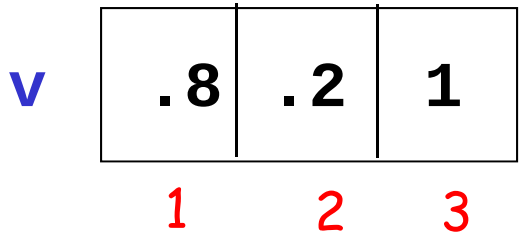
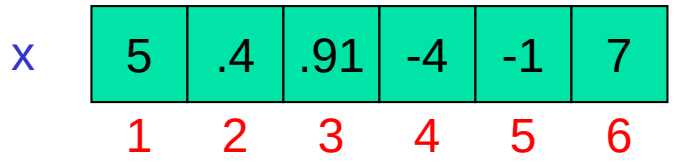
- Previous Lecture:
  - User-defined functions
    - Local memory space
    - Subfunctions
- Today's Lecture:
  - 1-d array—vector
  - Probability and random numbers
  - Simulation using random numbers, vectors
- Announcements:
  - Project 3 posted; due Monday 10/1
  - Discussion this week in classrooms as listed on the roster
  - Prelim 1 next Thursday, 10/4, at 7:30pm

1-d array: **vector**

- An array is a named collection of like data organized into rows or columns
- A 1-d array is a row or a column, called a *vector*
- An *index* identifies the position of a value in a vector



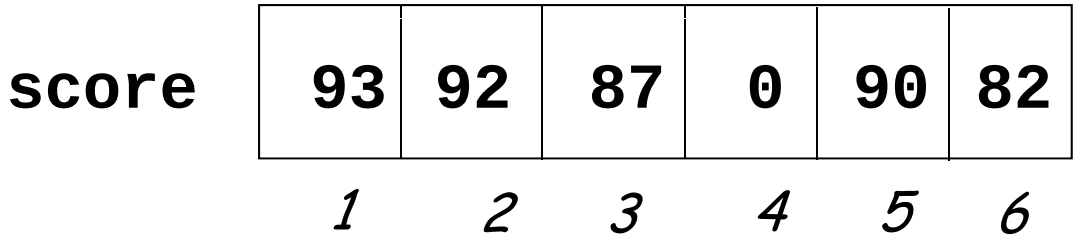
Array index starts at 1



Let k be the index of vector x, then

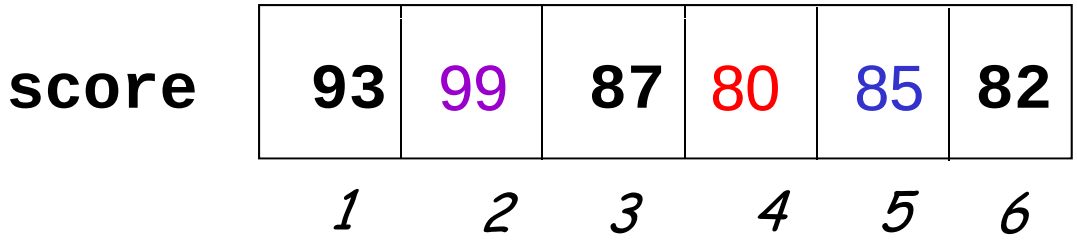
- k must be a positive integer
- 1<= k <= length(x)
- To access the k^{th} element: x(k)

Accessing values in a vector



Given the vector **`score`** …

Accessing values in a vector

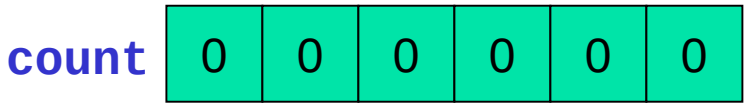


Given the vector **score** …

```
score(4)= 80;
score(5)= (score(4)+score(5))/2;
k= 1;
score(k+1)= 99;
```

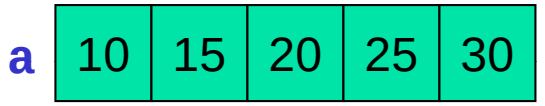
Here are a few different ways to create a vector

```
count= zeros(1,6)
```

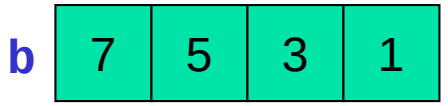


Similar functions: **ones**, **rand**

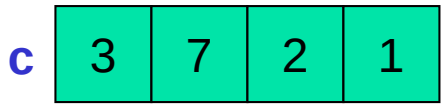
```
a= linspace(10,30,5)
```



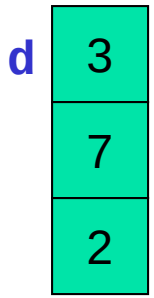

```
b= 7:-2:0
```



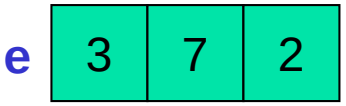

```
c= [3 7 2 1]
```




```
d= [3; 7; 2]
```




```
e= d'
```

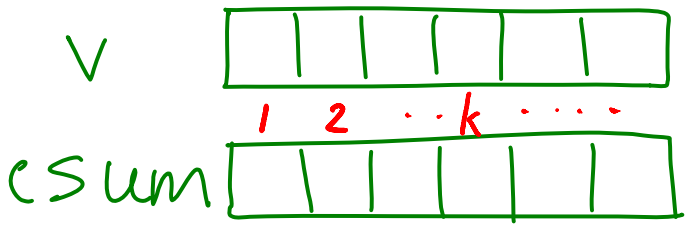


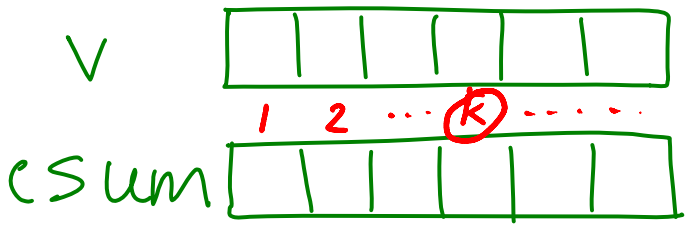
Example

- Write a program fragment that calculates the cumulative sums of a given vector **v**.
- The cumulative sums should be stored in a vector of the same length as **v**.

