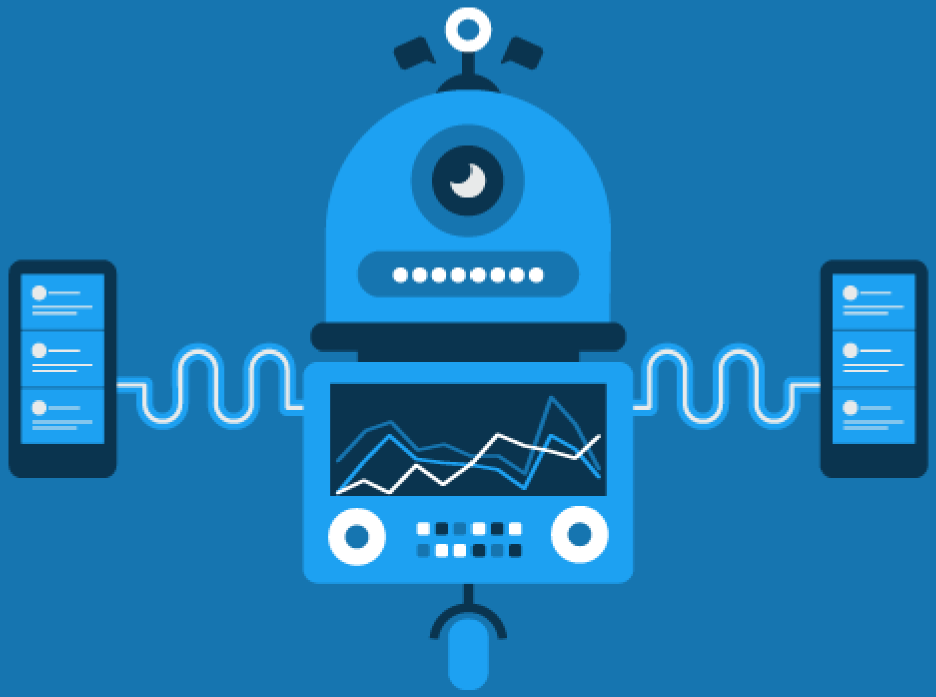
Algorithm Speed

Efficiency and Big Oh notation



Algorithm

- A series of steps to complete a task
- Eg: IKEA assembly instructions, computer program, flowchart, recipe to bake a cake
- Cornerstone of computer science; a break-though in an algorithm often means a radical change in the industry

PageRank: Took search results and ordered them

Google worth $632 billion in 2020

CHRONOLOGICAL LISTING OF A.M. TURING AWARD WINNERS

* person is deceased

(2019)
Catmull, Edwin E.
Hanrahan, Patrick M.

(2018)
Bengio, Yoshua
Hinton, Geoffrey E
LeCun, Yann

(2017)
Hennessy, John L
Patterson, David

(2016)
Berners-Lee, Tim

(2015)
Diffie, Whitfield
Hellman, Martin

(2014)
Stonebraker, Michael

(2000)
Yao, Andrew Chi-Chih

(1999)
Brooks, Frederick ("Fred")

(1998)
Gray, James ("Jim") Nicholas *

(1997)
Engelbart, Douglas *

(1996)
Pnueli, Amir *

(1995)
Blum, Manuel

(1994)
Feigenbaum, Edward A ("Ed")
Reddy, Dabbala Rajagopal ("Raj")

(1993)

(1981)
Codd, Edgar F. ("Ted") *

(1980)
Hoare, C. Antony ("Tony") R.

