HW1 (7+8(+3)) points

Welcome to HW1! There are two coding problems below (plus a 'bonus', extra credit one). Please work on them, and submit your solution .cpp files (via D2L) after naming them like so: FirstnameLastname_StudentID_HW1_Q1.cpp, FirstnameLastname_StudentID_HW1_Q2.cpp, etc. Please create a single .zip out of your multiple source files, and **upload the .zip file to the HW1 slot on D2L.**

On D2L, there's a forum called 'HW1 discussions' - please use it to ask questions, and post answers [but NOT direct answers to the HW questions :)].

Have fun! Programming is best learned by coding, so treat this as a chance to do so - also, you develop "finger memory" (typing in various non-alphanumeric characters, indenting code, etc.) that way [note that on a standard keyboard, all those extra characters ({}, [], `, ~ etc) are there, mostly for use by us programmers!].

Q1: **Square root computation** (7 points)

The assignment is to write code for numerically finding a number's square root. The built-in math library function sqrt() can already do this, of course - but this is assigned to you as a problem so you can gain coding experience.

Create a function with the following signature, then use it in main() as shown:

```
double mySqrt(double x); // eg. mySqrt(4) will return 2

...

int main() {

  double s = mySqrt(5);
  cout << s << endl; // expect 2.236

  return 0;
}// main()
```

The following will help you code mySqrt.

Given a number n, we calculate its square root via a series of converging calculations, expressed using the following pseudocode:

```
estimate = 1 // for starters, imagine that n's square root is 1

// if 1 is a square root, the 'other' square root will be n/1 (si
nce the product of the roots needs to be n),
// so our new square root estimate would be the average of the tw
o roots
newEstimate = 0.5*(estimate + n/estimate)

while the difference between 'estimate' and 'newEstimate' is high
er than a small epsilon (eg. 0.000001),
repeat the above calculation step, with 'newEstimate' becoming 'e
stimate' each time

done: output the result ('estimate' or 'newEstimate'), this is th
e square root
```

We can express the above loop as a recurrence relation:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{n}{x_n}\right), \text{ with } x_0 = 1.$$

Q2: **Random number distributions** (8 points)

In this problem, you are going to visualize random number distributions, using a sideways graph of asterisks (which you can 'cout'), eg.:

```
**********
*********
********
*******
******
*****
*****
******
*******
```

The code you are going to implement, is for random numbers that fall between 0.0 and 1.0 (not 0 to 32767, etc). Out of N random numbers that you generate (eg. N can be 1000000), you are going to count how many lie between these ranges:

```
0.0 to 0.1
0.1 - 0.2
0.2 - 0.3
....
0.9 - 1.0
```

This is called 'binning' (where we assign each random number to one of 10 bins/buckets). If the bins are numbered 0,1,2..9, then you can determine a random number r's bin using:

```
floor(r*10) // eg. if r is 0.5900345, r*10 is 5.900345, and floor
(r*10) is 5
```

Once you know what the bin# is, you can keep track of how many numbers fall in a bin, using 10 counters. Given 10 vars a0,a1,a2..a9 all initialized to 0, you can use a set of 'if' statements [or better, a 'switch' statement] to increment the appropriate counter: eg. if a random # falls in bin 5, do 'a5++' to increment that bin's counter.

If you generate N random numbers and do the above binning and track each bin's count, the total bin counts would/should add up to N:

```
a0+a1+a2+a3...a9 = N
```

Note: while this is rare, r could be 1.0, in which case, floor(r*10) will be 10 - when this happens, simply increment a9. If you don't do this, you will find that a0+a1..+a9 will be less than N, instead of being equal to it.

Rather than just 'cout' the 10 bin subtotals, you could transform each into a row of *s and cout them instead. Eg. for a5:

```
nAsterisks = floor(100 * a5/N)
cout a row of nAsterisks *s
```

So your program would print out 10 rows of *s, one row for each of a0,a1,a2.. a9.

