries $p = 0$, then $p =$inc, then $p = 2$ inc, and so on), and which makes use of `good-bag`. The value for the increment should be an additional parameter of the function, called `inc`:

```
(define minimum-p
  (lambda (n inc)
    YOUR-CODE-HERE))
```

Test your code for the following case:

```
(minimum-p 12 0.01)
```

You should get a value of 0.63. To check that this is correct, what do you get for the following applications of `good-bag`?:

```
(good-bag 12 0.63)              ; ->
(good-bag 12 (- 0.63 0.01))     ; ->
```

Now try

```
(minimum-p 12 0.1)
(minimum-p 12 0.01)
(minimum-p 12 0.001)
(minimum-p 12 0.0001)
(minimum-p 12 0.00001)
```

Create 3 other test cases that are similar to the case above.

Problem 5: Choosing p More Efficiently

Alyssa discovers that her code for calculating `minimum-p` (in Problem 4) is too slow; her telephone bill for calls to the company is becoming so big that she's running out of money for Smarties. She decides to try to make a more efficient version of the function.

First, though, you should implement a new version of `minimum-p` which displays (or prints out) the number of times that `good-bag` is called by the recursive search procedure. Call this procedure `minimum-p-new`. For example, it should show the following behavior:

```
(minimum-p-new 12 0.01)
Number of calls to good-bag: 63
0.63000000000003
```

(The procedure returns the value 0.63 (roughly), and also displays or prints out 63, i.e., the number of times good-bag was called, which is ironically very similar in this case.) You may find the procedures `newline` (with no arguments), and `display` (with one argument) convenient – the former outputs a blank line, and the latter will print out the value of its argument on the screen.

Try the following test cases:

```
(minimum-p-new 15 0.1)
(minimum-p-new 15 0.01)
(minimum-p-new 15 0.001)
(minimum-p-new 15 0.0001)
(minimum-p-new 15 0.00001)
```

Alyssa realises that there's a more efficient way of calculating `minimum-p`. It is based on an idea called *binary search*. First, note that we know that the value for `minimum-p` is between 0 and 1. Suppose we try a value of 0.5 (halfway between). There are then two possible outcomes:

- If a value of $p = 0.5$ gives a good bag, we know that the minimum value for p is between 0 and 0.5. We can then search this range of values, by recursing, and testing a value of $p = 0.25$.
- If $p = 0.5$ does not give a good bag, we know the minimum value for p is between 0.5 and 1. We can then search this range of values, by recursing, and testing a value of $p = 0.75$.

To implement this procedure, fill in the following code:

```
(define minimum-p-binary
  (lambda (n inc)
     (minimum-p-binary-helper n inc 0 1 0)))

(define minimum-p-binary-helper
  (lambda (n inc a b count)
     YOUR-CODE-HERE))
```

The procedure `(minimum-p-binary-helper n inc a b count)` should implement binary search for the best value of p between a and b, up to a level of precision defined by `inc`. If $(b - a) < inc$, then it should just return b as its final answer. Otherwise, it should test whether a value for $p = (a + b)/2$ leads to a good or bad bag, and make a recursive call to `minimum-p-binary-helper` with new values of a and/or b which depend on this answer. It should also keep track (using `count`) of how many times `good-bag` is called.

Test your code on the following cases:

```
(minimum-p-binary 12 0.1)
(minimum-p-binary 12 0.01)
(minimum-p-binary 12 0.001)
(minimum-p-binary 12 0.0001)
(minimum-p-binary 12 0.00001)
```

Your code should again print the number of times that `good-bag` is called when `minimum-p-binary` is used. You should find that it calls `good-bag` far fewer times than the number of times used by `minimum-p-new`, particularly when `inc` becomes small.

Problem 6: Monte-Carlo Simulations

Unfortunately, the company has a change in management, and while still willing to customize bags for Alyssa, they are slightly less flexible. The company will only produce bags in batches of some size m (e.g., $m = 8$) that they choose. Furthermore, they will specify the parameters n and p which are used to generate all of the bags in a given batch. Alyssa now wants to build a function `(estimate-good-probability m n p)` which returns the probability that at least one bag out of the m bags she receives will be a "good bag", where a "good bag" is as defined before.

