MAKE
SCHOOL

MARKOV CHAINS

Scrambling Russian poetry since 1913

MAKE SCHOOL

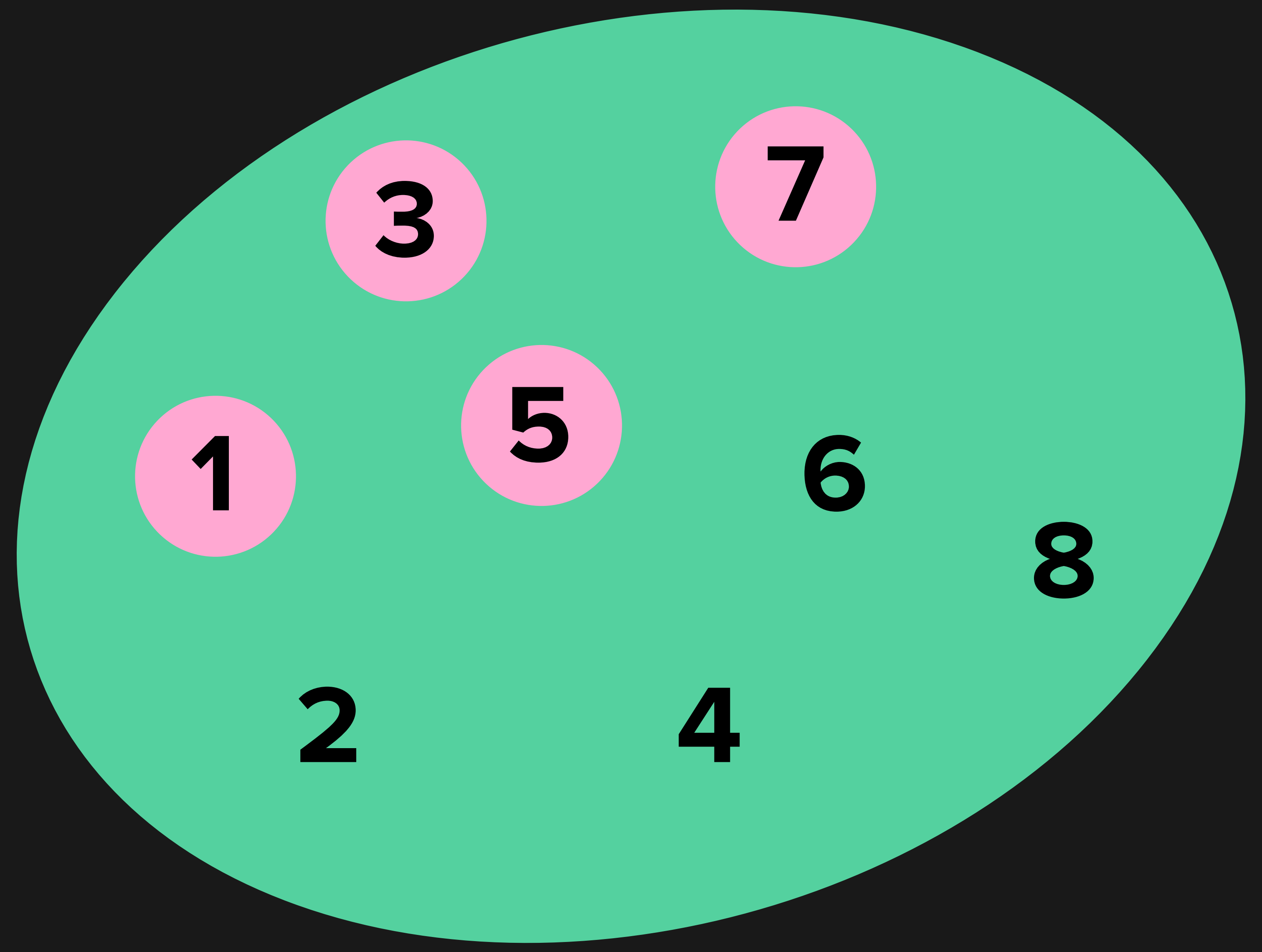
FIRST, MORE PROBABILITY

Probabilities are defined using a *sample space* of possible states of the world

An *event* can occur or not in each world

Example: a die roll comes out odd

The probability of an event is the fraction of worlds in which it occurs



VARIABLES AND DISTRIBUTIONS

We often talk about a *random variable* having a certain *distribution*:

`token` is uniformly distributed over the strings `'see'`, `'spot'`, and `'run'`

A distribution just assigns probabilities to the events of the variable having particular values

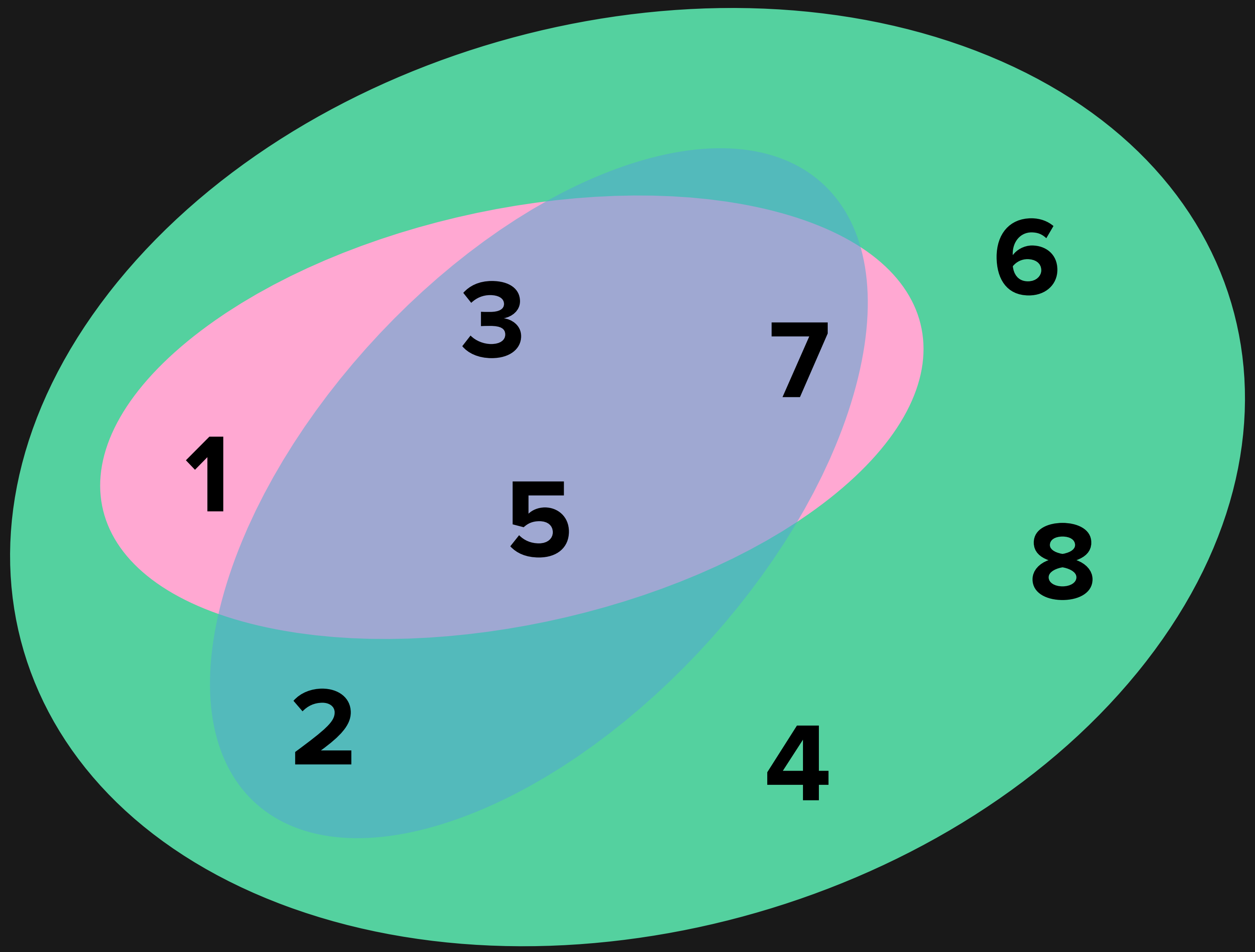
COMBINING EVENTS

Consider the events of rolling odd or prime

These events each have probability ½

What's the probability of rolling odd and prime?