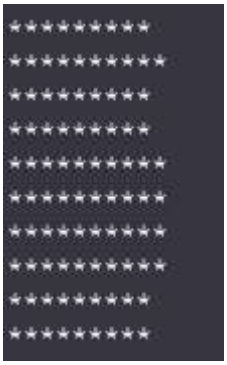
To print nAsterisks *s, you could use a function like so:

```
void printStars(int n) {
  for(int i=0;i < n;i++) {
    cout << "*";
  }
  cout << endl;
}/// printStars()

...

printStars(nAsterisks) // prints a chain of nAsterisks *s
```

When all the 10 rows of *s all have nearly equal # of *s, that means the random # distribution is 'uniform' - a random number in such a distribution has equal probability of falling in any of the 10 bins, which is why the bins end up with nearly identical counts (and therefore, *s). This is what your code would show. Here is a sample output:



So, **visualize a uniform distribution** using asterisks, as described above.

Next, **visualize a 'triangular' distribution** (eg. see http://en.wikipedia.org/wiki/Triangular_distribution (http://en.wikipedia.org/wiki/Triangular_distribution) /). To create such a distribution, simply generate two rand() values (each normalized to 0..1) and AVERAGE them together!

```
r1 = first rand#
r2 = second rand#
r = 0.5*(r1+r2) // values around 0.5 are most likely, endpoints a
t 0 and 1 are least likely
'bin' r as described above, ie do 'floor(r*10)'
```

Now your 10 rows of *s will not be all nearly identical, like they were in the uniform distribution case: you should see a peak for 0.4-0.5 and 0.5-0.6, with the * count tapering off on both sides (this makes the distribution look like a triangle). Here is a sample output:

In this modern era of high resolution displays that are in our handhelds even, why bother with visualization using plaintext *s? Because it is much simpler to do so, is a 'portable' solution, and makes for good coding practice :)

EXTRA CREDIT: **Pareto distribution** (3 points)

Create the Pareto distribution (ProbabilityDistributions.pdf) and graph it using asterisks just like specified above. Note: unlike the uniform and triangular distributions above, the Pareto distribution will not always give you values between 0 to 1! So the binning (of the value into 10 slots between 0 and 1) will not be accurate (a lot of values will get missed because they are >1, for example).

In the PDF link given above, look at the end of the 'Summary' section, for the entry on Pareto - that gives you what you need to generate the distribution.

Here's what you do (if you get values between 0 and >1 from your distribution) - loop TWICE - the first time, don't bin the values; instead, find the largest value (ie. max) that gets output; the second time, divide each value by the max you just found, to get a 0 to 1.0 result which you can now bin as usual. Pseudocode:

```
seed the rand function with a known seed
double largestVal = -10000000; // a SMALL value
for each of N times
  generate a Pareto value
  if value>largestVal, largestVal=value // we have a new largestV
al
next iteration

// now we know the largest value, it's stored in largestVal
RE-seed the rand function with the same seed used earlier (to res
tart the sequence)
for each of N times
  generate a Pareto value
  divide by largestVal // this will now be 0..1
  'bin' the result
```

But what if the min is not 0 (eg. the distribution gives you values between 1 and 3, instead of 0 to 3)? Solution: track the 'min' too in addition to tracking 'max', and normalize. Pseudocode again:

```
seed the rand function with a known seed
double largestVal = -10000000, smallestVal = 10000000;
for each of N times
  generate a Pareto value
  if value > largestVal, largestVal=value // we have a new larges
tVal
  if value < smallestVal, smallestVal=value
next iteration

// now we know the largest AND smallest values, use them to norma
lize
RE-seed the rand function with the same seed used earlier (to res
tart the sequence)
for each of N times
  generate a Pareto value 'p'
  pNorm = (p - smallestVal)/(largestVal - smallestVal) // 0..1
  do binning on pNorm
```
