STA312

Chapter 3: Methods for Generating Random Variables

Part I

Dr. Luai Al Labadi

Spring 2020

Introduction

- One of the fundamental tools in computational statistics is to simulate random variables from certain probability distributions.

- **Example:** gamma, binomial, $f_X(x) = 3x^2$, $0 < x < 1$, ...

- **Overall Goal:** Generate sample of a random variable X with a given density f_X.

- The sample is called a *random variate*.

- **What does this mean?**

  - Develop an algorithm such that if one used it repeatedly (and independently) to generate a sequence of samples $X_1, X_2, \dots, X_n$ then as n becomes large, the proportion of samples that fall in any interval $[a, b]$ is close to $P(X \in [a, b])$, i.e.

$$\frac{\# X_i \in [a, b]}{n} \approx P(X \in [a, b]).$$

Generating Random Numbers

- Generating random variates from many probability distributions depends on the uniform random number generator.
- Therefore, it necessary to explain how random numbers are generated?
  - We are looking at procedures which give successive outcomes that are independent and equally likely.
  - **Examples (physical methods):**
    - Successive tosses of a fair coin yield 0s (tails) or 1s (heads) at random.
    - Rolling a fair die repeatedly produces a sequence of numbers 1 through 6.
  - **How to generate hundreds of thousands or millions of random numbers?**

Linear Congruential Generators

- **How it is possible to get random numbers from a computer?**
- Strictly speaking, it is **NOT** possible.

- However, with ingenuity and care, programs can be devised to produce pseudo-random numbers.
  - That is, numbers that behave, for practical purposes, as if they were random.

- In what follows, we discuss how it is possible for the "random" procedures to achieve an excellent and useful illusion of randomness.

Linear Congruential Generators

- A very common method of generating pseudo-random numbers on computers is called a **linear congruential generator**.
- First, we state and illustrate the general "linear" formula for such a generator.

The Expected Value of a Function

- A linear congruential generator produces integers r_i iteratively according to the mathematical formula:

$$r_{i+1} = ar_i + b(mod\, d)$$

for integers $a > 0$, $b \geq 0$, and $d > 0$, where $i = 1, 2, \dots$.

  - The notation mod, short for modulo, means that r_{i+1} is the remainder when $ar_i + b$ is divided by d.
  - For example, we say that 3 is equal or "congruent" to 8 mod 5.
  - Here, a is called the **multiplier**, b the **increment**, and d the **modulus** of the generator.

Linear Congruential Generators

- The process start with a positive integer **seed** $s = r_1 < d$.
- For suitably chosen a, b, and d, the previous formula can **shuffle** integers from among $0, 1, 2, \ldots, d-1$ in ways that are not random but that pass many of the tests that randomly shuffled integers should pass.
- If $b = 0$, then the generator is called multiplicative and r_i cannot take the value 0.
- The procedure requires only simple computations. Thus, numbers can be generated fast enough to support practical computer simulations
- Following standard terminology, we sometimes refer to the results from a linear congruential generator as “random numbers”.

Example: $a = 5$, $b = 3$, $d = 16$, and seed$=s = r_1 = 7$.

R Code


```
> d = 16 # modulus
> a = 5 # multiplier
> b = 3 # shift
> s = 7 # seed
> m = 20 # length of run (counting seed as #1)
> r = numeric(m) # initialize vector for random integers
> r[1] = s # set seed
> for (i in 1:(m-1)) r[i+1] = (a * r[i] + b) %% d
> # generates random integers
> r # list of random integers generated
 [1]  7  6  1  8 11 10  5 12 15 14  9  0  3  2 13  4  7
6  1  8
>
```

Linear Congruential Generators

- It is customary to rescale the output by a congruential generator to fit into the interval $(0, 1)$.

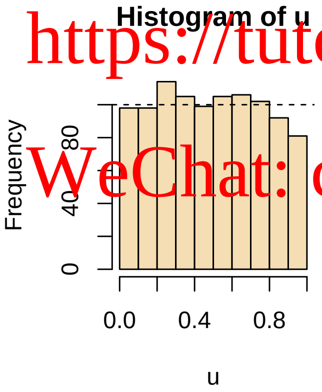
- For example, if a generator with modulus d takes values $0, 1, \ldots, d-1$ then we can use $u_i = \frac{r_i+0.5}{d}$.