1, 3, 5, 0 **v**

1, 4, 9, 9 cumulative sums of **v**


v
1 2 .. k ....
csum



$csum(k) = csum(k-1) + V(k)$

$csum(3) = V(1) + V(2) + V(3)$

$csum(4) = \underbrace{V(1) + V(2) + V(3)}_{csum(3)} + V(4)$

```
csum(1) = V(1);
for k = 2 : length(V)
    csum(k) = csum(k-1) + V(k);
end
```

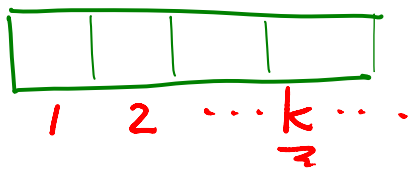
Example

- Write a function **`evalPoly`** to evaluate an n^{th} order polynomial of x :

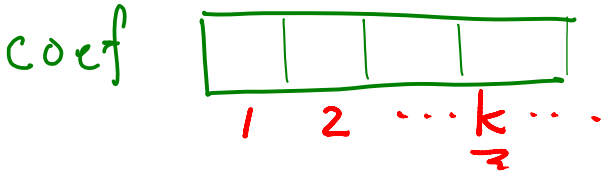
$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

- Input parameter **`coef`** has length $n+1$, contains the coefficients of the polynomial
- **`coef(1)`** is the coefficient for the term x^0
- Input parameter **`x`**
- Return the value of the polynomial evaluated at x
- No Matlab predefined function other than **`length`**

coef



$$c_1 x^0 + c_2 x^1 + c_3 x^2 + c_4 x^3$$



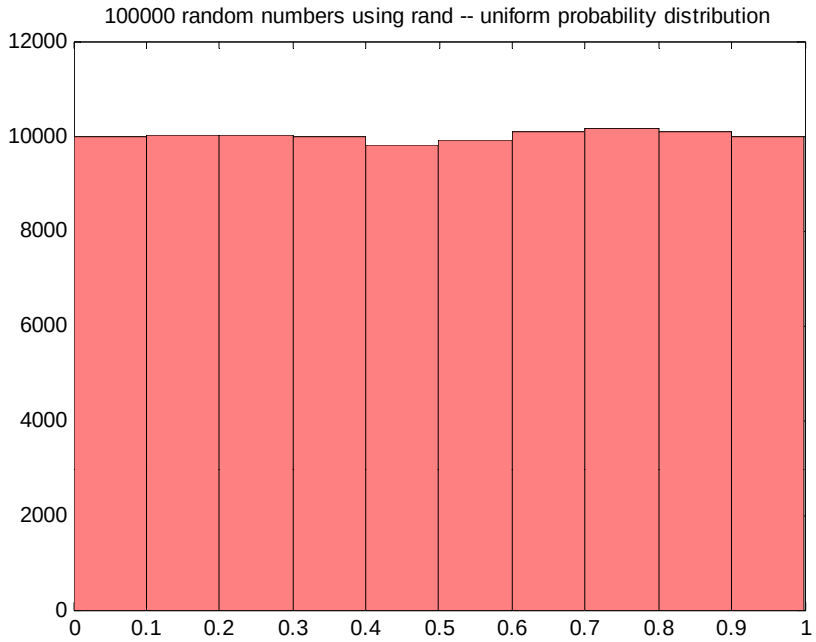
$c_1 x^0 + c_2 x^1 + c_3 x^2 + c_4 x^3$

```
function  val = evalPoly(coef, x)
% val is polynomial evaluated at x
% coef is a vector where coef(1) is for term x^0

val = coef(1)
for  k = 2:length(coef)
     val = val + coef(k) * x^(k-1);


end
```

Random numbers

- *Pseudorandom* numbers in programming
- Function `rand(...)` generates random real numbers in the interval (0,1). All numbers in the interval (0,1) are equally likely to occur—uniform probability distribution.
- Examples:

`rand(1)`	one random # in (0,1)
`6*rand(1)`	one random # in (0,6)
`6*rand(1)+1`	one random # in (1,7)

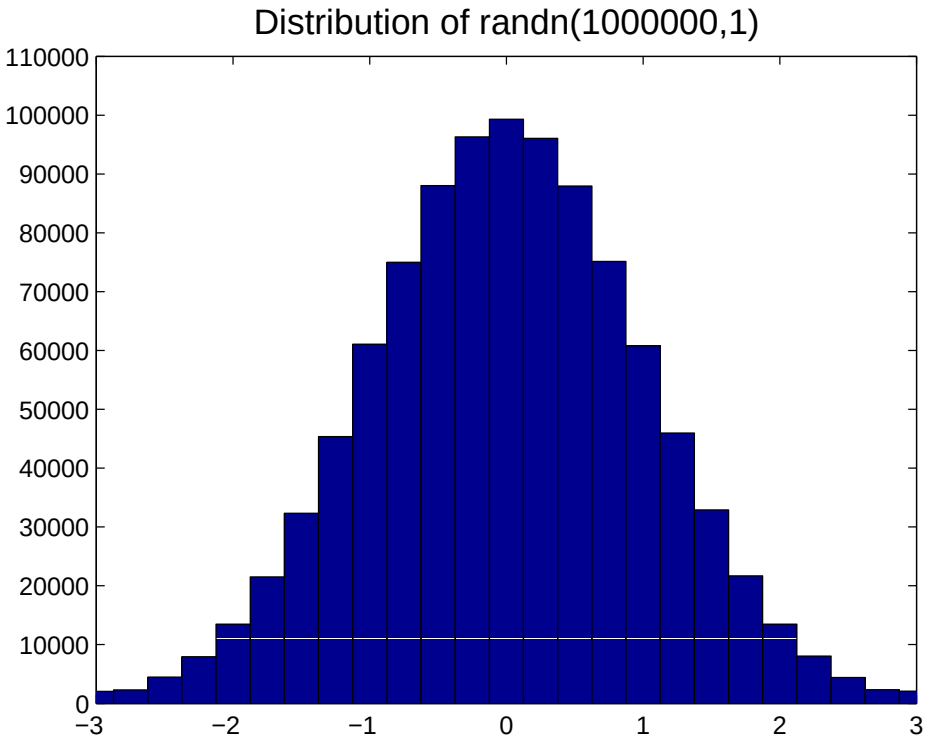


Uniform probability distribution in (0,1)

rand

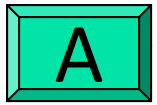
Normal distribution with zero mean and unit standard deviation

randn

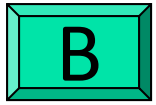


Simulate a fair 6-sided die

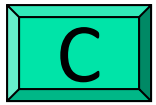
Which expression(s) below will give a random *integer* in [1..6] with equal likelihood?



