STA 032 R Homework 2

Hardy Jones
999397426
Professor Melcon
Winter 2015

1. (a) 9.845806e-64
   (b) 0.02522502
   (c) 0.05937067

2. (a)

x	BinomialProb2(10, 0.10, x)
0	0.3486784401
1	0.3874204890
2	0.1937102445
3	0.0573956280
4	0.0111602610
5	0.0014880348
6	0.0001377810
7	0.0000087480
8	0.0000003645
9	0.0000000090
10	0.0000000001

   (b)

x	BinomialProb2(6, 0.2, x)
0	0.262144
1	0.393216
2	0.245760
3	0.081920
4	0.015360
5	0.001536
6	0.000064

   (c)

x	BinomialProb2(3, 0.5, x)
0	0.125
1	0.375
2	0.375
3	0.125

3. (a) 0.5606259
   (b) 0.9347622
   (c) 0.9960579

4. (a) i. 32
   ii. 10
   iii. 2
   (b) i. 0.027032
   ii. 0.038524
   iii. 0.016892

Appendix A R code

Problem 1

```
BinomialProb <- function(n, p, x) {
  # We put the `n` and `p` arguments first
  # in an attempt to make things more composable.
  # Of course, it doesn't really matter as the function isn't curried.
  # If it were, it could provide usage such as
  # prob_n_p <- BinomialProb(n)(p)
  # prob_n_p_x <- foo(x)

  # The following law should hold
  # forall n, p, x. dbinom(x, n, p) == BinomialProb(n, p, x)
  # for some definition of `==`

  # Use a direct translation of the probability.
  choose(n, x) * (p ^ x) * ((1 - p) ^ (n - x))
}
```

Problem 2

```
source("./prob1.R")

BinomialProb2 <- function(n, p, x) {
  # We can use `BinomialProb` to just `sapply` each element of `x`.
  # This is a case where the curried version would make things clearer.
  # We could say:
  # sapply(x, BinomialProb(n)(p))

  sapply(x, function(x1) { BinomialProb(n, p, x1) })
}
```

Problem 3

```
source("./prob2.R")

BinomialWithinK <- function(n, p, k) {
  # Using the hint from the description, we calculate `mu` and `sigma`,
  # then use these in order to find the lower and higher bounds.
  # Next we compute the binomial probability with `BinomialProb2` and
  # `sum` it all up.

  mu    <- n * p
  sigma <- k * sqrt(mu * (1 - p))
  low   <- ceiling(mu - sigma)
  high  <- floor(mu + sigma)

  sum(BinomialProb2(n, p, (low:high)))
}
```

Problem 4

(a)

```r
GeometricRv <- function(p) {
  # We just recursively count the number of iterations.
  # The function is pure, and we only need to worry about recursion depth.
  # However, since we're using decent sized probabilities,
  # we shouldn't hit the limit.
  go <- function(count) {
    if (runif(1) < p) count else go(count + 1)
  }

  # Start it off.
  go(1)
}
```

(b)

```r
source("./prob4a.R")

Trials <- function(r, n, p) {
  # We simulate `n` runs by just calling `GeometricRv` and
  # comparing the output to `r` that many times.
  simulate <- sapply(1:n, function(x) {
    GeometricRv(p)
  })

  # Count how many are equal to `r`, and divide by `n`.
  # This is our probability.
  length(Filter(function(x) {x == r}, simulate)) / n
}
```