What about rolling odd or prime?



EVENTS AS SETS

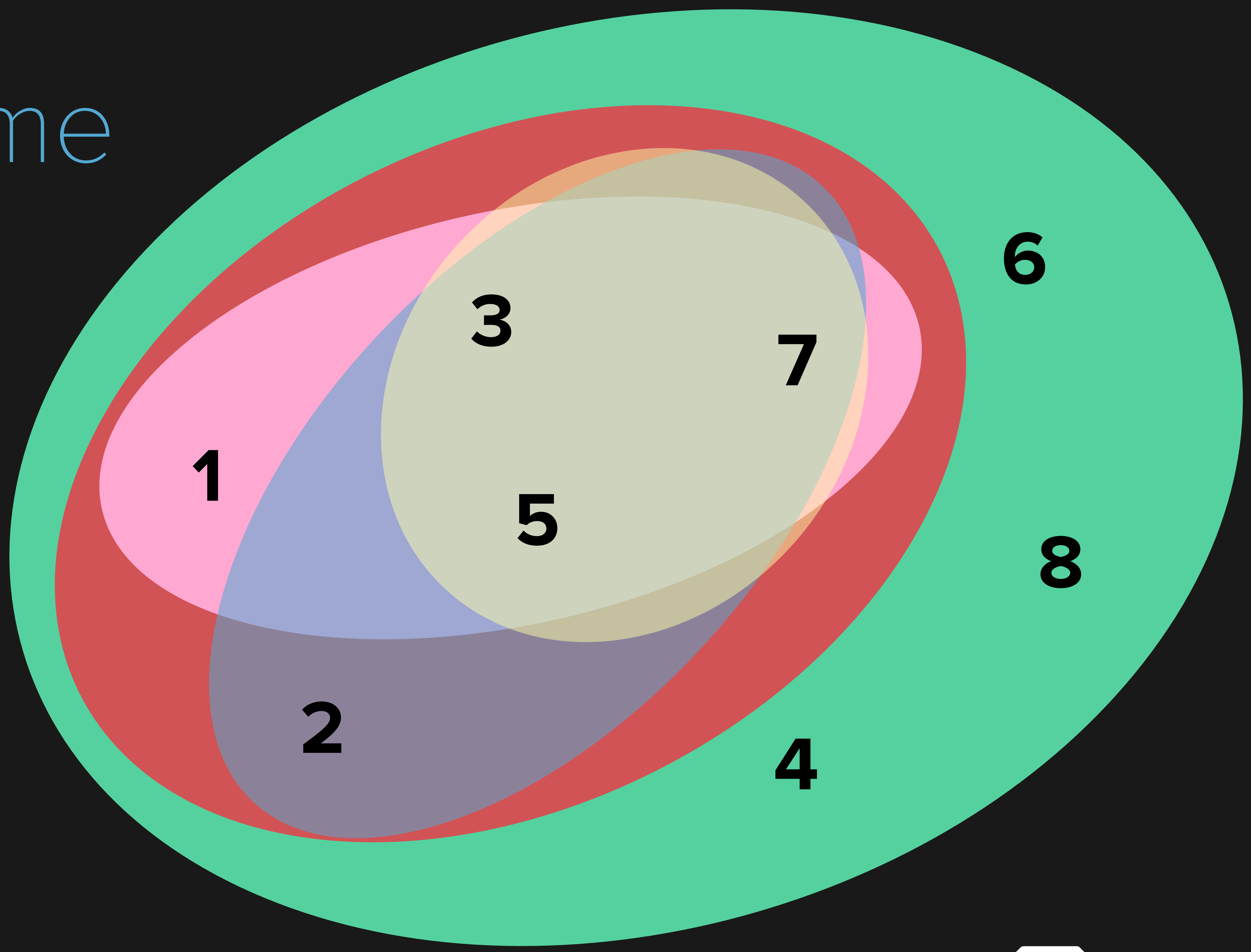
odd and prime = odd ∩ prime

odd or prime = odd ∪ prime

How can we compute

Pr(●) or Pr(●) given

Pr(●) and Pr(●)?



INCLUSION-EXCLUSION

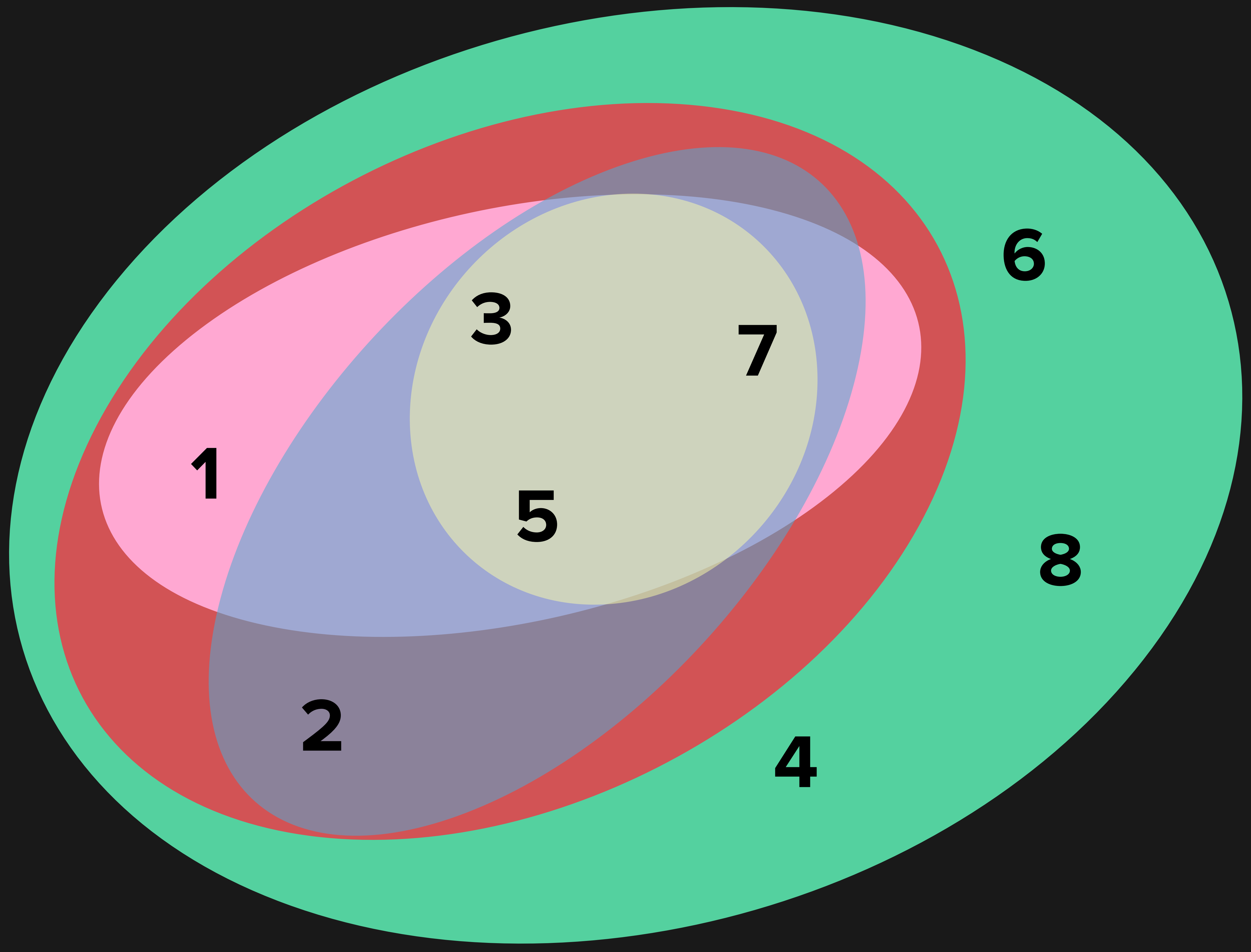
Notice: |●| = |●| + |●| - |●|

5 = 4 + 4 - 3

● gets counted twice

Pr(A or B) =

Pr(A) + Pr(B) - Pr(A and B)