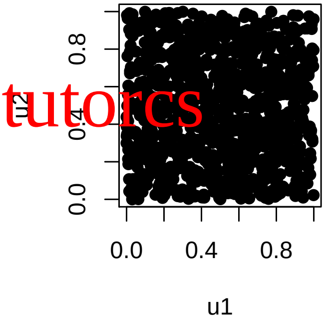
Example: $a = 1093$, $b = 18257$, $d = 86436$, and seed$=s = r_1 = 7$.

R Code


```
> # Initialize
> a = 1093; b = 18257; d = 86436; s = 7
> m = 1000; r = numeric(m); r[1] = s
> # Generate
> for (i in 1:(m-1)) {r[i+1] = (a*r[i] + b) %% d}
> u = (r + 1/2)/d # values fit in (0,1)
> # Display Results
> par(mfrow=c(1,2), pty="s") # 2 square panels in a plot
> hist(u, breaks=10, col="wheat") # left panel
> abline(h=m/10, lty="dashed")
> u1 = u[1:(m-1)]; u2 = u[2:m] # right panel
> plot (u1, u2, pch=19)
> par(mfrow=c(1,1), pty="m") # return to default
```

Outcome



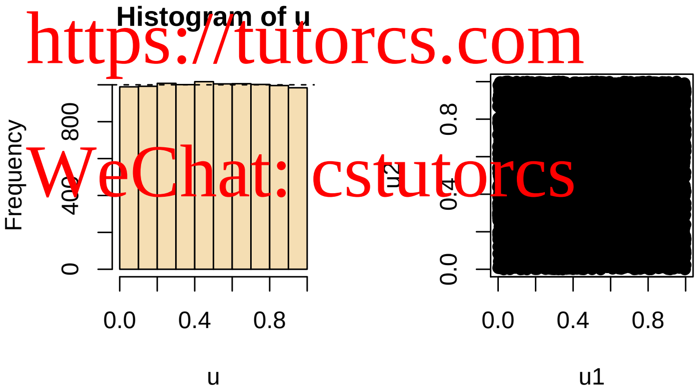


Example: $a = 1093$, $b = 18257$, $d = 86436$, and seed=$s = r_1 = 7$.

R Code


```
> # Initialize
> a = 1093; b = 18257; d = 86436; s = 7
> m = 10000; r = numeric(m); r[1] = s
> # Generate
> for (i in 1:(m-1)) {r[i+1] = (a*r[i] + b) %% d}
> u = (r + 1/2)/d # values fit in (0,1)
> # Display Results
> par(mfrow=c(1,2), pty="s") # 2 square panels in a plot
> hist(u, breaks=10, col="wheat") # left panel
> abline(h=m/10, lty="dashed")
> u1 = u[1:(m-1)]; u2 = u[2:m] # right panel
> plot (u1, u2, pch=19)
> par(mfrow=c(1,1), pty="m") # return to default
```

Outcome


Histogram of u
Frequency
0
400
800
0.0
0.4
0.8
u
u2
0.0
0.4
0.8
0.0
0.4
0.8
u1

Uniform Pseudo-random Numbers in R

- The uniform pseudo random number generator in **R** is runif.
- To generate a vector of n (pseudo) random numbers between 0 and 1 use $runif(n)$.

- To generate n random Uniform(a, b) numbers use $runif(n, a, b)$.
- To generate an n by m matrix of random numbers between 0 and 1 use $matrix(runif(n*m), nrow = n, ncol = m)$.

Example: Generate a sample of size 5 from $U(0,1)$.

R Code

```
> runif(5)
 [1] 0.9429808 0.8682551 0.5160173 0.1876722 0.4380817
> runif(5)
 [1] 0.4192251 0.5517926 0.3325744 0.5930924 0.3459959
```


- We get two different samples.
- **How to get the exact same sample?**
- **Answer:** Use “set.seed”.

set.seed

R Code


```
> set.seed(1)
> runif(5)
 [1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819
> set.seed(1)
> runif(5)
 [1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819
> set.seed(1234)
> runif(5)
 [1] 0.1137034 0.6222994 0.6092747 0.6233794 0.8609154
> set.seed(1234)
> runif(5)
 [1] 0.1137034 0.6222994 0.6092747 0.6233794 0.8609154
```

set.seed

- In the overwhelming majority of cases, we do not set the seed.
- It is usually used in settings where we need to reproduce the exact same sequence of random simulations.