(1979)
Iverson, Kenneth E. ("Ken") *

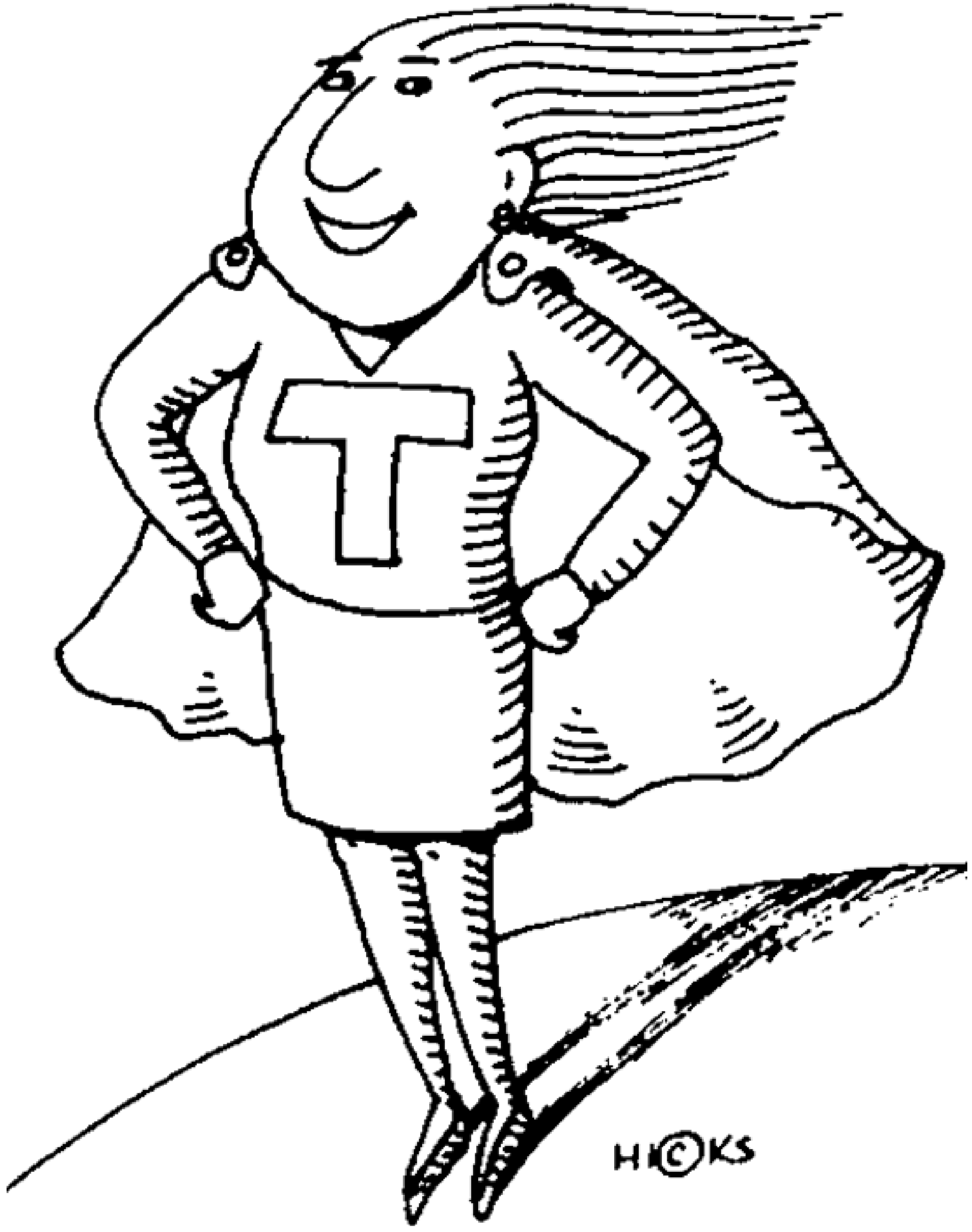
(1978)
Floyd, Robert (Bob) W *

(1977)
Backus, John *

(1976)
Rabin, Michael O.
Scott, Dana Stewart

(1975)
Newell, Allen *
Simon, Herbert ("Herb") Alexander *

Problem:
The teacher needs to hand out a set of assignments, one to each student.



