Please answer the following questions in complete sentences in submit the solution on Blackboard September 16, 2016.

Update 1, Thursday, September 15 Corrected comment in the `sample_vertex_small` and `sample_vertex_large` commands. Also fixed a typo with the name of the Julia package.

Homework 2

Problem 1: Warm up problems (15 points)

Please complete the following warm up problems.

1. G&C Chapter 3 Problem 1
2. G&C Chapter 5 Problem 1
3. G&C Chapter 5 Problem 4

Problem 2: Floating point mathematics (20 points)

1. G&C Chapter 5 Problem 9c (10 points)
2. G&C Chapter 5 Problem 12 (10 points)

Problem 3: Fibonacci, Floating point, and the Quadratic equation (15 points)

We'll study a common floating point problem, finding the roots of a quadratic equation. Recall that if we wish to find x such that:

$$ax^2 + bx + c = 0$$

then, almost every high-school student knows:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Suppose that we wish to solve this equation using coefficients from the Fibonacci series:

$$F_n x^2 - 2F_{n-1}x + F_{n-2} = 0,$$

where we use the Fibonacci series:

$$F_1 = 1, F_2 = 1, F_{n+1} = F_n + F_{n-1}.$$

1. (5 points) Show that $b^2 - 4ac = (-1)^n 4$. (Hint, first work out the case for $n = 3$ and $n = 4$, and then tackle the general problem.)

2. (5 points) The first step is important because once it's done we can write a nice formula for the roots:

$$x_1 = \frac{F_{n-1} + \sqrt{(-1)^n}}{F_n}, x_2 = \frac{F_{n-1} - \sqrt{(-1)^n}}{F_n}.$$

Write a Julia code to evaluate these roots using this expression when n is even (Hint: this is really easy!) What's the largest n such that we have distinct roots, i.e. such that $x_1 = x_2$ in floating point. Here's a section of my code to solve this problem. You don't have to do it this way if you'd like to write your own, but getting the Matlab loop write isn't the point of this question so I thought having some code would make it easier.

```
# COMPLETE and play around with N
N =
fib = zeros(N)
roots1 = zeros(Complex{Float64}, N)
roots2 = zeros(Complex{Float64}, N)
fib[1] = 1; roots1[1] = 1 # avoid reporting these equal
fib[2] = 1; roots1[2] = 1 # because we don't compute them
for i=3:N
    fib[i] = fib[i-2] + fib[i-1]
    # COMPLETE this section to fill in
end
bign = findfirst(roots1.==roots2) # finds the first zero
```

The idea is that once n gets sufficiently large, we lose the precision to represent the difference in the roots when computed exactly and so although, mathematically, there are distinct roots, numerically, there are not.

3. (5 points) Now, let's see how well our high-school formula does! Write a function `myroots(c)` with the following template:

```
"""
`myroots`
=========

Solve a quadratic equation given as a vector of its coefficients

Functions
---------
* `r = myroots(c)` returns the values of x that solves
   the quadratic equation \$c[1] x^2 + c[2] x + c[3] = 0\$
   based on formula
    \$r[1] =(-b + \sqrt{b^2 - 4ac})/2a\$
    \$r[2] =(-b - \sqrt{b^2 - 4ac})/2a\$

Example
-------
~~~~
@show myroots([1. 0. 1.])
@show myroots([-1. 0. 1.])
~~~~
"""
function myroots(c)
end
```

Now find the smallest value of N such that these numerically computed roots are the same. (Again, you don't have to use the following code, but it's a quick way to check!)

```
# COMPLETE this line
N =
fib = zeros(N)
roots1 = zeros(Complex{Float64}, N)
roots2 = zeros(Complex{Float64}, N)
fib[1] = 1; roots1[1] = 1 # avoid reporting these equal
fib[2] = 1; roots1[2] = 1 # because we don't compute them
for i=3:N
    fib[i] = fib[i-2] + fib[i-1]
    r = myroots([fib[i]; -2*fib[i-1]; fib[i-2]])
    roots1[i] = r[1]
    roots2[i] = r[2]
end
bign = findfirst(roots1.==roots2) # finds the first zero
```

Your number should be different than your other one.

4. The problem with this computation is that evaluating $b^2 - 4 * a * c$ results in losing all of the precision of the coefficients far earlier than necessary. Correctly evaluating the value $b^2 - 4 * a * c$ is possible, but much more complicated than I'd like you to look at. (Please see Kahan's On the cost of floating point computation without extra-precise arithmetic for more detail on the nearly 200 line Matlab code to correctly evaluate this function.) Even Julia itself gets this problem wrong in their `roots` function to determine the roots of a polynomial. (Hey, you are done! There isn't actually a question here!)

Here's a fun personal story. My own discovery of the importance of this problem was when I was working as an intern at Microsoft on an image deforming task. When I evaluated the solution of the quadratic in double precision, the image looked correct; but when I evaluated the solution of the quadratic in single-precision (which was faster!) the image "looked" wrong. I tracked the issue down to a cancellation in the solution of the quadratic I could avoid by refactoring the code. So these problems do crop up!