MAKE SCHOOL

JOINT PROBABILITIES

Pr(A) and Pr(B) don't give enough information to determine the *joint probability* Pr(A and B), usually written Pr(A, B)

Pr(odd) = Pr(prime) = Pr(even) = ½, but

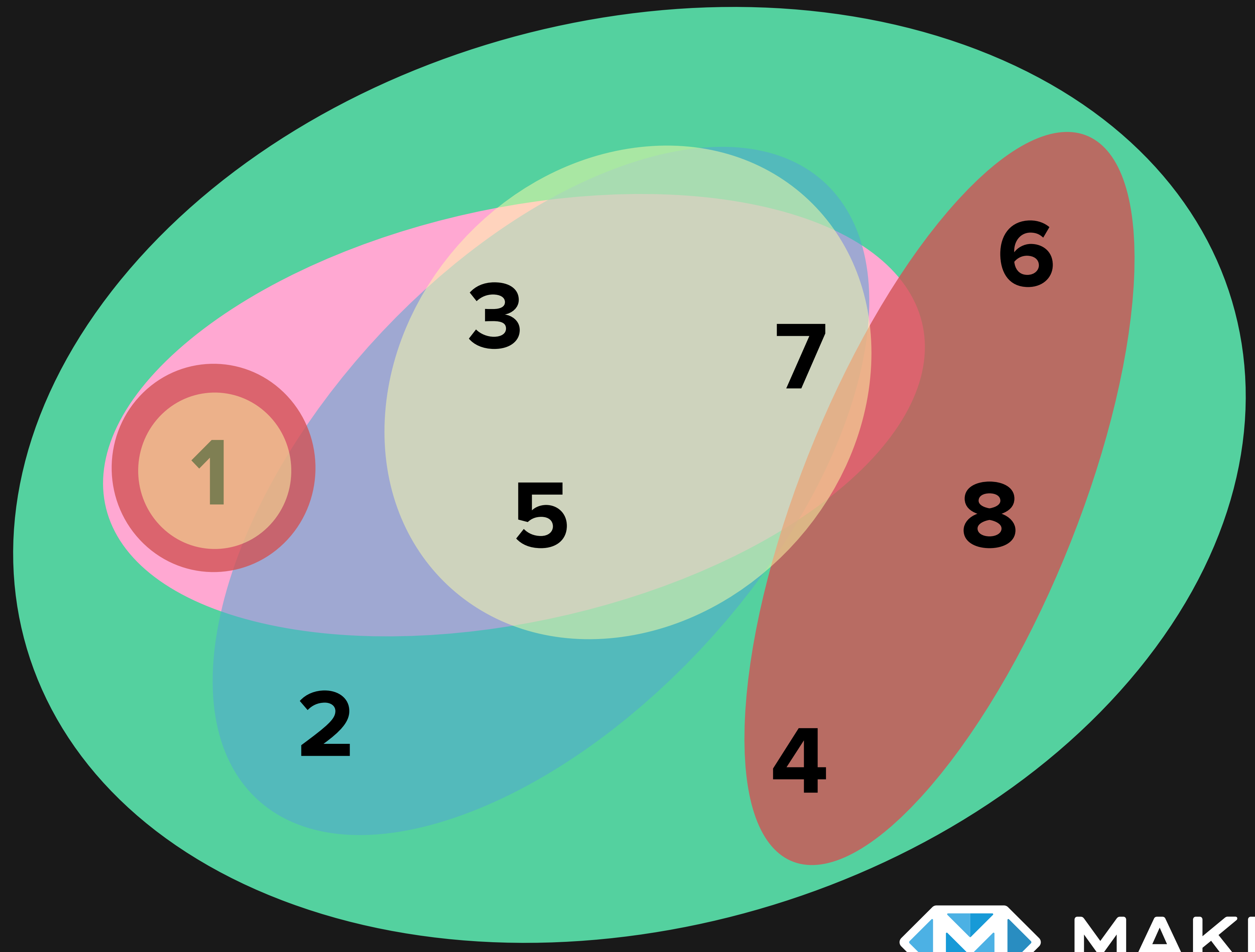
Pr(odd, prime) = ⅜

Pr(odd, even) = 0

MARGINAL PROBABILITIES

We can go the other way, and recover the *marginals* Pr(A) and Pr(B) from the joints:

Pr(odd) = Pr(odd, prime)
+ Pr(odd, not prime)



MAKE SCHOOL

CONDITIONAL PROBABILITIES

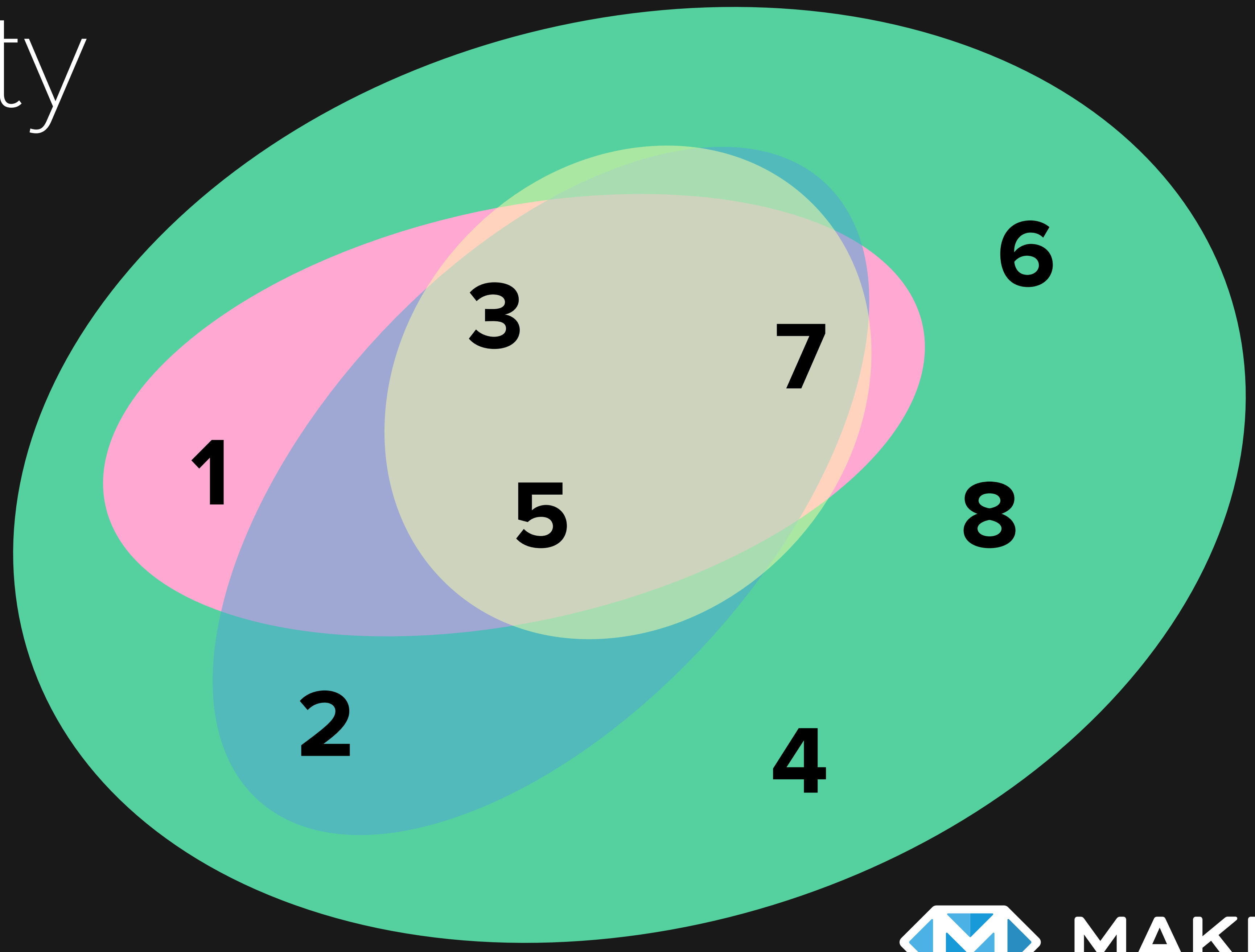
If we know the roll was odd, what is the probability it was prime?