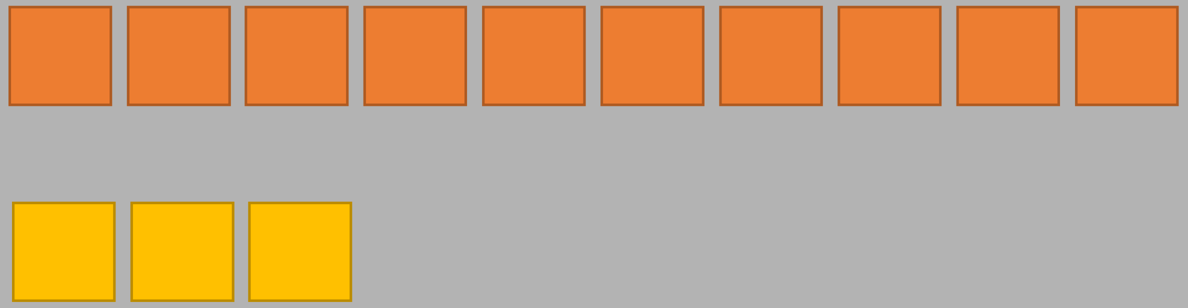
How will we hand out the papers?

One of the big considerations is the time it will take to complete.

That is related to to the efficiency of the algorithm.

Because time
(seconds or nanoseconds)
is hardware dependant,
we measure an algorithm in the
number of operations it takes.

The number of operations depends on the size of the data set.



In this case, the "data set" is the class size.

Thus, we will measure it in terms of n, which will be the class size.

Later this lesson, n will be the array size.