In case you do want to see what Julia's polynomials package does, here is the code for that

```
using Polynomials
N = 100
fib = zeros(N)
roots1 = zeros(Complex{Float64}, N)
roots2 = zeros(Complex{Float64}, N)
fib[1] = 1; roots1[1] = 1 # avoid reporting these equal
fib[2] = 1; roots1[2] = 1 # because we don't compute them
for i=3:N
    fib[i] = fib[i-2] + fib[i-1]
    r = roots(Poly([fib[i]; -2*fib[i-1]; fib[i-2]]))
    roots1[i] = r[1]
    roots2[i] = r[2]
end
bign = findfirst(roots1.==roots2) # finds the first zero
```

Problem 4: Fun with floating point (10 points)

Do one of the following two problems.

1. G&C Chapter 5 Problem 13 (10 points)

or

2. G&C Chapter 5 Problem 14 (10 points)

Problem 5: A big integral (10 points)

G&C Chapter 3 Problem 12

1. Part a. (3 points)
2. Parb b. (4 points) You *cannot use* the bad single-pass standard deviation function we discussed in class and you *must use* the alternative instead. Recall that the correct online variance computation is available in pseudo-code from Wikipedia

```
def online_variance(data):
    n = 0
    mean = 0
    M2 = 0

    for x in data:
        n = n + 1
        delta = x - mean
        mean = mean + delta/n
        M2 = M2 + delta*(x - mean)

    variance = M2/(n - 1)
    return variance
```

3. Part c. (3 points)

Problem 6: Resolving the birthday paradox (15 points)

This problem is based on G&C Chapter 3, Problem 7, but it goes far beyond the textbook.

A famous question in probability is the following: > How many people do you need in a room before there is a 50% > chance of finding two people with the same birthday? If we assume that there are 365 days in a year and people are born on a day picked uniformly at random, then we'll compute:

$$\text{Prob}(n \text{ people, one same birthday}) = 1 - \text{Prob}(n \text{ people, all distinct birthdays})$$

This second probability is straightforward to determine:

$$\text{Prob}(n \text{ people, all distinct birthdays}) = \frac{\text{number of ways to have n distinct birthdays}}{\text{number of ways to have n birthdays}}.$$

The bottom number is just 365^n because we pick birthdays totally at random. The top number is just $365 \cdot 364 \cdot 363 \cdots (365 - n + 1)$, in which case the probability is:

$$\text{Prob}(n \text{ people, one same birthday}) = \frac{365 \cdot 364 \cdot 363 \cdots (365 - n + 1)}{365^n}.$$

There's more about this problem on the Wikipedia page for the Birthday problem When n is 23, the probability is 50.7%. (Also, with 98.8% confidence I believe there are two people in our class with the same birthday because there are 56 people registered.)

1. (5 points) The problem with this mathematical model is that people are *not* born on uniformly distributed days throughout the year. I once found a distribution of people born at a hospital in New York City from 1978. It's far from uniform I've used this distribution to write a function to generate birthdays with probabilities similar to how they occurred that year. **Download the function `get_birthdays.m`** (In Julia, you can run

   ```
   download("https://www.cs.purdue.edu/homes/dgleich/cs314-2016/julia/get_birthdays.jl",
       "get_birthdays.jl")
   ```

   to download it to the current directory.) Then run

   ```
   using Plots                  # load the plotting package
   include("get_birthdays.jl") # load the data
   d = get_birthdays(10000);   # generate 10000 birthdays
   histogram(d,bins=12)
   ```

   You should see a spike in birthdays between days 200 and 275. (This is July through August). Show your figure.

2. (10 points) Now, as in the book problem (G&C Chapter 3, Problem 7), empirically evaluate the probability that a group of m people will have at least one birthday in common *using* the `get_birthdays` function. (Hint: you can test if a set of birthdays has a duplicate through

   ```
   length(unique(d)) < length(d)
   ```

   As in the book, use 10,000 samples and vary m until the probability exceeds 0.5. Report the probability for each value of m you tried.

Problem 7: Inverting the birthday paradox (15 points)

It turns out that a good rule of thumb for the birthday paradox is that if we have n items and m possible slots (or n people and $m = 365$ birthdays in the previous problems), then the probability finding two items in the same slot is

$$p(n) \approx \frac{n^2}{2m}.$$

If we plug in $n = 22$ and $m = 365$, then we get a probability of 72%, which is too large, but "close". It turns out that the birthday paradox is tremendously useful for trying to estimate the number of unique items in a database. But, we use it in reverse. Suppose that we take a random sample of n items and see k duplicates. We wish to use this to estimate how many slots there are. We'll see how you can use this to estimate the number of nodes in a large graph without looking at the graph itself.

The mathematics of deriving the following estimate get a bit involved, but the estimate is simple to use. Suppose we have an undirected graph. We can generate a random vertex in that graph by performing a random walk. It turns out that this vertex is not generated uniformly at random (that is, there is a strong bias in which vertex you see), but the following estimate corrects for that fact.