This is the *conditional* probability

Pr(odd | prime)

Pr(A | B) = Pr(A, B) / Pr(B)

¾ = ⅜ / ½



EXERCISE

A = roll is a power of 2

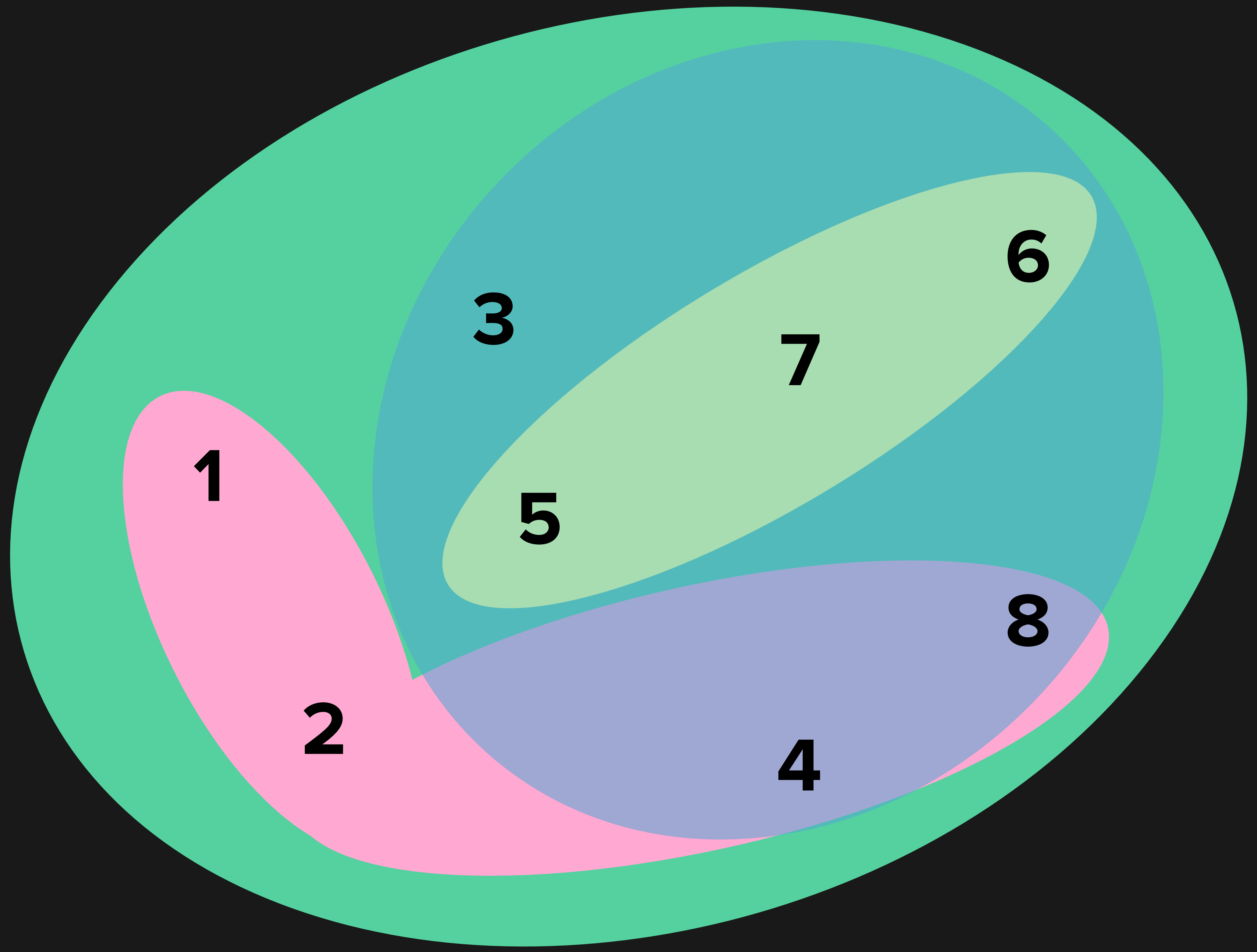
$B = (\text{roll} > 2)$

$C = (5 \leq \text{roll} \leq 7)$

Compute:

$\Pr(A, B)$, $\Pr(B \mid C)$,

$\Pr(C \mid B)$, $\Pr(C \mid B, A)$



$$\Pr(A \mid B) = \Pr(A, B) \,/\, \Pr(B)$$

INDEPENDENCE

If A and B don't influence each other, then

$$\Pr(A, B) = \Pr(A \mid B)\,\Pr(B) = \Pr(A)\,\Pr(B)$$

When their joint probability factors like this, A and B are said to be *independent*

Example: $\Pr(n \text{ coin flips all being heads}) = (\tfrac{1}{2})^n$

SAMPLING A DISTRIBUTION

Given a list of tokens and their probabilities, how can we *sample* from that distribution?

‘the’: 1/2, ‘best’: 1/8, ‘times’: 1/4, ‘worst’: 1/8

A SIMPLE APPROACH

```
tokens = [‘the’, ‘the’, ‘the’, ‘the’, ‘best’, ‘times’,
‘times’, ‘worst’]

def uniformSample(items):
  index = random.randint(0, len(items) - 1)
  return items[index]

sample = uniformSample(tokens)
```

Can you verify that the event `sample` = `‘the’` has probability 1/3?

ANOTHER ATTEMPT

```
types = [‘the’, ‘best’, ‘times’, ‘worst’]
probs = [0.5, 0.125, 0.25, 0.125]

def weightedSample(items, probs):
  while True:
    for (index, prob) in enumerate(probs):
      if random.random() <= prob:
        return items[index]

sample = weightedSample(types, probs)
```

Does this work?

WHAT GOES WRONG

```python
def weightedSample(items, probs):
  while True:
    for (index, prob) in enumerate(probs):
      if random.random() <= prob:
        return items[index]
```

Let's look at a simple example: A and B both with probability ½

$$\Pr(A \text{ returned}) = \tfrac{1}{2} + (1 - \tfrac{1}{2})\,(1 - \tfrac{1}{2})\,\tfrac{1}{2} + (1 - \tfrac{1}{2})^4\,\tfrac{1}{2} + \ldots$$