`round(rand*6)`

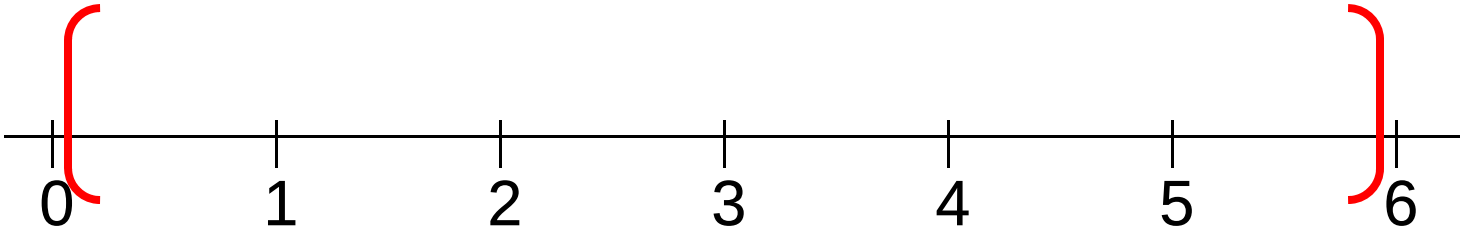


`ceil(rand*6)`

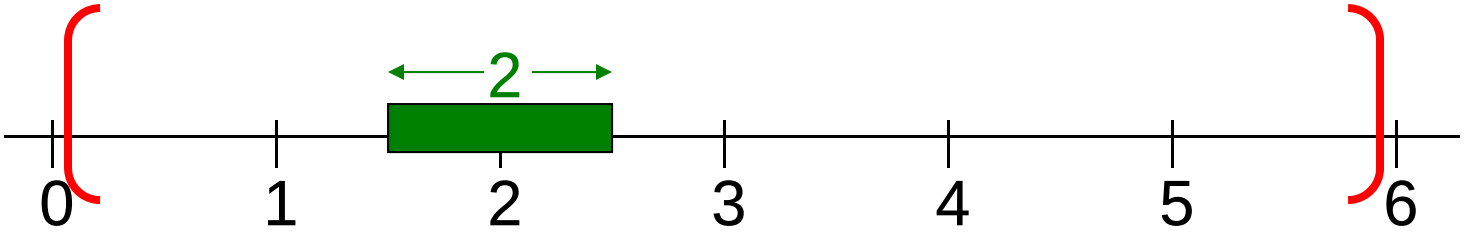


Both expressions above

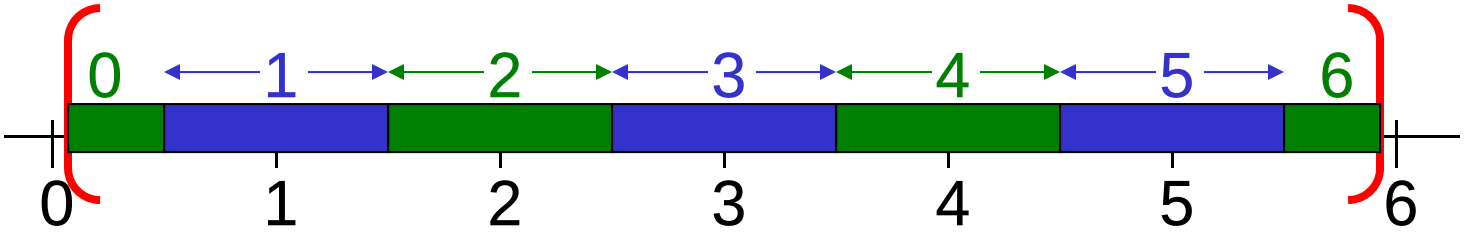
`(rand(1)*6)`



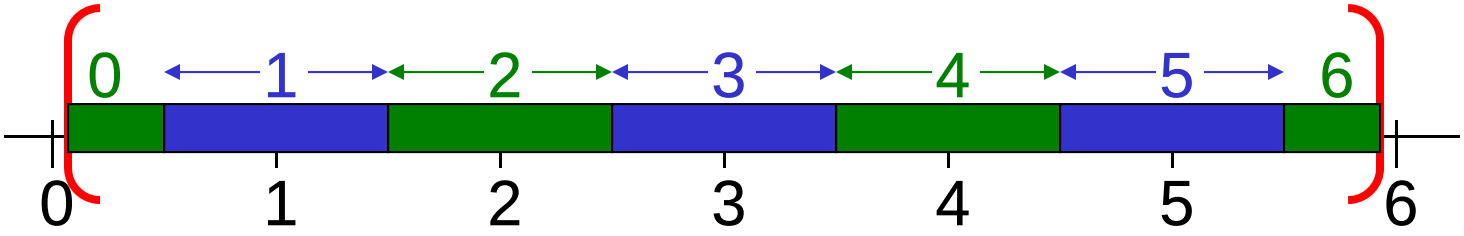
`round(rand(1)*6)`



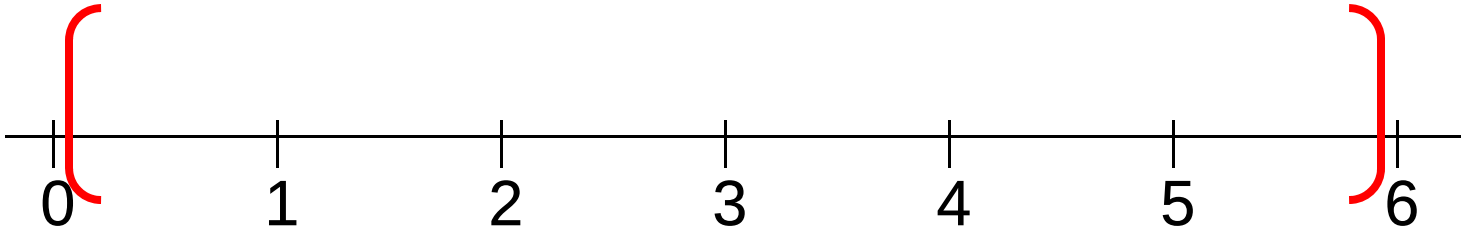
`round(rand(1)*6)`



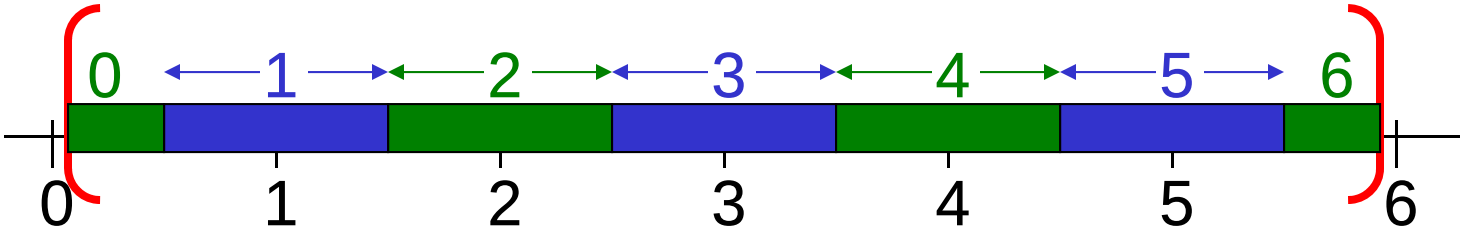
`round(rand(1)*6)`