Suppose we have a sample of vertices $X_1, \ldots, X_n$ from a random walk in a graph. Let C be the total number of collisions in this sample where we have 1 collision if we see a vertex twice, 3 collisions if we see a vertex three times, 6 collisions if we see it four times, and in general $k(k-1)/2$ collisions if we see a vertex k times. We also need the degrees of each vertex that we sample, so let $d_1, \ldots, d_n$ be the degrees. Then an estimate of the number of nodes of the graph is:

$$\text{num nodes} \approx \frac{(\sum_{i=1}^n d_i)\,(\sum_{i=1}^n 1/d_i)}{2C}.$$

1. (1 point) Suppose we saw the sequence of vertices and degrees

$$\begin{bmatrix} \text{vertex} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 2 & 9 & 10 & 7 & 2 & 11 & 12 \\ \text{degree} & 5 & 9 & 2 & 3 & 2 & 8 & 8 & 3 & 9 & 4 & 3 & 8 & 9 & 2 & 3 \end{bmatrix}$$

   What does the previous estimator report as the size of this graph? (Hint, the answer I got is 311.35/8.)

2. (1 point) I wrote two commands to get links from a graph that I have attempted to anomymize. To make these work, you'll need a Julia package (ick!).

   Run this command once in julia.

```
Pkg.add("Requests")
```

   Then download

```
download("https://www.cs.purdue.edu/homes/dgleich/cs314-2016/julia/get_links.jl",
    "get_links.jl")

include("get_links.jl")
@show get_links_big(0)
@show get_links_small(0)
```

   (When I tested this, it worked from off-campus as well as from on-campus.)

3. (6 points) Implement the following two Julia functions

```
"""
`sample_vertex_small`
=====================

Return a near-random vertex from the small graph
along with it's degree.

`x,d = sample_vertex_small(k)` takes k steps of a random walk starting
from vertex 0 and returns x, the identifier of the last vertex we
visited, along with the degree of the vertex x. To get the neighbors
of the vertex, this function calls `get_links_small`

Example
-------
~~~~
@show x,d = sample_vertex_small(1)
@show x,d = sample_vertex_small(2)
~~~~
"""
function sample_vertex_small(k)
end


"""
`sample_vertex_big`
===================

Return a near-random vertex from the big graph
along with it's degree.

`x,d = sample_vertex_big(k)` takes k steps of a random walk starting
from vertex 0 and returns x, the identifier of the last vertex we
```

```
visited, along with the degree of the vertex x. To get the neighbors
of the vertex, this function calls `get_links_big`

Example
-------
~~~~
@show x,d = sample_vertex_big(1)
@show x,d = sample_vertex_big(2)
~~~~
"""
function sample_vertex_big(k)
end
```

Report the results of

```
vbig,d = sample_vertex_big(1)
vsmall,d = sample_vertex_small(1)
```

Aside on Julia programming The contents of sample_vertex_small and sample_vertex_big will be identical except you'll call `get_links_big` instead of `get_links_small`. It's generally bad programming practice to repeat code like this. One Juila construct that would help is to use a function handle. If you write:

```
f = get_links_big
```

Then the variable `f` will "point to" the `get_links_big` function and you can call it via `f(0)` to get the same output you saw in the previous problem. **You do not have to implement it this way, but it's something I wanted to tell you about.**

4. (5 points) Implement the missing pieces of this program to estimate the number of nodes in a graph. I placed this code into a Matlab script called `estimate_small.m` but you are welcome to use any name you wish.

```
nsamp = 125
X = zeros(Int64,nsamp)
d = zeros(Int64,nsamp)
C = Dict{Int64,Int64}() # this is a hash-table or dictionary
ncollisions = 0
for i=1:nsamp
    X[i],d[i] = sample_vertex_small(25)
    @show X[i]
    if haskey(C,X[i])
        # we have already seen X[i] before, update the collisions
        # COMPLETE
    else
        # then we haven't seen vertex X(i) before, record that
        # we've seen it.
        C[X[i]] = 0
    end
end
# estimate the number of nodes
# COMPLETE
```

The small graph has around 6500 vertices, and so you should see something fairly similar.

5. (2 points) Alter your code above to produce an estimate for the big graph of unknown size. Use `nsamp = 1500` and the call `sample_vertex_big(100)`. **Note** this can take almost a few hours to run! So I recommend testing with

the small graph and then just using the result from the big graph straight away.

For fun Use the julia package "ProgressMeter" to get a nice looking progress meter as this is running!

5. (0 points) Note, this estimate is derived in the paper by Liran Katzir, Edo Liberty, and Oren Somekh called Estimating Sizes of Social Networks via Biased Sampling (See section 3.) They used this idea to produce an estimate of how many nodes are on the facebook network. If you wish to play around, see how changing the values of `nsamp` and the random walk length affect your results.