returns 1st A

fails 1st A

fails 1st B

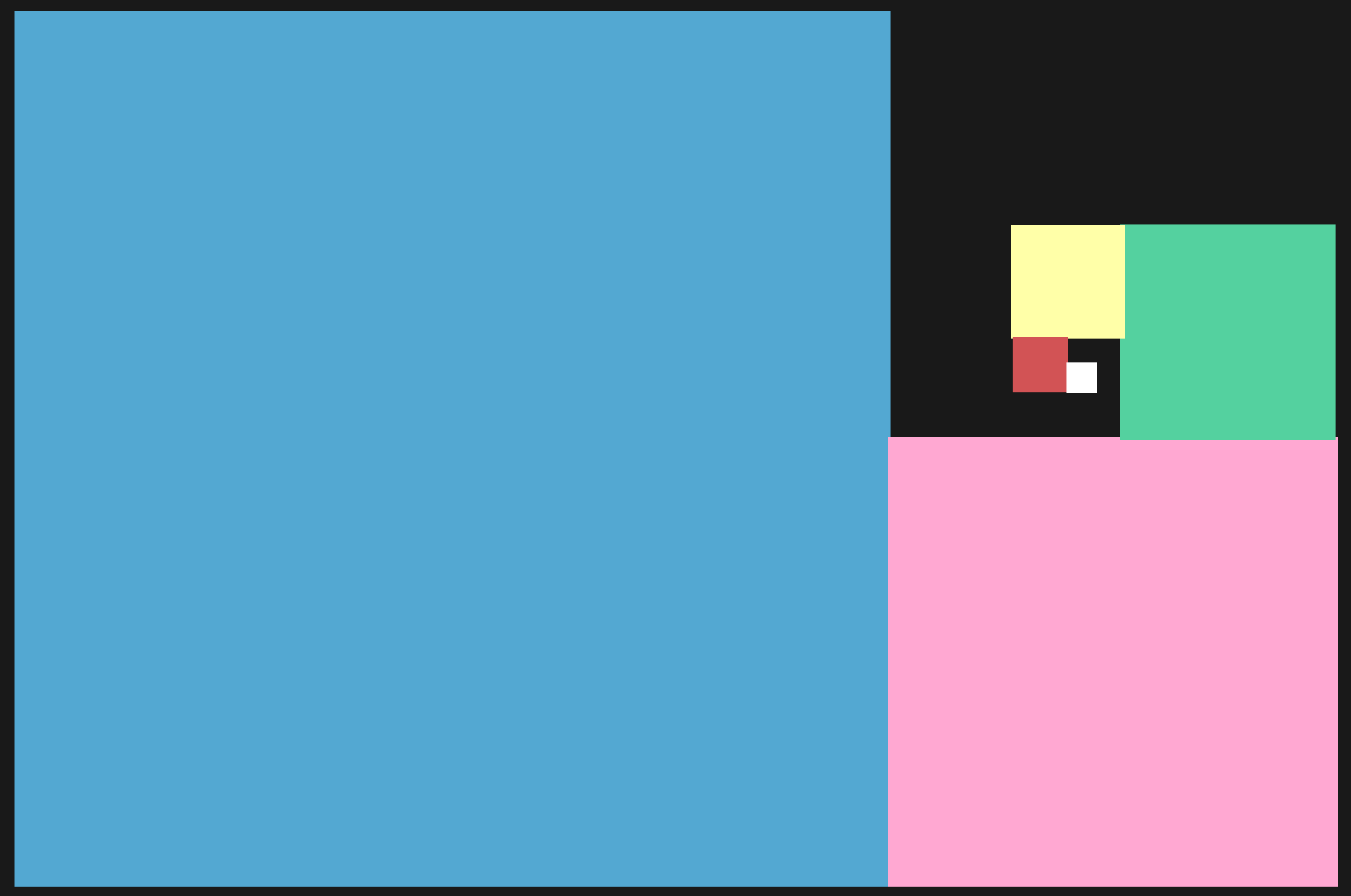
returns 2nd A

SUMMING THE SERIES

$$\Pr(A \text{ returned}) = \tfrac{1}{2} + \tfrac{1}{4} \times \tfrac{1}{2} + \tfrac{1}{4}^2 \times \tfrac{1}{2} + \ldots$$

$$= \tfrac{1}{2} \left(1 + \tfrac{1}{4} + \tfrac{1}{4}^2 + \ldots \right)$$

$$= \tfrac{1}{2} / (1 - \tfrac{1}{4}) = \tfrac{2}{3}$$



MAKE SCHOOL

ONE LAST TRY

```
types = [‘the’, ‘best’, ‘times’, ‘worst’]
cumulativeProbs = [0.5, 0.625, 0.875, 1.0]

def weightedSample(items, cprobs):
  dart = random.random()
  for (index, cprob) in enumerate(cprobs):
    if dart <= cprob:
      return items[index]

sample = weightedSample(types, cumulativeProbs)
```

Why does this work?

A TALE OF ONE DISTRIBUTION

So far, we've learned a distribution on tokens by counting how many times they occur

Thus we don't generate rare words too often

If we want *N* words, we sample *N* times from that one distribution, so the most likely pair of words to generate is 'the the'

CONTEXT TO THE RESCUE

This is a problem about *context*: ‘the’ is common, but it’s very rare after another ‘the’

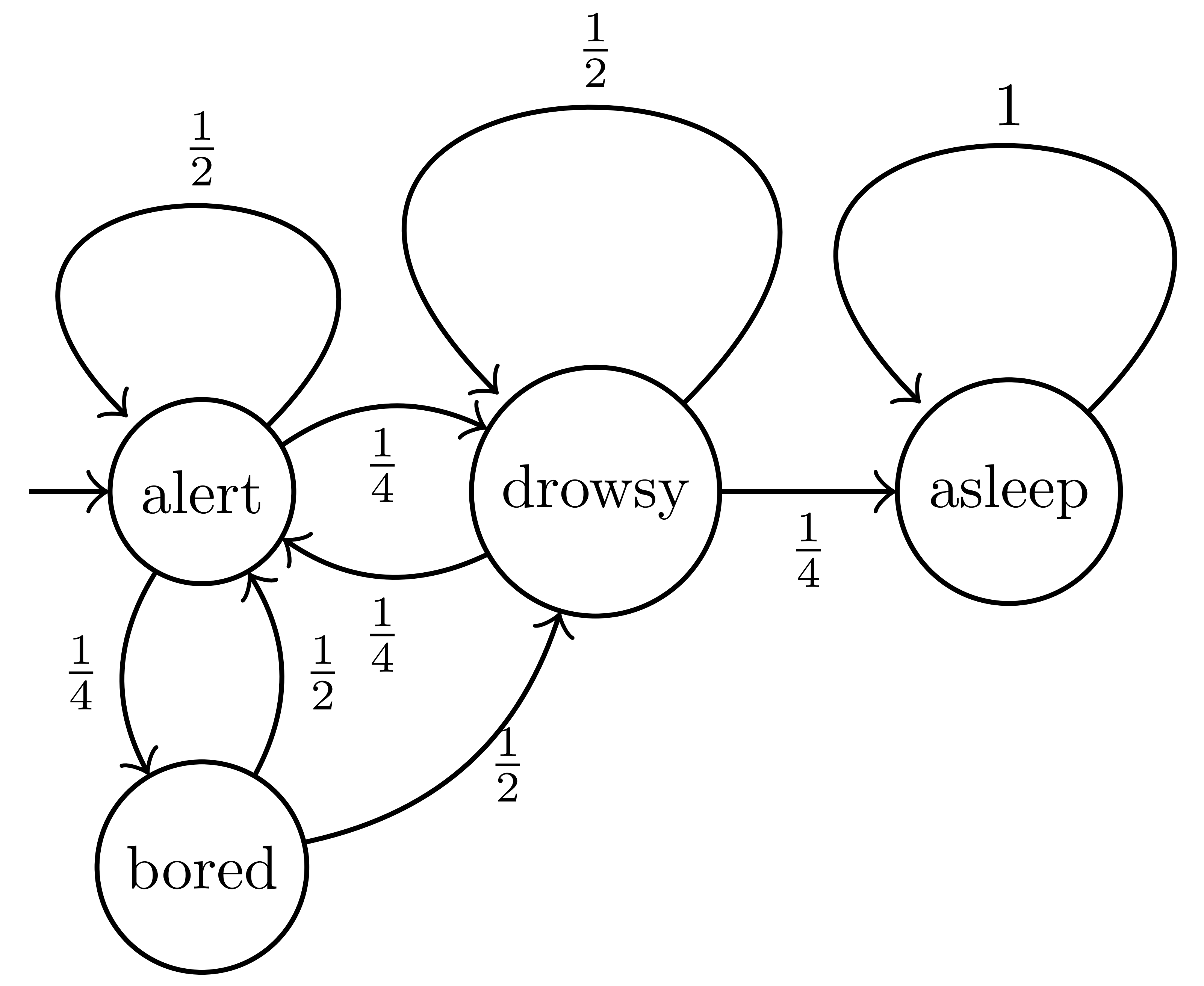
Another example: ‘Zappa’ is rare, but much more likely after ‘Dweezil’

We can model this by using a different distribution depending on what the last generated token was

MAKE SCHOOL

MARKOV CHAINS

A *Markov chain* consists of *states* linked by *transitions* labeled with probabilities