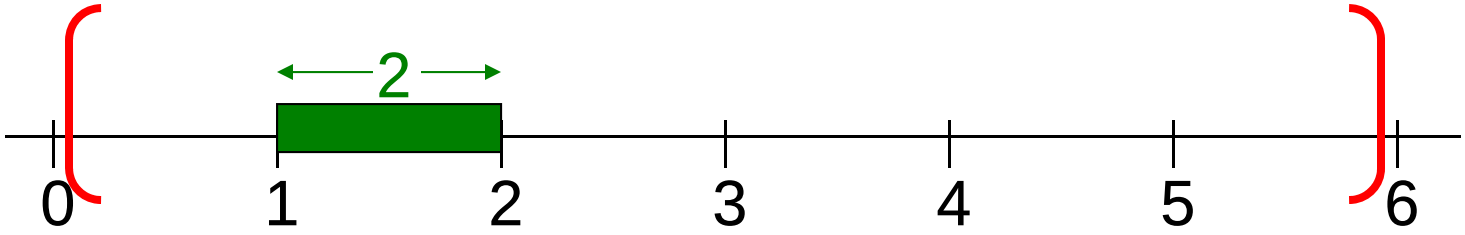
`(rand(1)*6)`



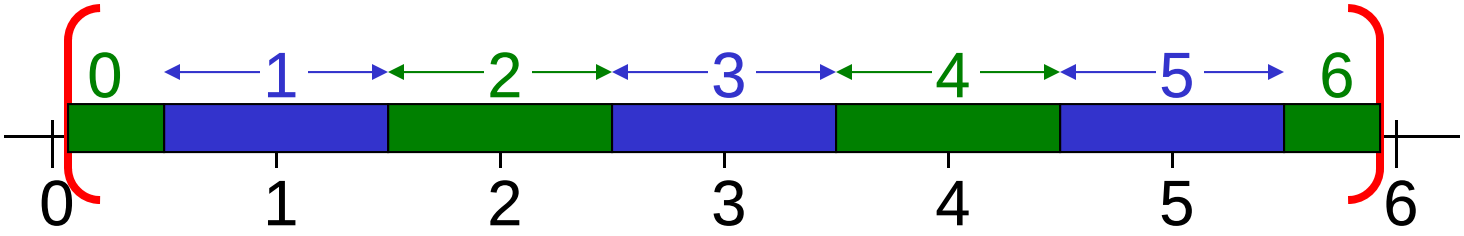
`round(rand(1)*6)`



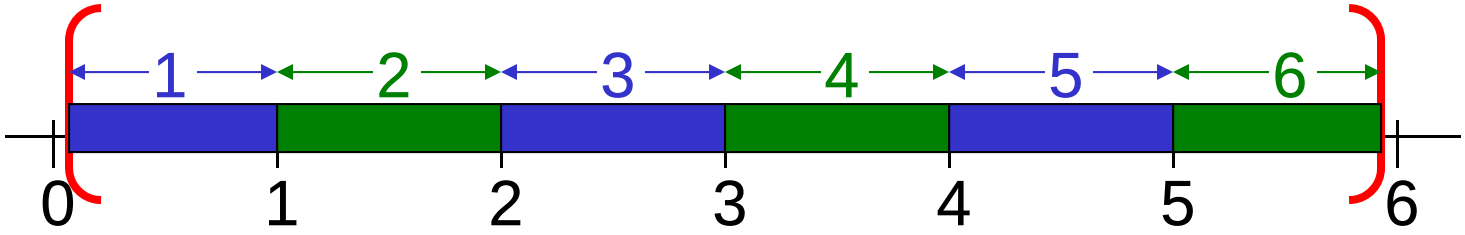
`ceil(rand(1)*6)`



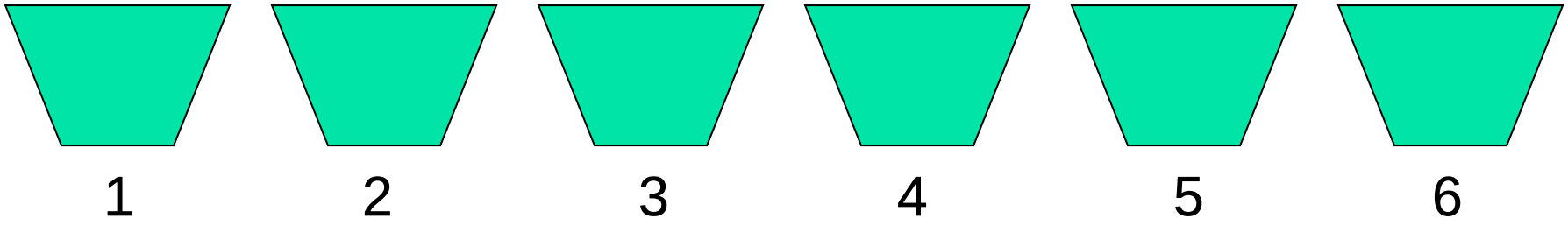
`round(rand(1)*6)`



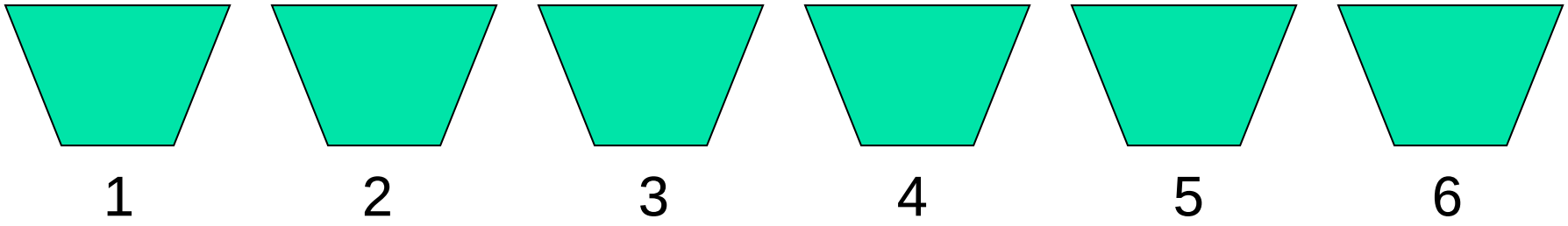
`ceil(rand(1)*6)`



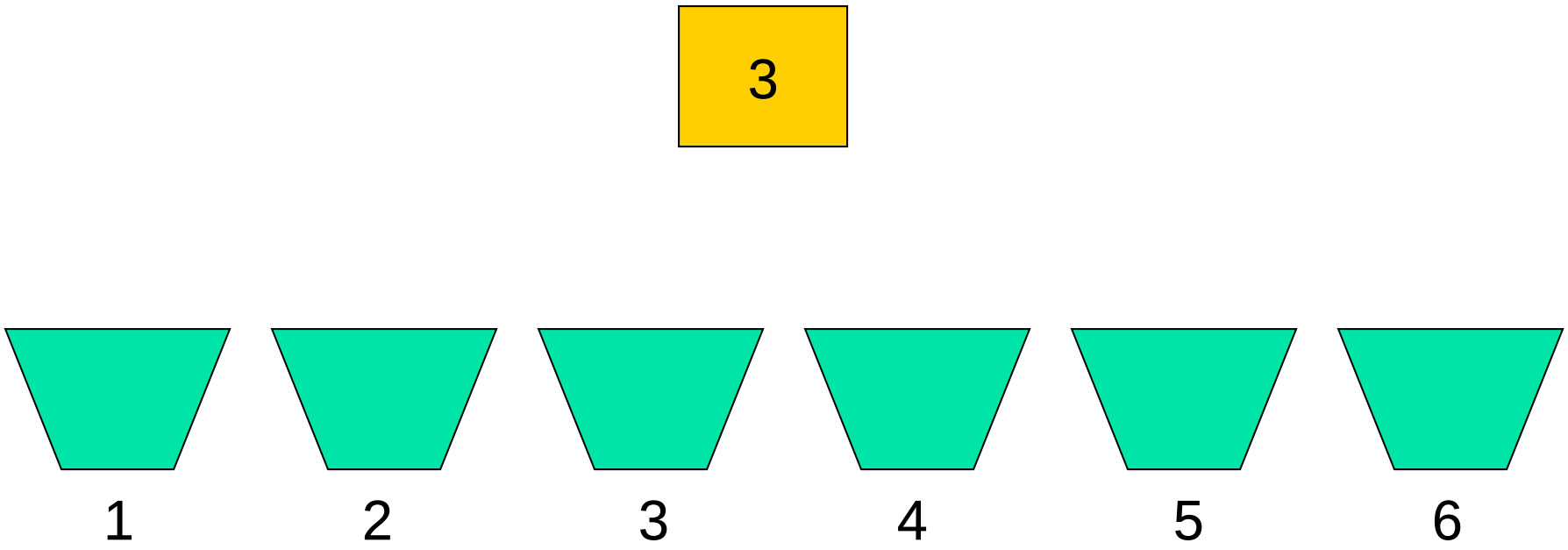
Possible outcomes from rolling a fair 6-sided die