- **Example:** Compare two procedures or two speeds.

How to Simulate any Probability Distribution

2-step Process

1. Generate a random variate uniformly distributed in $[0, 1]$ (also called a random number).

2. Use an appropriate transformation to convert the random number to a random variate of the correct distribution.

- Why is this approach makes sense? [**answer in the next slides**]

The Inverse Transform Method: Continuous Case

Goal: How does one transform a sample of the *uniform*[0, 1] random variable into a sample of a given distribution?

- The inverse transform method of generating random variables is based on the following well known result.

Theorem (Probability Integral Transformation)

If X is a continuous random variable with cdf $F_X(x)$, then $U = F_X(X) \sim Uniform(0, 1)$.

Proof

Corollary

Let $U \sim Uniform(0,1)$. Define $X = F^{-1}(U)$, where F is a cdf. Then F is the cdf of X.

Proof:

The Inverse Transform Method

- **The method (ALGORITHM) can be summarized as follows:**

  1. Find $F_X(x)$.
  2. Derive the inverse function $F_X^{-1}(u)$.
  3. Write a command or function to compute $F_X^{-1}(u)$.
  4. For each random variate required:
     (a) Generate a random u from Uniform(0, 1).
     (b) Deliver $x = F_X^{-1}(u)$.

- **Assumption:** F_X^{-1} exist (easy to compute).

Example

Use the inverse transform method to simulate a random sample from the distribution with density

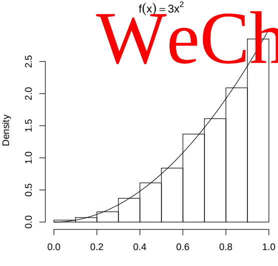
$$f_X(x) = 3x^2, 0 < x < 1$$

Write the algorithm.

R Code

R Code

```
>     n <- 1000
>     u <- runif(n)
>     x <- u^(1/3)
>     hist(x, prob = TRUE, main = bquote(f(x)==3*x^2))
 #density histogram of sample
>     y <- seq(0, 1, .01)
>     lines(y, 3*y^2)     #density curve f(x)
```

The histogram and density plot suggests that the empirical and theoretical distributions approximately agree.

Example: Exponential

Use the inverse transform method to simulate a random sample from the distribution with density

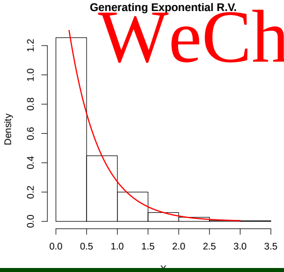
$$f_X(x) = \lambda e^{-\lambda x}, x > 0$$

Write the algorithm. **Note:** in this form of pdf, λ is called the rate as $E(X) = 1/\lambda$.

R Code: $\lambda = 2$

R Code

```
> # Inverse Transform Sampling
> num.samples <- 1000 # number of samples
> U           <- runif(num.samples) # generate U
> X           <- -log(1-U)/2 # return X
> # plot
> hist(X, freq=F, xlab='X', main='Generating Exponential R.
> curve(dexp(x, rate=2) , 0, 3, lwd=2, xlab = "", ylab = ""
```

The histogram and density plot suggests that the empirical and theoretical distributions approximately agree.

Example: Weibull

The Weibull density is

$$f_X(x) = \begin{cases} \frac{\alpha}{\beta}\left(\frac{x}{\beta}\right)^{\alpha-1} e^{-\left(\frac{x}{\beta}\right)^{\alpha}} & x \geq 0 \quad \alpha > 0 \quad \beta > 0 \\ 0 & \text{otherwise} \end{cases}.$$

Let $\beta = 1$.

(a) Show that $f_X(x)$ is a valid pdf.

(b) Show that $E(X) = \Gamma(1 + \alpha^{-1})$.

(c) Find the cumulative distribution function $F_X(x)$.

(d) Find $F_X^{-1}(x)$.

(e) Develop an algorithm to generate a sample of size 10^4 from $Weibull(\alpha)$ distribution using the inverse-transform method.