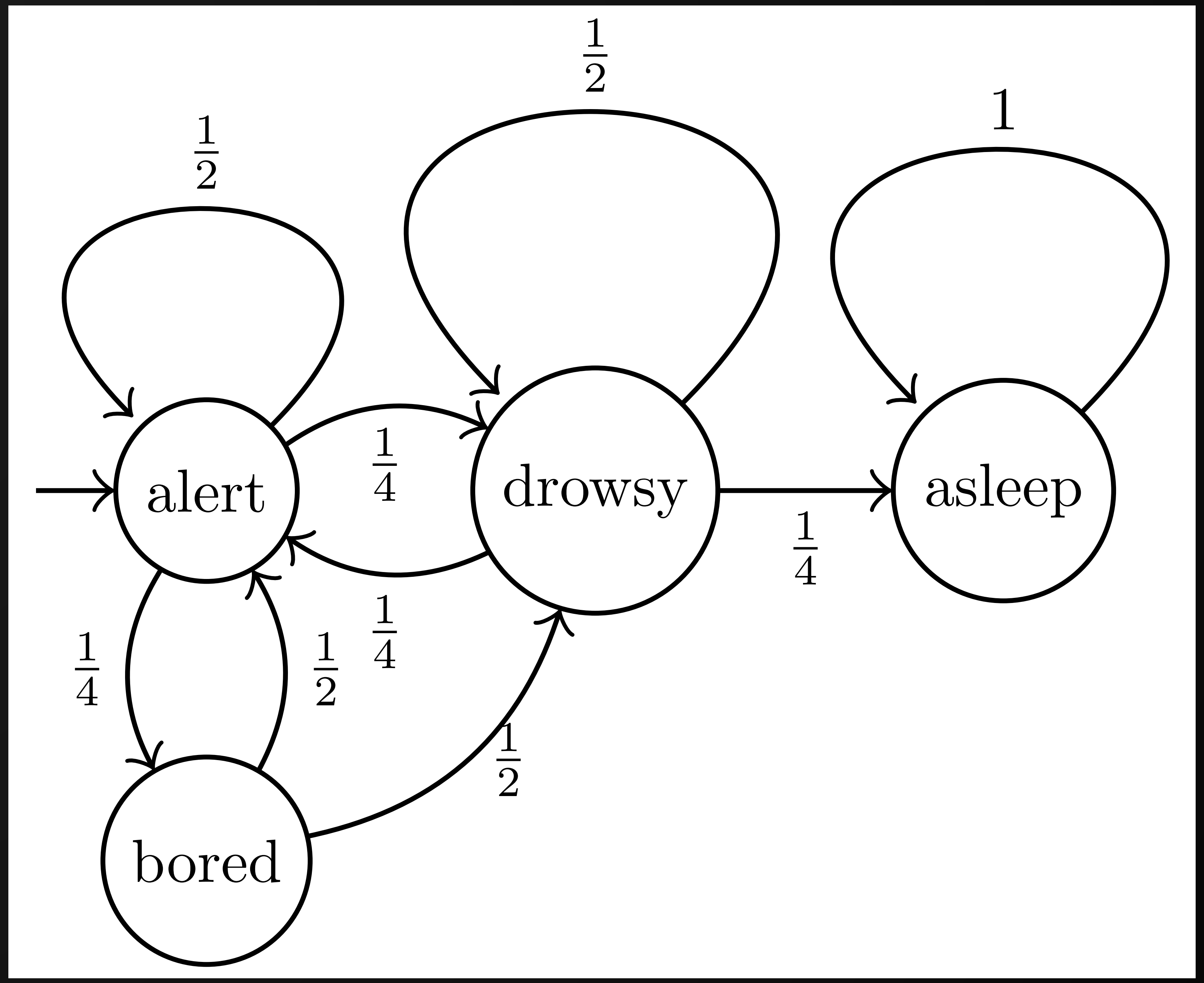
RANDOM WALKS

A Markov chain defines a *stochastic process* for generating sequences of states via a *random walk*

Starting from a state, we repeatedly pick transitions according to their probabilities

Example:

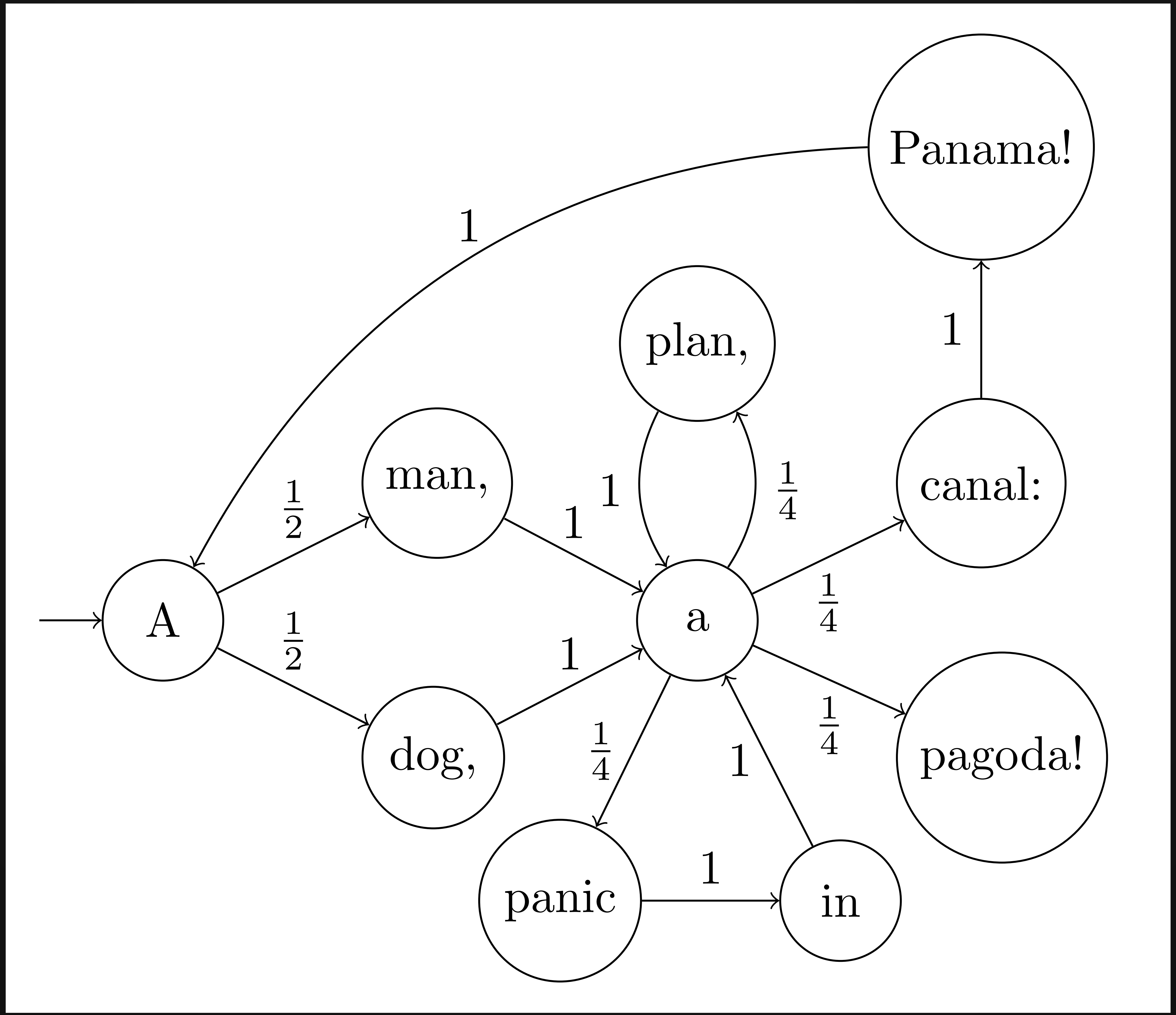
alert-drowsy-drowsy-asleep



MAKE SCHOOL

LEARNING A MARKOV CHAIN

"A man, a plan, a canal: Panama! A dog, a panic in a pagoda!"