Algorithm

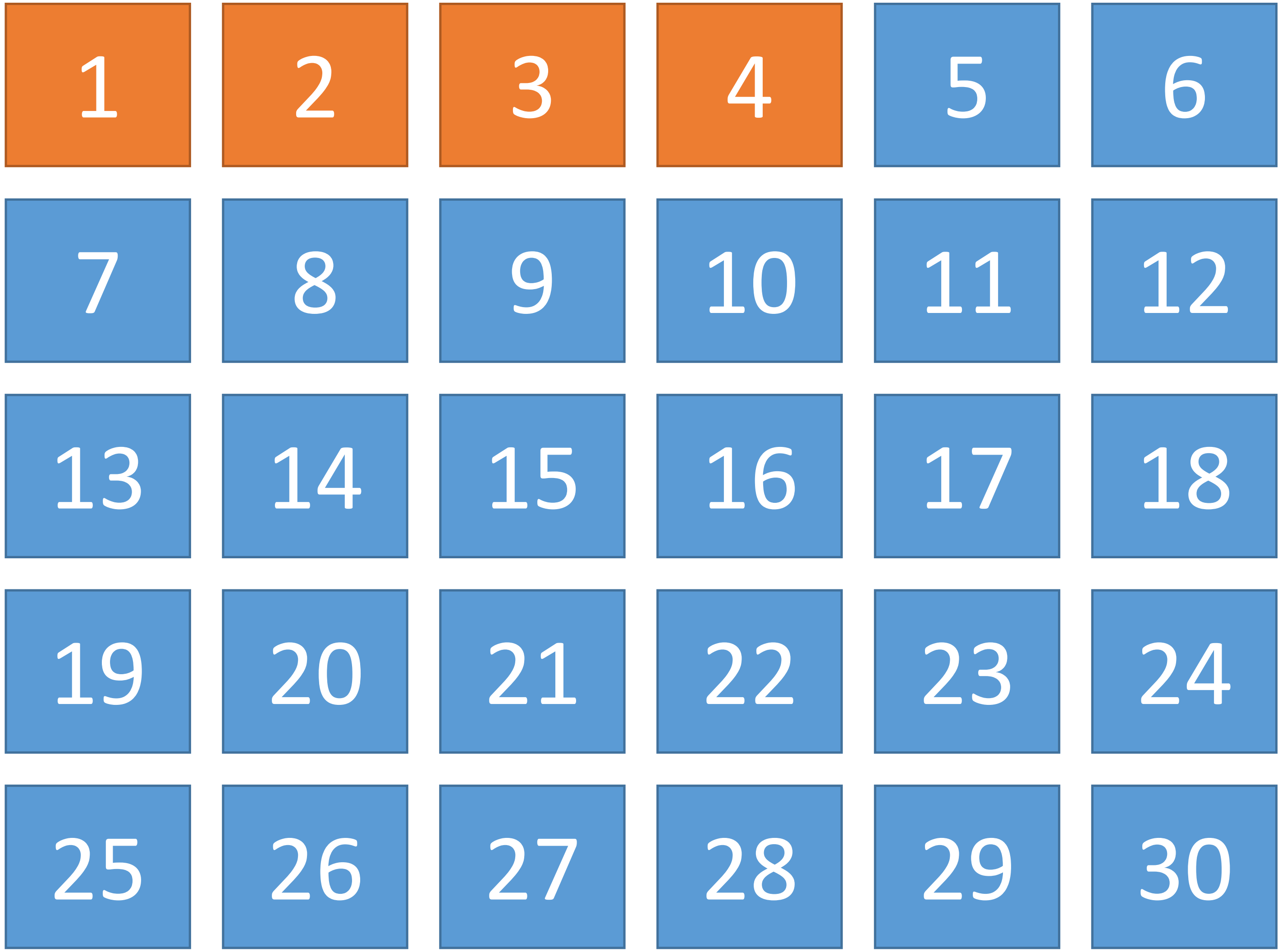
1

Start at one corner,
Go up and down the rows,
Handing out the paper one by one.