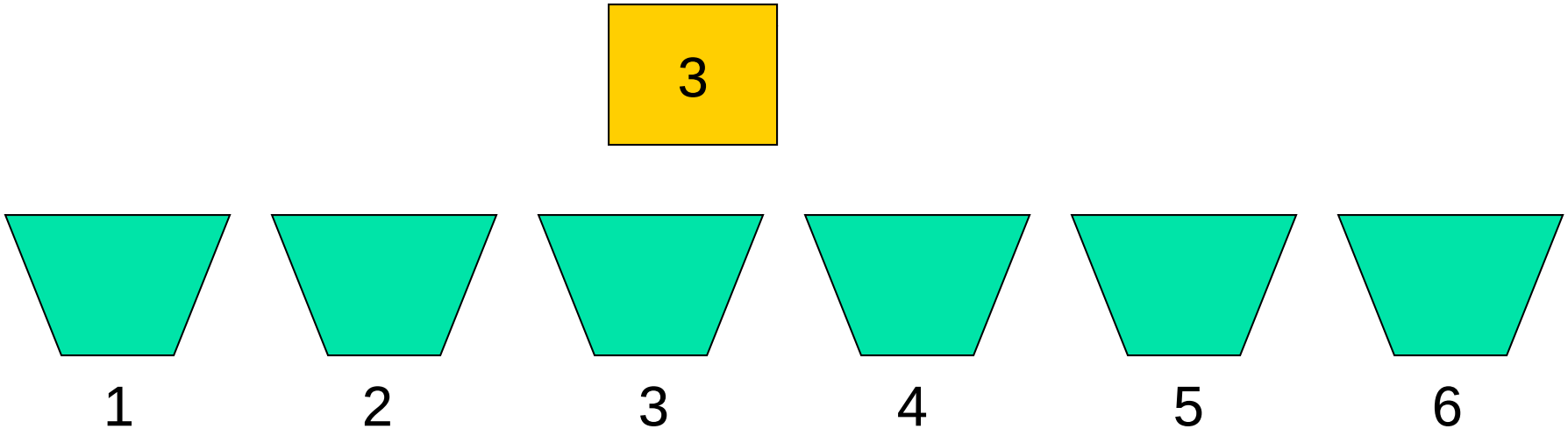
Simulation



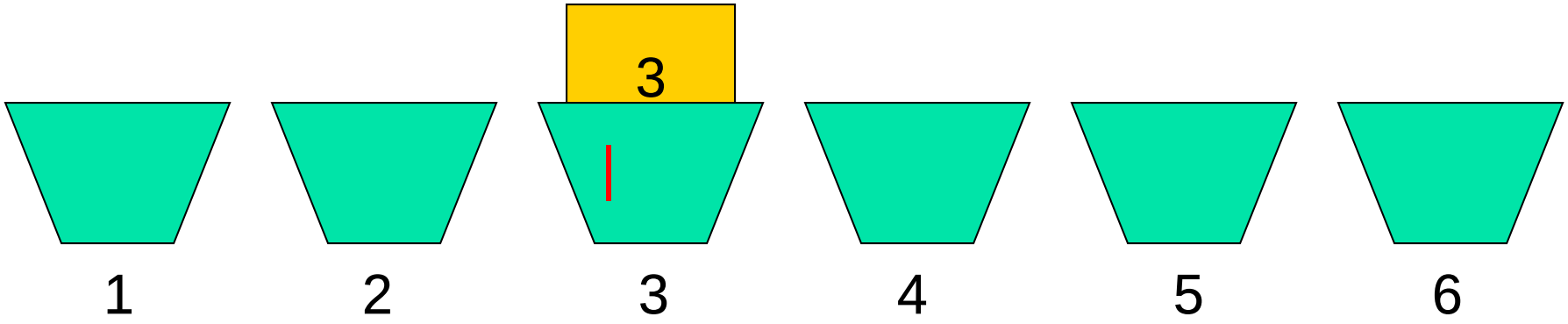
Simulation



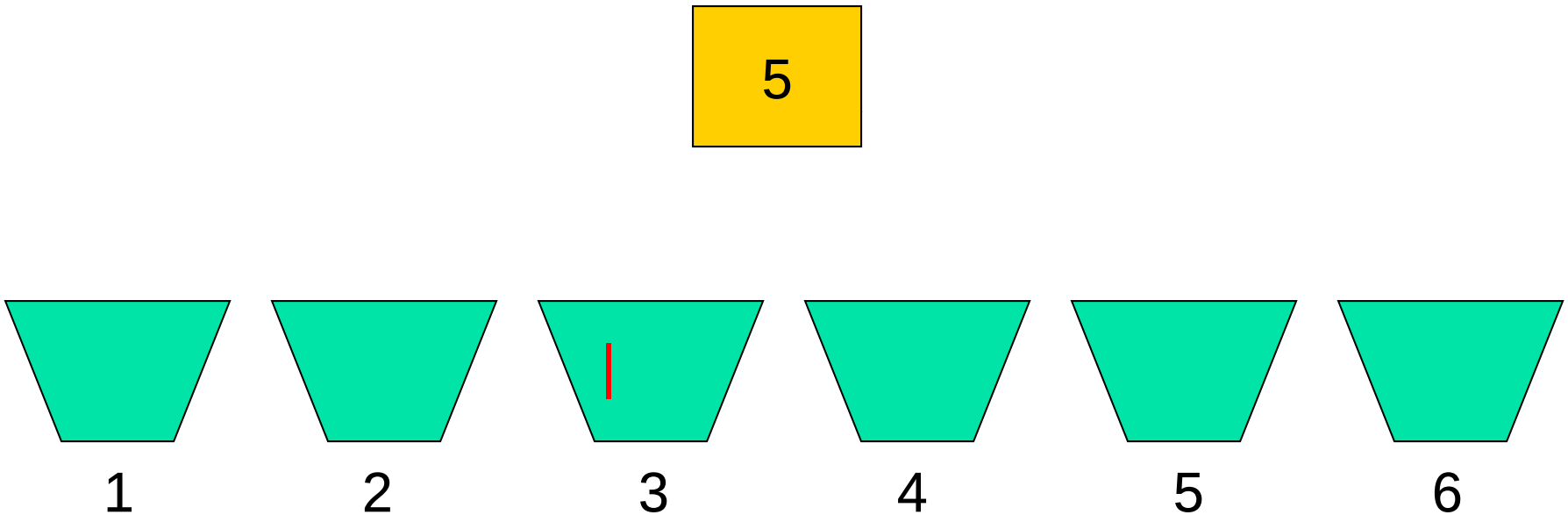
Simulation



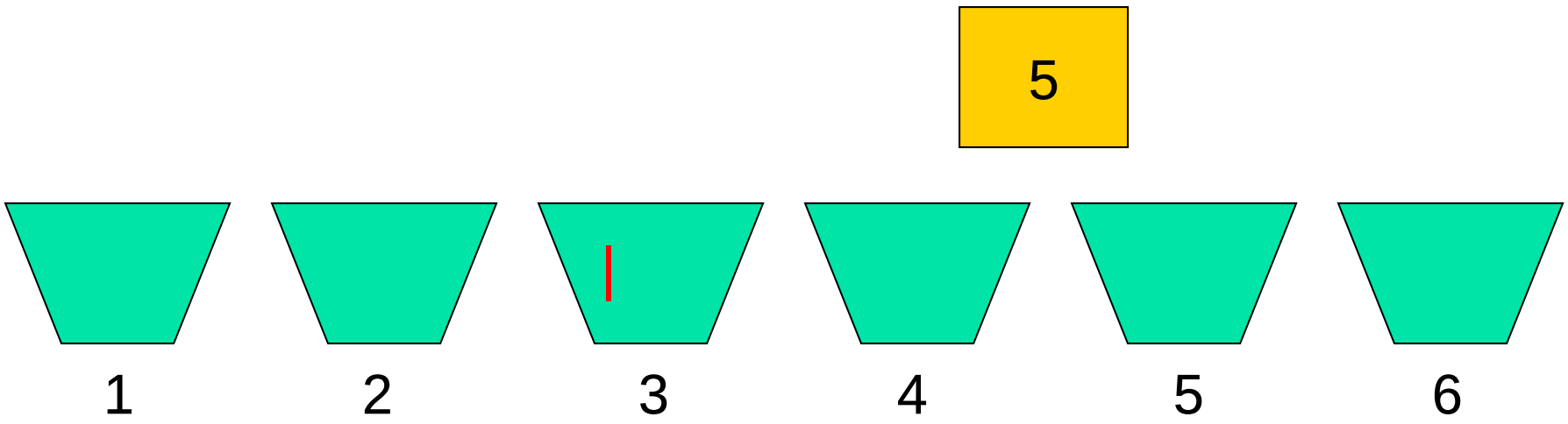
Simulation



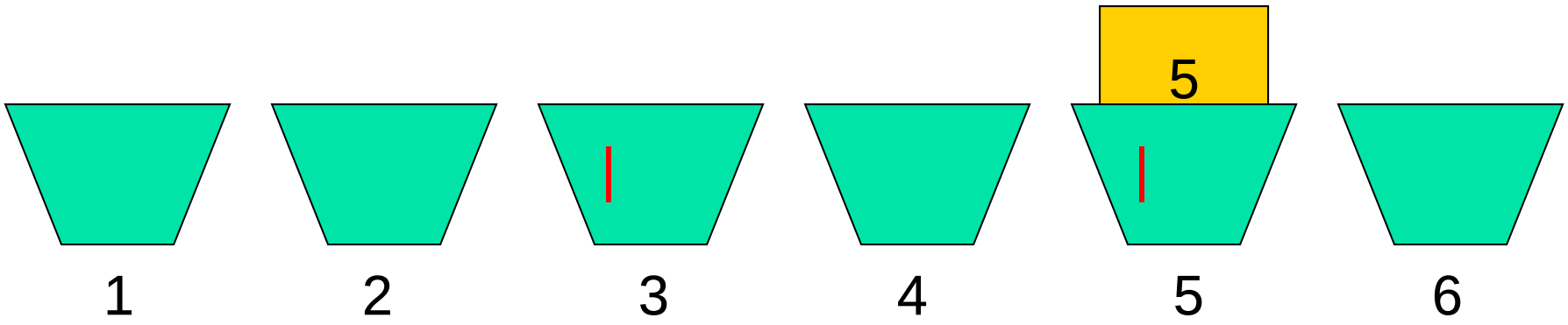
Simulation



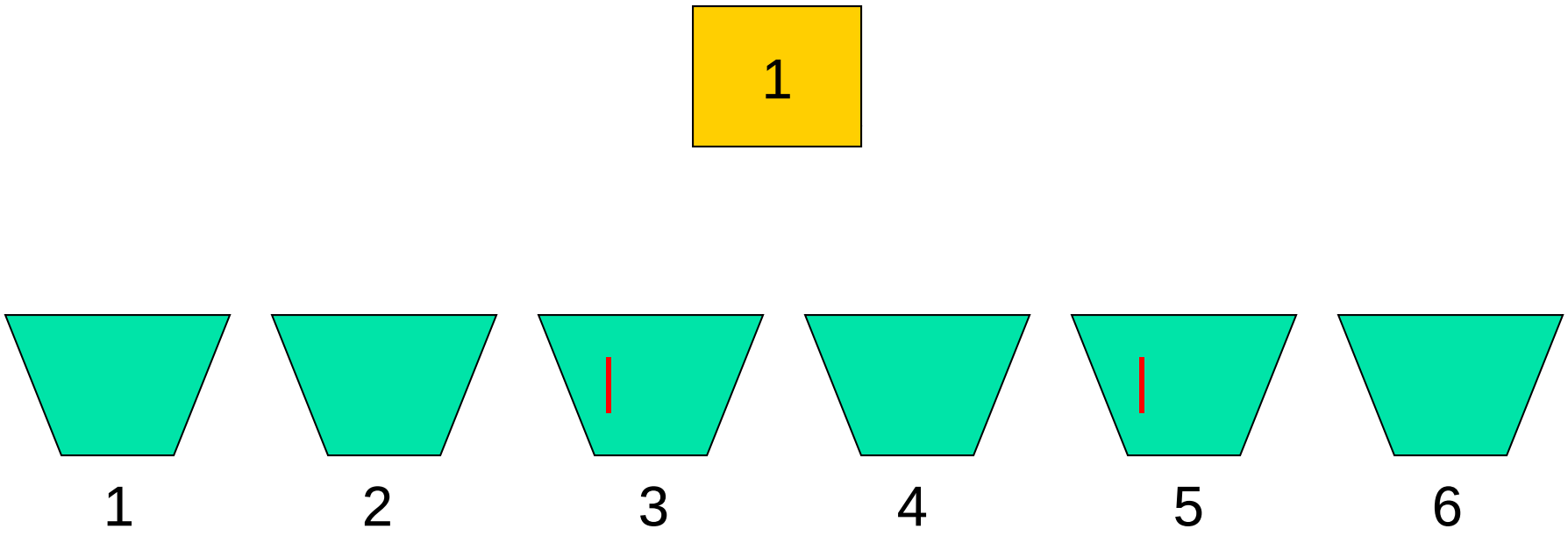
Simulation



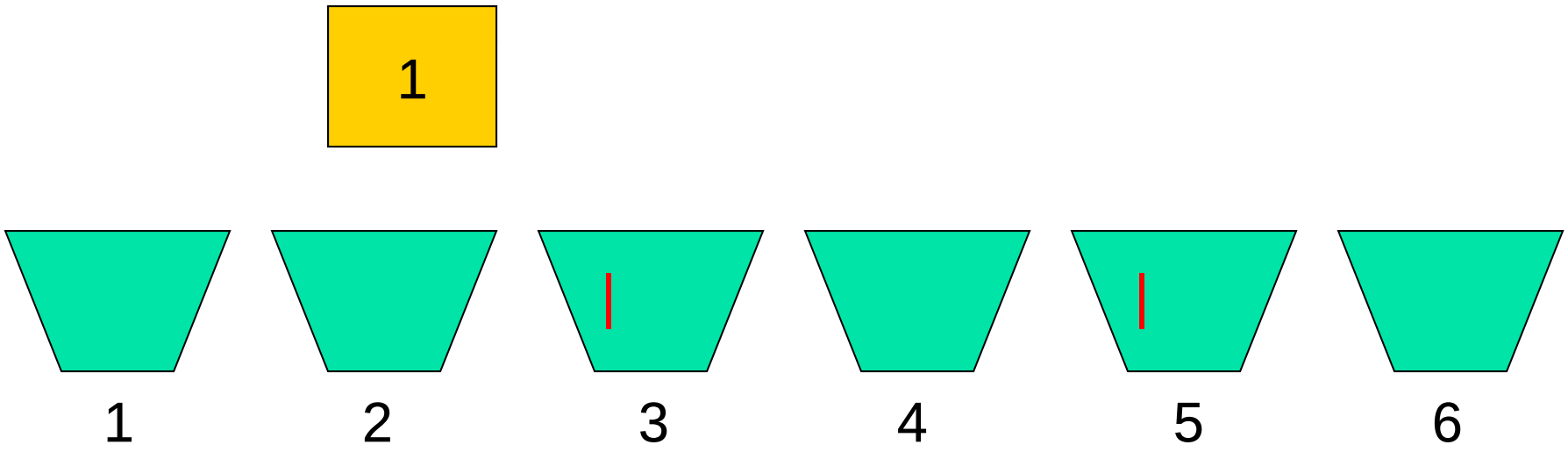
Simulation



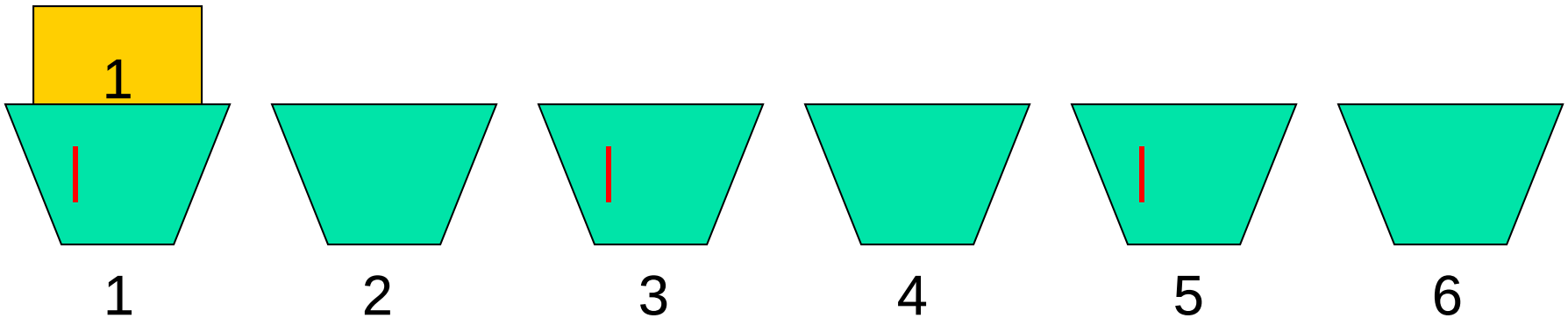
Simulation



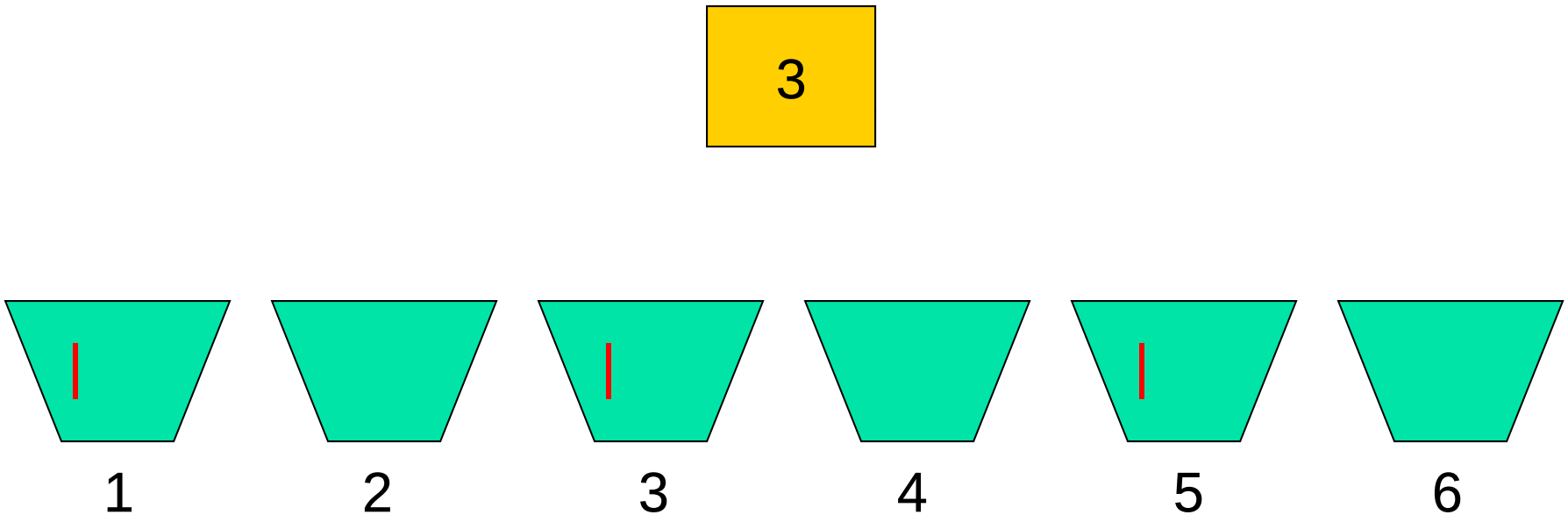
Simulation



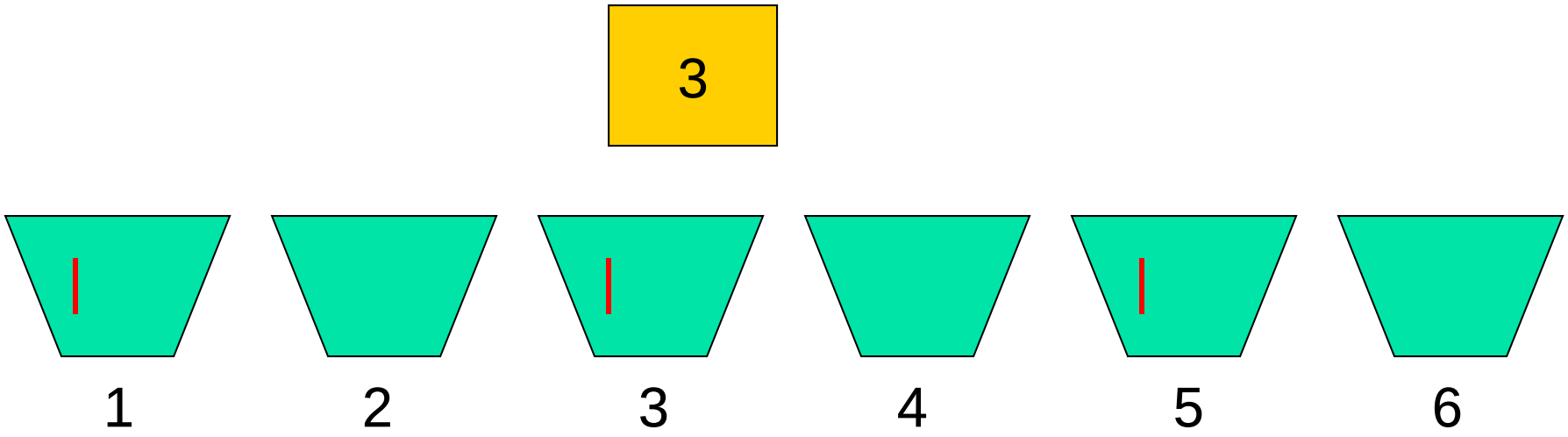
Simulation



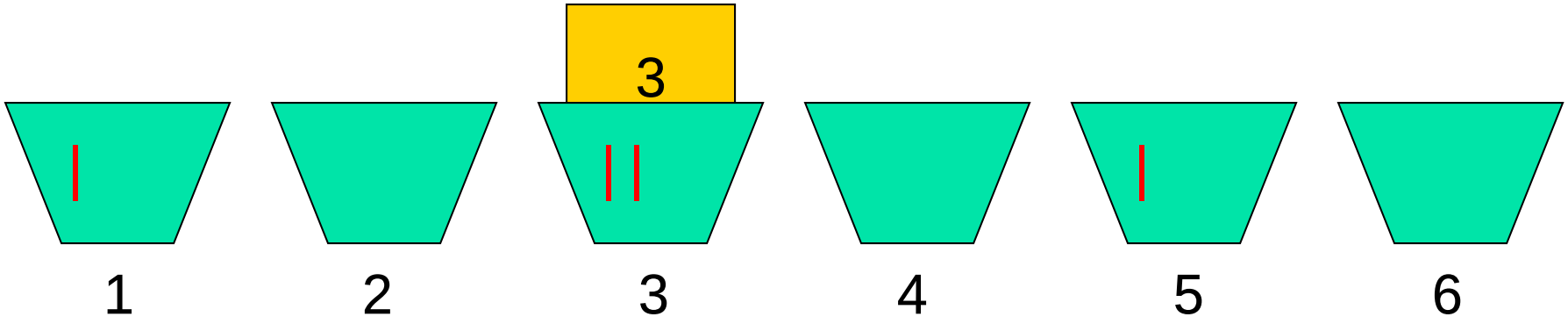