LEARNING TRANSITIONS

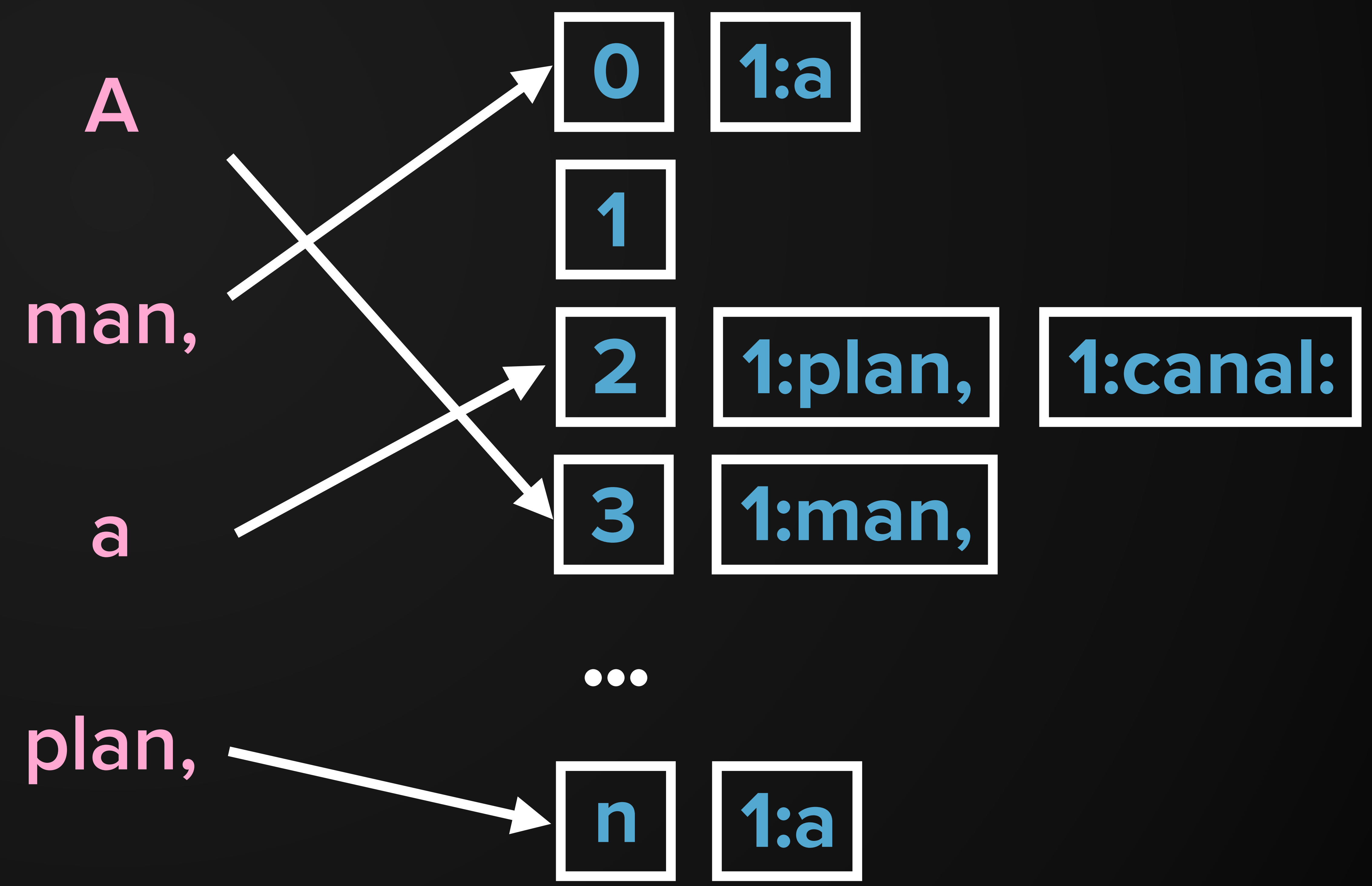
Like when counting tokens, we only want to make one pass through the corpus

How can we efficiently build the transitions leaving each state?

Use a hash table

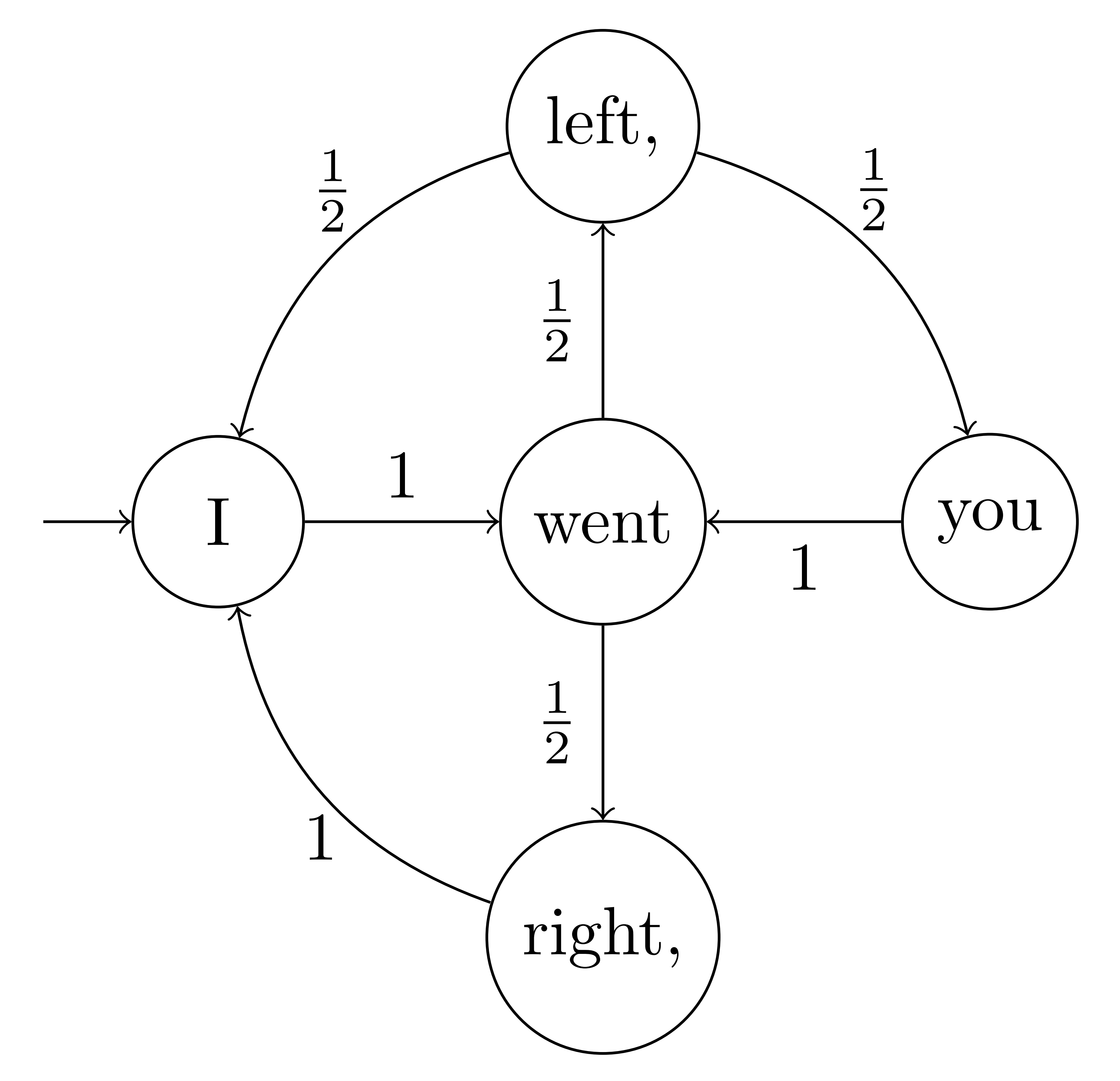
LEARNING TRANSITIONS

"A man, a plan, a canal: Panama! A dog, a panic in a pagoda!"



MORE CONTEXT?