(f) Now, based in the algorithm in (e), simulate a sample of size 10^4 from $Weibull(\alpha)$ for the values of $\alpha = 0.5, 1, 4$ and 10. Plot the relative frequency histogram and the corresponding density on the same picture. Check that the mean is close to the theoretical mean $\Gamma(1 + \alpha^{-1})$.

R Code: Weibull

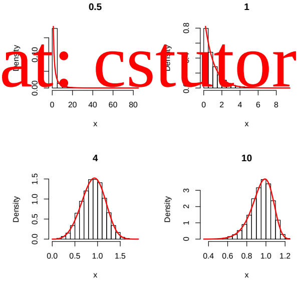
R Code


```
> n <- 10000
> alpha <- c(0.5, 1, 4, 10)
> sample.mean=rep(NA, length(alpha))
> par(mfrow=c(2,2))
> for (i in 1:length(alpha)) {
+ u <- runif(n)
+ x <- (- log(1-u))^(1/alpha[i])
+ sample.mean[i]=mean(x)
+ hist(x, prob=TRUE, main=alpha[i])
+ curve(dweibull(x, alpha[i], scale = 1), lwd=2, xlab = "",
+ }
> exact.mean=gamma(1+alpha^(-1))
```

R Code: Weibull

R Code

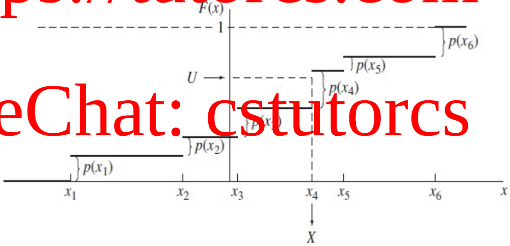
```
> cbind(alpha, exact.mean,sample.mean)
     alpha exact.mean sample.mean
[1,]   0.5  2.0000000   2.0546442
[2,]   1.0  1.0000000   0.9911315
[3,]   4.0  0.9064025   0.9046273
[4,]  10.0  0.9513508   0.9514688
```

Discrete Case

- Suppose $x_1 < x_2 < \ldots < x_m$ then the cdf of X is

$$F_X(x) = \begin{cases} 0 & x < x_1 \\ \sum_{i=1}^{j} p_i & x_j \leq x < x_{j+1}, \quad j \leq m-1. \\ 1 & x \geq x_m \end{cases}$$



$$P(X = x_j) = P\left[F_X(x_{j-1}) < U \leq F_X(x_j)\right] = F_X(x_j) - F_X(x_{j-1}) = p_j.$$

Generalized Inverse

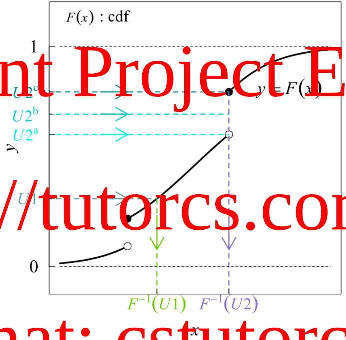


Figure: Source: wikipedia.org.

Definition (Inverse Function)

$$F_X^{-1}(u) = \inf\{x : F_X(x) \geq u\}, 0 < u < 1.$$

The Inverse Transform Method: Discrete Case

Goal: Want to generate a discrete random variable X with pmf

$$P(X = x_i) = p_i, \quad i = 1, \dots, \quad \sum_i p_i = 1$$

- **The method (ALGORITHM) can be summarized as follows:**

1. Generate a random u from Uniform$(0, 1)$.
2. Transform u into X as follows,

$$X = x_j \text{ if } F_X(x_{j-1}) < u \le F_X(x_{j-1})$$

That is,

$$X = x_j \text{ if } \sum_{i=1}^{j-1} p_i < u \le \sum_{i=1}^{j} p_i$$

The Inverse Transform Method: Discrete Case

So, discrete random variables can be generated by slicing up the interval (0, 1)) into subintervals which define a partition of (0, 1):

$$(0, F_X(x_1)], (F_X(x_1), F_X(x_2)], (F_X(x_2), F_X(x_3)], \ldots, (F_X(x_{k-1}), 1],$$

generating $U = U$niform$(0, 1)$ random variables, and seeing which subinterval U falls into. It follows that,

$$X = \begin{cases} x_1 & u \le F_X(x_1) = p_1 \\ x_2 & F_X(x_1) = p_1 < u \le F_X(x_2) = p_1 + p_2 \\ \vdots & \\ x_j & F_X(x_{j-1}) = \sum_{i=1}^{j-1} p_i < u \le F_X(x_j) = \sum_{i=1}^{j} p_i \\ \vdots & \end{cases}$$

Proof: Why the Algorithm Works?