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

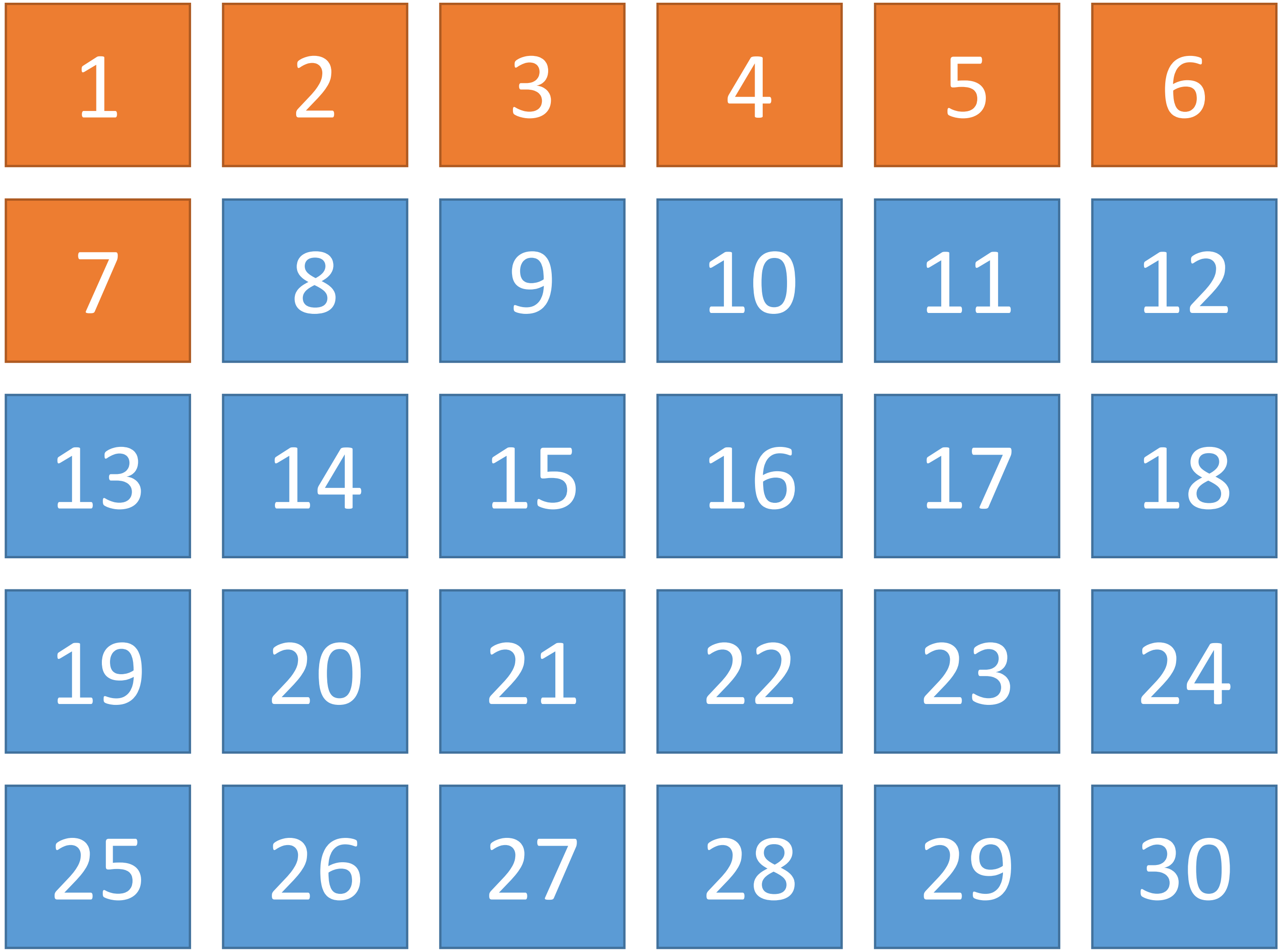
1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