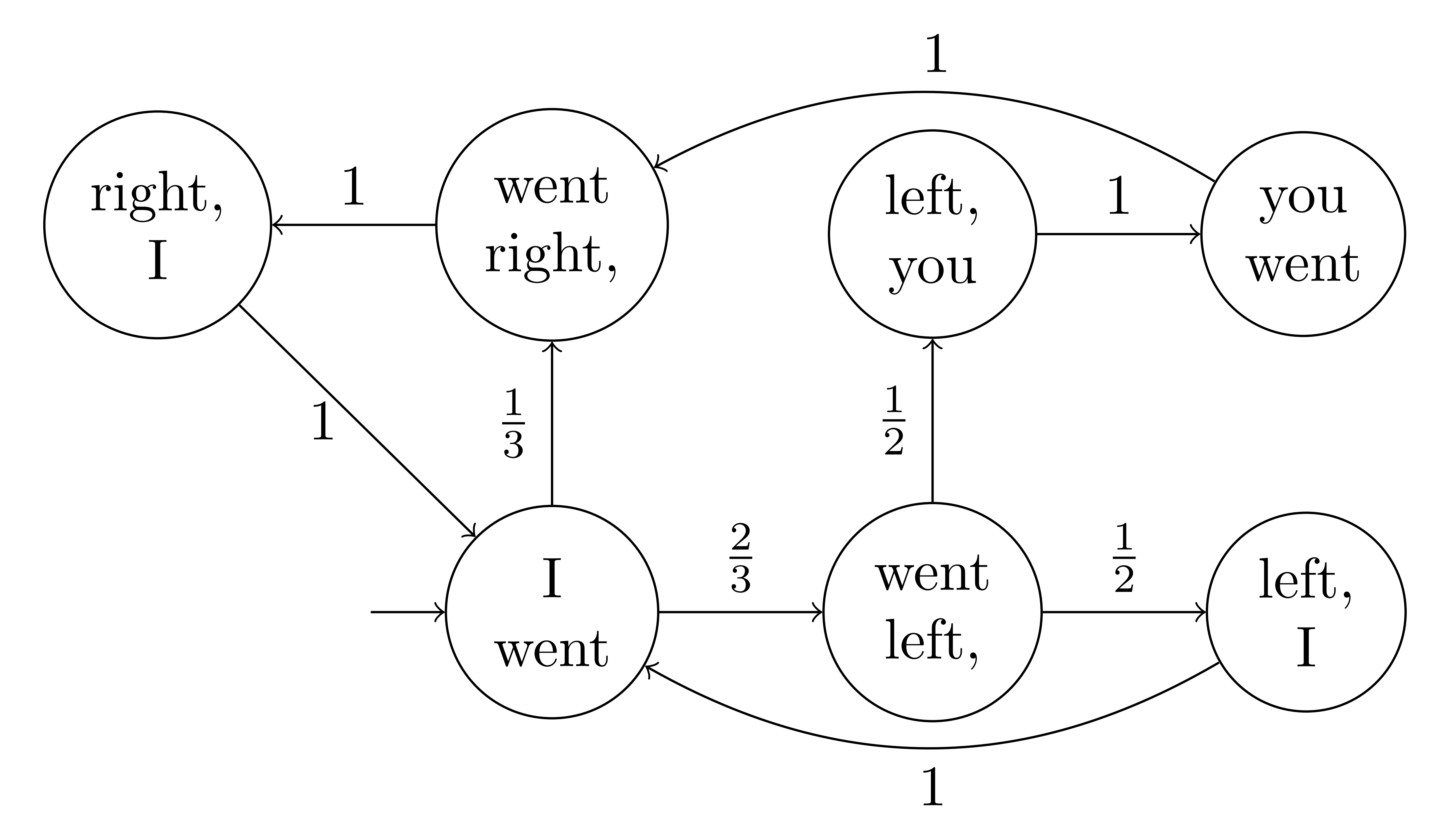
"I went left, you went right, I went left, I went right,"



MAKE SCHOOL

MORE CONTEXT!

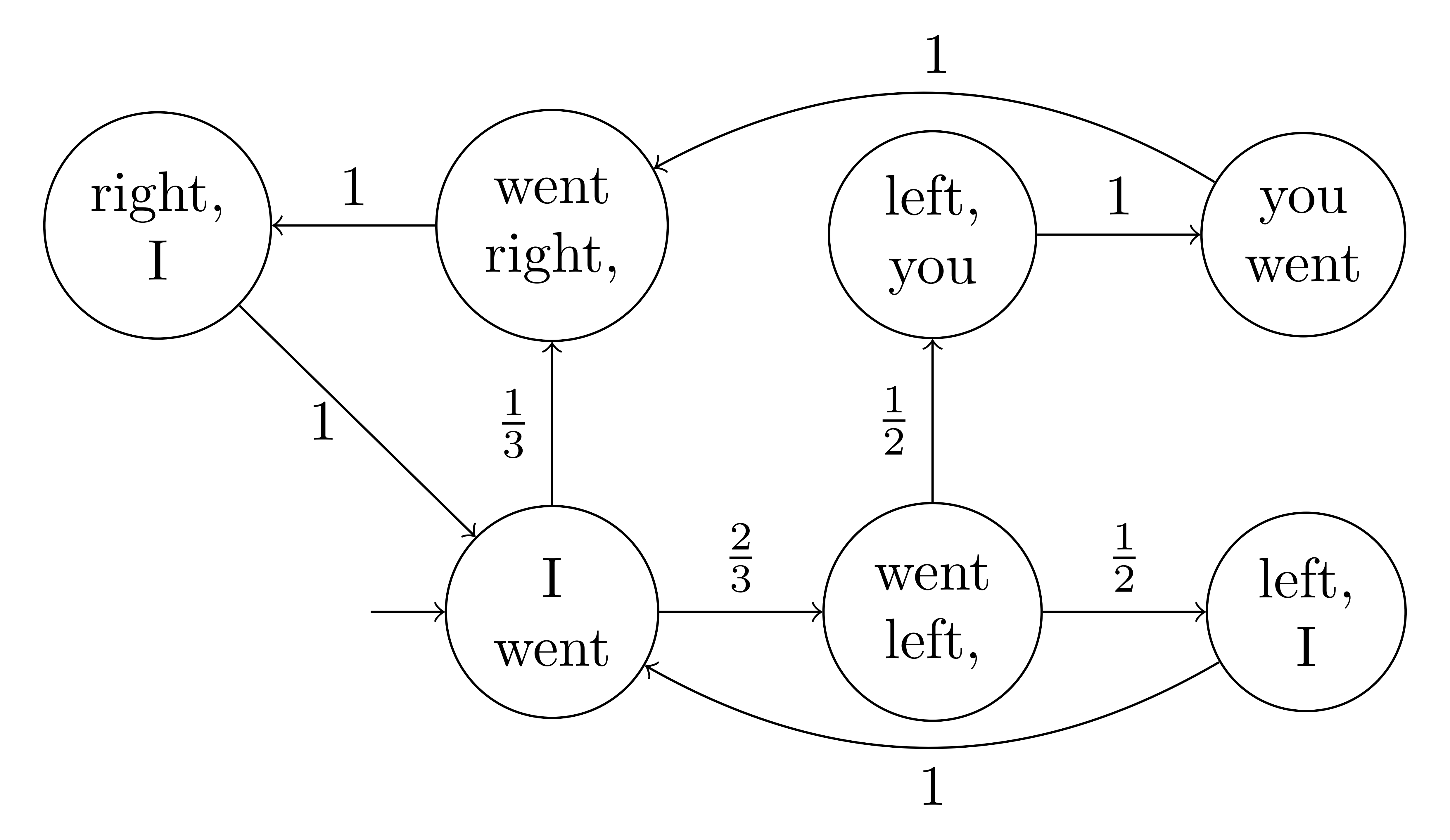
A *second-order Markov chain* has transitions depending on the *two* previous states



MAKE SCHOOL

TRANSITION PROBABILITIES

"I went left, you went right, I went left, I went right,"



EVEN MORE CONTEXT

An *nth-order Markov chain* has transitions depending on the *n* previous states

The probability of moving to a state is a function of the last *n-gram*

Higher-order chains model English better, but can you think of some downsides?

LEARNING TRANSITIONS

Now we have to keep track of the previous *n* tokens, not just one

Since we have the corpus in an array, we can just back up and re-read them

Another approach is to use a *queue*

QUEUES

A *queue* (FIFO buffer) is like an actual line:

Typical operations:

- **enqueue** an item: add it at the back
- **dequeue** the item at the front: remove it
- **iterate** over the queue from front to back

FURTHER DIRECTIONS

Many, many applications. Google's original PageRank algorithm models user behavior with a Markov chain

Endless queue variations and extensions: circular buffers, deques, priority queues (try the heap exercises)