Example: $Bernoulli(p = 0.4)$

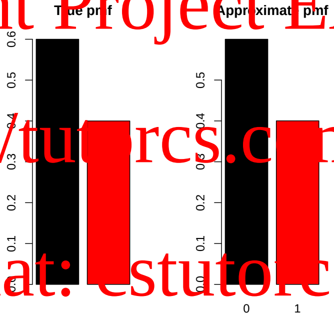
Simulate the random variable $X \sim Bernoulli(p = 0.4)$. That is $P(X = 1) = 0.4 = 1 - P(X = 0)$.

R Code: $Bernoulli(p = 0.4)$

R Code

```
> n <- 10000
> p <- 0.4
> u <- runif(n)
> x <- as.integer(u > 0.6)
 #(u > 0.6) is a logical vector
> mean(x)
 [1] 0.4002
> var(x)
 [1] 0.240064
> rbind(table(x)/length(x), c(p,1-p))
          0       1
 [1,] 0.5998 0.4002
 [2,] 0.4000 0.6000
> par(mfrow=c(1,2))
> barplot(c(1-p,p), col=c("1","2"), main='True pmf')
> barplot(table(x)/length(x) , col=c("1","2"),
 main='Approximate pmf')
```

R Code: $Bernoulli(p = 0.4)$

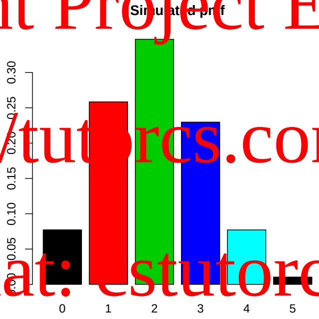


How to Generate a Sample from $Binomial(n, p = 0.4)$?

R Code

```
> n=5
> p=0.4
> m = 10^5 # replicates
> x = numeric(m)
> for (i in 1:m) {
+ u = runif(n)
+ b = (u < .4) # logical vec b
+ x[i] = sum(b)  }
> mean.sample=mean(x)
> mean.exact=n*p
> table(x)/m
x
      0       1       2       3       4       5
0.07727 0.25843 0.34705 0.22973 0.07715 0.01037
> barplot(table(x)/m, col=c("1","2","3","4","5"),
main='Simulated pmf')
```

R Code: $Binomial(n = 5, p = 0.4)$



Example: Simple pmf

Simulate the random variable X with the pmf

$$p_1 = 0.1, p_2 = 0.2, p_3 = 0.2, p_4 = 0.2, p_5 = 0.3$$

R Code

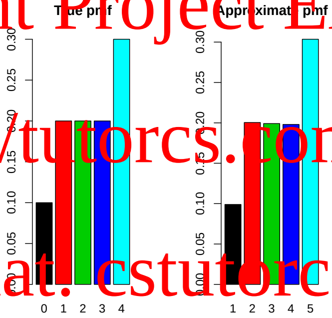
R Code


```
> #Method 1
> discrete.inv.transform.sample <- function( p ) {
+   U  <- runif(1)
+   if(U <= p[1]){
+     return(1)
+   }
+   for(state in 2:length(p)) {
+     if(sum(p[1:(state-1)]) < U && U <= sum(p[1:state]) )
+     return(state)
+     }
+   }
+ }
> num.samples <- 50000 # number of sample
> p <- c(.1, .2, .2, .2, .3) # probabilities
> names(p) <- 0:4
> samples      <- numeric(num.samples)
```

R Code


```
> for(i in seq_len(num.samples)) {
+   samples[i] <- discrete.inv.transform.sample(p)
+ }
> rbind(table(samples)/num.samples, p)
        1       2       3     4       5
  0.09894 0.20042 0.19912 0.198 0.30352
p 0.10000 0.20000 0.20000 0.200 0.30000
> par(mfrow=c(1,2))
> barplot(p, col=c("1","2","3","4","5"), main='True pmf')
> barplot(table(samples)/num.samples,
 col=c("1","2","3","4","5"), main='Approximate pmf')
```

R Code



Example: Geometric

Use the inverse transform method to simulate a random sample from the distribution with pmf

$$p_i = P(X = i) = pq^i, i = 0, 1, 2, \ldots, q = 1 - p.$$

Write the algorithm.