This is a more difficult problem, and Alyssa decides to abandon her existing code (you should too: don't use the code developed in the previous problems!). Instead, she decides to develop an approach based

on *Monte-Carlo* methods. (See `http://en.wikipedia.org/wiki/Monte_Carlo_method` for an overview of the history of Monte-Carlo methods. Monte-Carlo methods have had widespread use since their invention, one notable case being in the development of the hydrogen bomb. But Alyssa is delighted that they finally have a serious application, i.e., in orange-smartie-consumption-optimization.)

In the Monte-Carlo method we develop, the central idea will be to build a *simulator* that randomly generates batches of bags of Smarties. We can then see how often a randomly generated batch contains at least one good bag; we can use this to estimate the probability of there being at least one good bag under parameters m, n, b.

At the center of our approach is a function, `random`. Each time a call to `(random)` is made, it returns a value chosen *uniformly at random* from the interval between 0 and 1. Roughly speaking, this means that any real number between 0 and 1 will be returned with the same, equal probability.

Try calling `(random)` a few times in the read-eval-print loop: you should find a random value between 0 and 1 returned each time.

The first function you should write is `(coin-toss p)`. This takes a parameter p which is between 0 and 1. With probability p it returns #t, with probability $(1-p)$ it returns #f:

```
(define coin-toss
  (lambda (p)
    YOUR-CODE-HERE))
```

Thus this is the simplest form of simulator: `(coin-toss p)` simulates a coin being tossed that has probability p of turning up heads, and probability $(1-p)$ of turning up tails. It returns #t or #f for heads/tails respectively. (Hint: use `(random)` to generate a number between 0 and 1, and then test whether this number is $\geq p$.)

Next, write a function `random-bag` which takes parameters n and p as input. The function should generate a bag of Smarties of size n where each Smartie has probability p of being orange. It returns a single value, which is the number of Smarties in the bag which were orange.

```
(define random-bag
  (lambda (n p)
    YOUR-CODE-HERE))
```

Your code should make use of n calls to `coin-toss`.

Next, write a function `get-m-bags` which takes parameters m, n and p as input. It should generate m bags at random using the parameters n and p, and return #t if at least one of these bags is good, i.e. has 8 or more orange smarties.

```
(define get-m-bags
  (lambda (m n p)
    YOUR-CODE-HERE))
```

Finally, write code `estimate-good-probability` which takes parameters m, n, p and t as input. As before, m is the number of bags in a batch; n is the number of Smarties in a bag; and p is the probability of

any individual Smartie being orange. The function should make t calls to `get-m-bags`, and return $\frac{g}{t}$ as its estimate, where g is the number of times `get-m-bags` returns true.

```
(define estimate-good-probability
  (lambda (m n p t)
    YOUR-CODE-HERE))
```

Give values that your code returns for the following cases:

```
(estimate-good-probability 24 12 0.5 1000)
(estimate-good-probability 24 16 0.5 1000)
(estimate-good-probability 24 16 0.3 1000)
(estimate-good-probability 24 16 0.2 1000)
```

Note: you should report numbers for **3 runs** of each case. Note that because the procedure makes use of random sampling, you'll almost certainly get slightly different answers in different runs with the same parameter values!

Problem 7: Monte-Carlo Again

Now write a new Monte-Carlo simulator for the following scenario. The company will again produce bags in batches of some size m that they choose; each bag will again have n Smarties. With probability p, each individual Smartie will be orange; with probability q, it will be blue; with probability $1 - p - q$ it will be some other color. Alyssa dislikes blue Smarties with a vengeance, and now defines a good bag to be any bag which has *more than half of the smarties of orange color and less than one-fifth of the smarties of blue color*. Alyssa now wants to build a function `(estimate-good-probability-2 m n p q t)` which returns the probability that at least one bag out of the m bags she receives will be a "good bag", where a "good bag" now has the new definition. As before, t is a parameter specifying the number of batches which are generated at random during the Monte-Carlo simulation.

Hint: use the same logic to build up your solution as you did in problem 6. Note however that your solution may require new functions, or may require substantially new versions of the functions in problem 6.