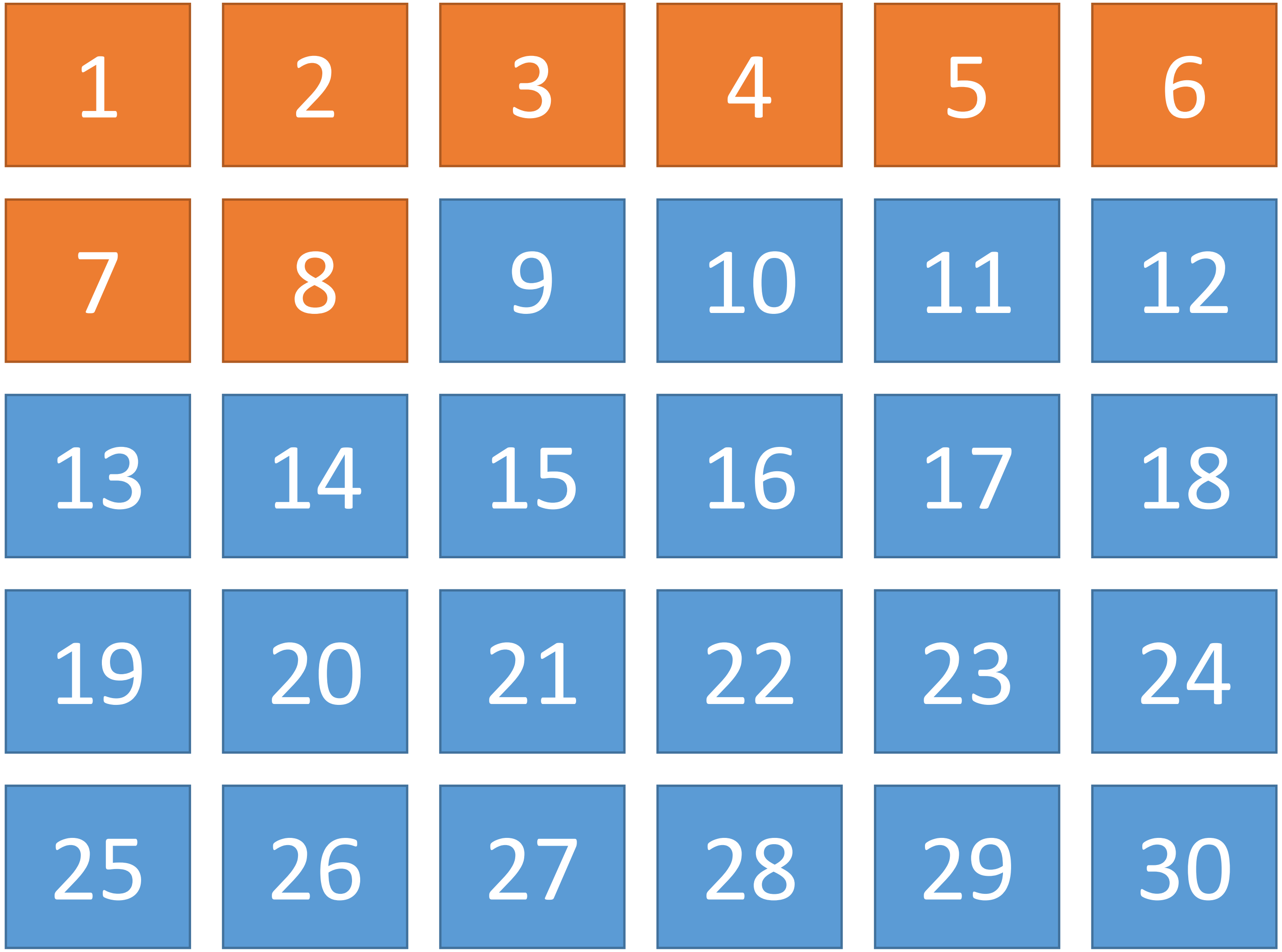
1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