Simulation



Simulation



Simulation



Keep tally on repeated rolls of a fair die

Repeat the following:

```
% roll the die

% increment correct "bin"
```

```
function count = rollDie(rolls)

FACES= 6;                    % #faces on die
count= zeros(1,FACES);  % bins to store counts

% Count outcomes of rolling a FAIR die
for k= 1:rolls
    % Roll the die

    % Increment the appropriate bin

end

% Show histogram of outcome
```

```
function count = rollDie(rolls)

FACES= 6;                    % #faces on die
count= zeros(1,FACES);  % bins to store counts

% Count outcomes of rolling a FAIR die
for k= 1:rolls
    % Roll the die
    face= ceil(rand*FACES);
    % Increment the appropriate bin

end

% Show histogram of outcome
```

```
function count = rollDie(rolls)

FACES= 6;                    % #faces on die
count= zeros(1,FACES);  % bins to store counts

% Count outcomes of rolling a FAIR die
for k= 1:rolls
    % Roll the die
    face= ceil(rand*FACES);
    % Increment the appropriate bin
    count(face)= count(face) + 1;
end

% Show histogram of outcome
```

rollDieV1.m

```
% Count outcomes of rolling a FAIR die
count= zeros(1,6);
for k= 1:100
    face= ceil(rand*6);
    if face==1
        count(1)= count(1) + 1;
    elseif face==2
        count(2)= count(2) + 1;
    ⋮
    elseif face==5
        count(5)= count(5) + 1;
    else
        count(6)= count(6) + 1;
    end
end
```

```
function count = rollDie(rolls)

FACES= 6;                    % #faces on die
count= zeros(1,FACES);  % bins to store counts

% Count outcomes of rolling a FAIR die
for k= 1:rolls
    % Roll the die
    face= ceil(rand*FACES);
    % Increment the appropriate bin
    count(face)= count(face) + 1;
end

% Show histogram of outcome
```