R Code

```
> n <- 10000
> p <- 0.25
> u <- runif(n)
> k <- ceiling(log(1-u) / log(1-p)) - 1
>
> # more efficient
> k <- floor(log(u) / log(1-p))
>
> mean.sample=mean(k)
> mean.exact=(1-p)/p
> cbind(mean.sample,mean.exact)
     mean.sample mean.exact
[1,]      3.0045          3
```

Example: Binomial

For the pmf

$$p_i = P(X = i) = \binom{n}{i} p^i (1-p)^{n-i}, i = 0, 1, \ldots, n.$$

1. Derive the recursive formula

$$p_{i+1} = \left(\frac{n-i}{i+1}\right)\left(\frac{p}{1-p}\right) p_i, \quad i = 0, 1, \ldots, n.$$

2. Use the inverse transform method to simulate a random sample X with the above pmf. Write the algorithm.

Example: Logarithmic Distribution

A random variable X has the logarithmic distribution if

$$p_X(x) = P(X = x) = \frac{a\theta^x}{x}, \quad x = 1, 2, \ldots,$$

where $0 < \theta < 1$ and $a = (-\log(1-\theta))^{-1}$.

1. Derive the recursive formula

$$p_X(x+1) = \left(\frac{\theta x}{x+1}\right) p_X(x), \quad x = 1, 2, \ldots.$$

2. Use the inverse transform method to simulate a random sample X with the above pmf. Write the algorithm.

Solution

1. Since $p_X(x) = P(X = x) = \frac{a\theta^x}{x}$, we have

$$\frac{p_X(x+1)}{p_X(x)} = \frac{\frac{a\theta^{x+1}}{x+1}}{\frac{a\theta^x}{x}} = \frac{x}{x+1}\theta.$$

Thus, $p_X(x+1) = \left(\frac{\theta x}{x+1}\right) p_X(x)$

Solution

Recall:

$$p_X(i) = P(X = i) = \frac{a\theta^i}{i}, \quad i = 1, 2, \ldots$$

and

$$p_X(i+1) = \left(\frac{\theta i}{i+1}\right) p_X(i)$$

2. **Algorithm:**

   (i) Let $i = 1, p = a\theta, F = p$

   (ii) If $U \leq F$, set $x = i$ and stop.

   (iii) $p = \frac{\theta i}{i+1} p$, $F = F + p$, $i = i + 1$.

   (vi) Go t step (iii).

Note: $i = i + 1$ means the value of i should be increased by 1.

R Code

```
> # function to generate from a logarithmic distribution
> simLogarithmic <- function(n.gen,theta){
+ urandom <- runif(n.gen)
+ sim.vector <- rep(0,n.gen)
+ for(j in 1:n.gen){
+ i <- 1
+ a=(-log(1-theta))^(-1)
+ p <- a*theta
+ F <- p
+ while(urandom[j] >= F){
+ p <- theta*i*p/(i+1)
+ F <- F+p
+ i<- i+1
+ }
+ sim.vector[j] <- i
+ }
+ # output
```

R Code


```
+ sim.vector
+ }
> n.gen=10000
> theta=0.5
> out1 <- simLogarithmic(n.gen,theta)
> sim.mean=mean(out1)
> exact.mean= (-1/log(1-theta))*(theta/(1-theta))
> cbind(sim.mean,exact.mean)
     sim.mean exact.mean
[1,]   1.4406   1.442695
```

R Code


```
> table(out1)/n.gen
out1
     1      2      3      4      5
0.7200 0.1822 0.0612 0.0222 0.0076
     6      7      8      9     10     11
  0.0042 0.0012 0.0010 0.0001 0.0001 0.0001
    12
0.0001
> barplot(table(out1)/n.gen,
main='Approximate pmf logarithmic distribution', )
```

R Code


Approximate pmf logarithmic distribution
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
1 2 3 4 5 6 7 8 9 10 12