To pass out one by one…

= n

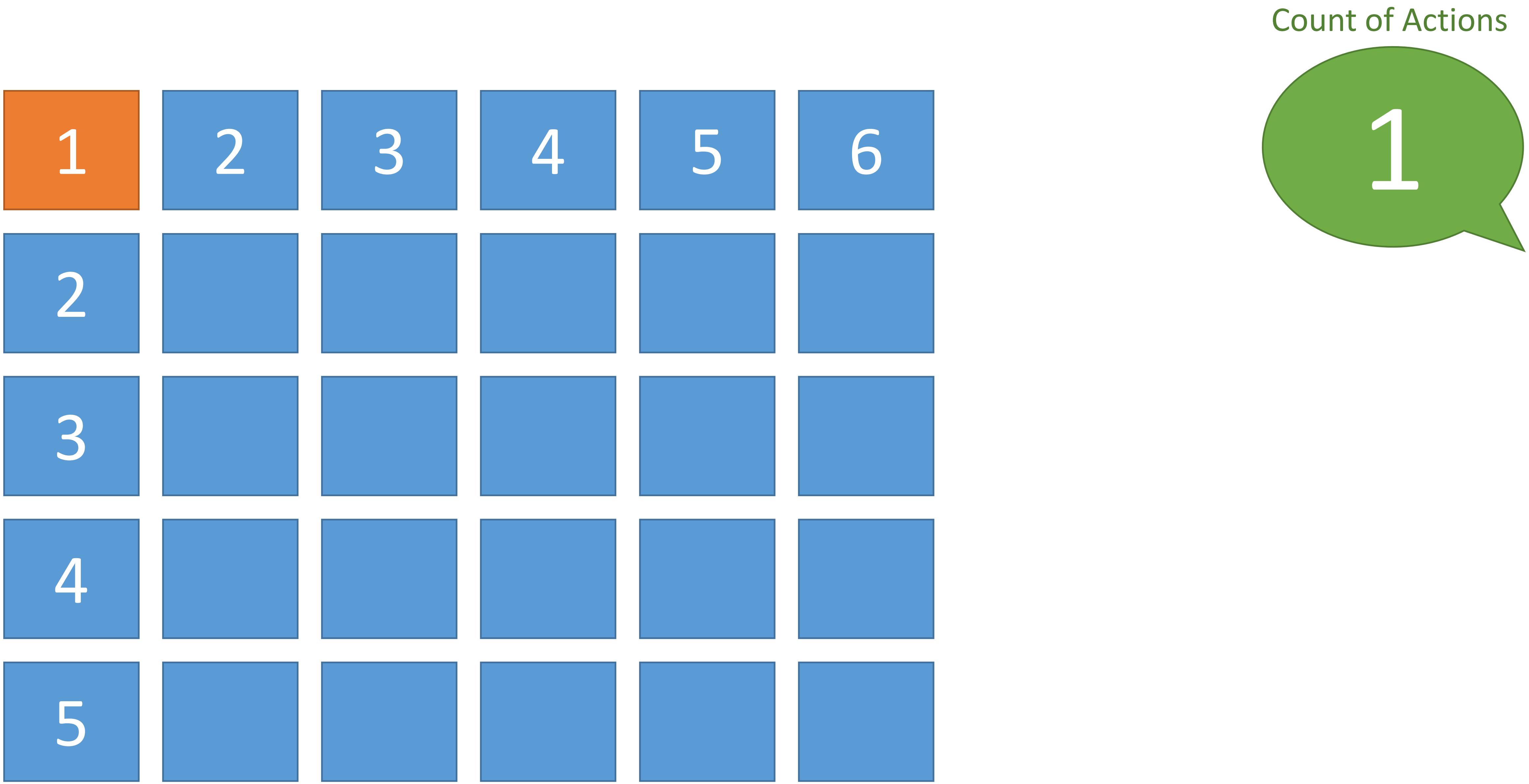
= 30

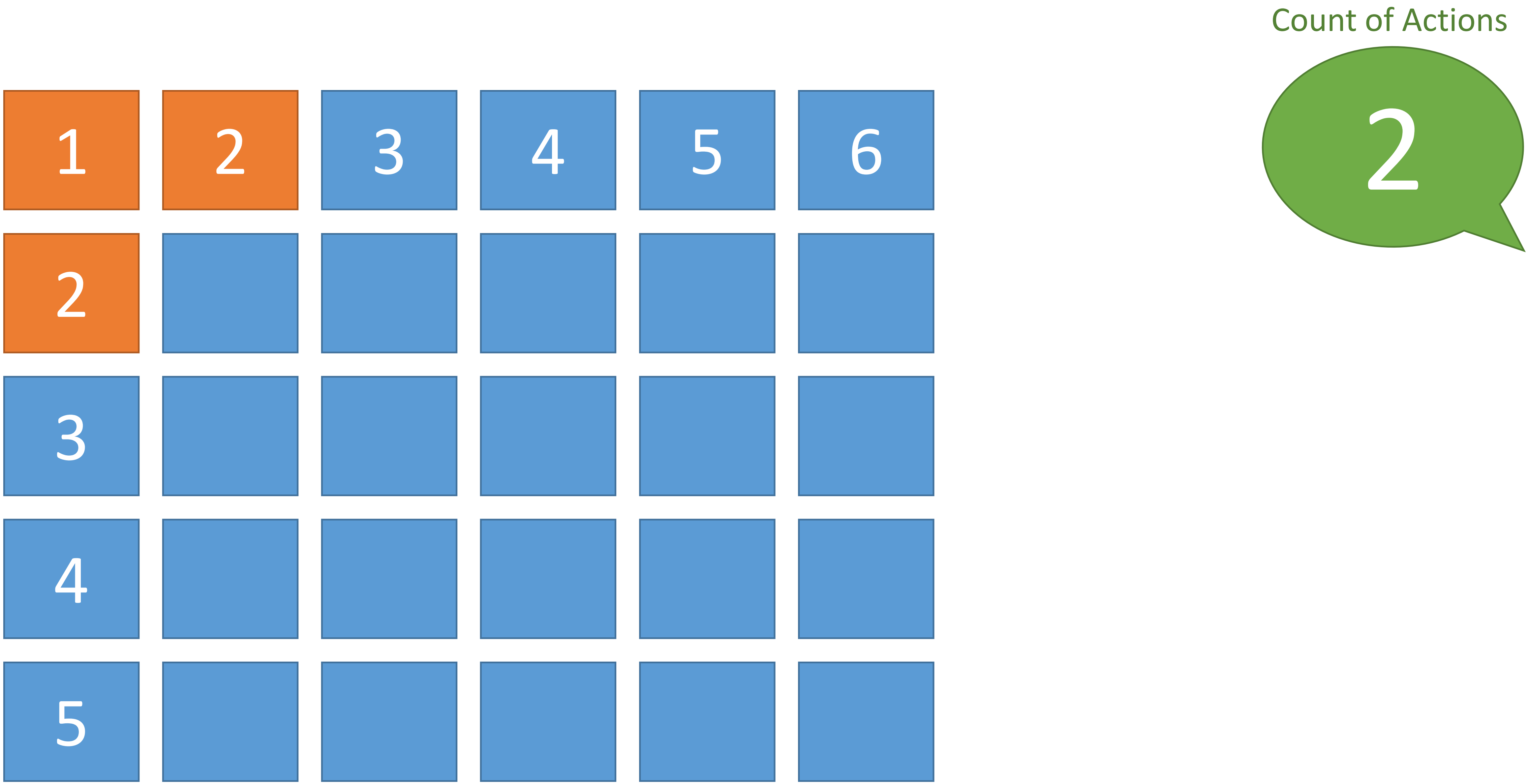
Algorithm

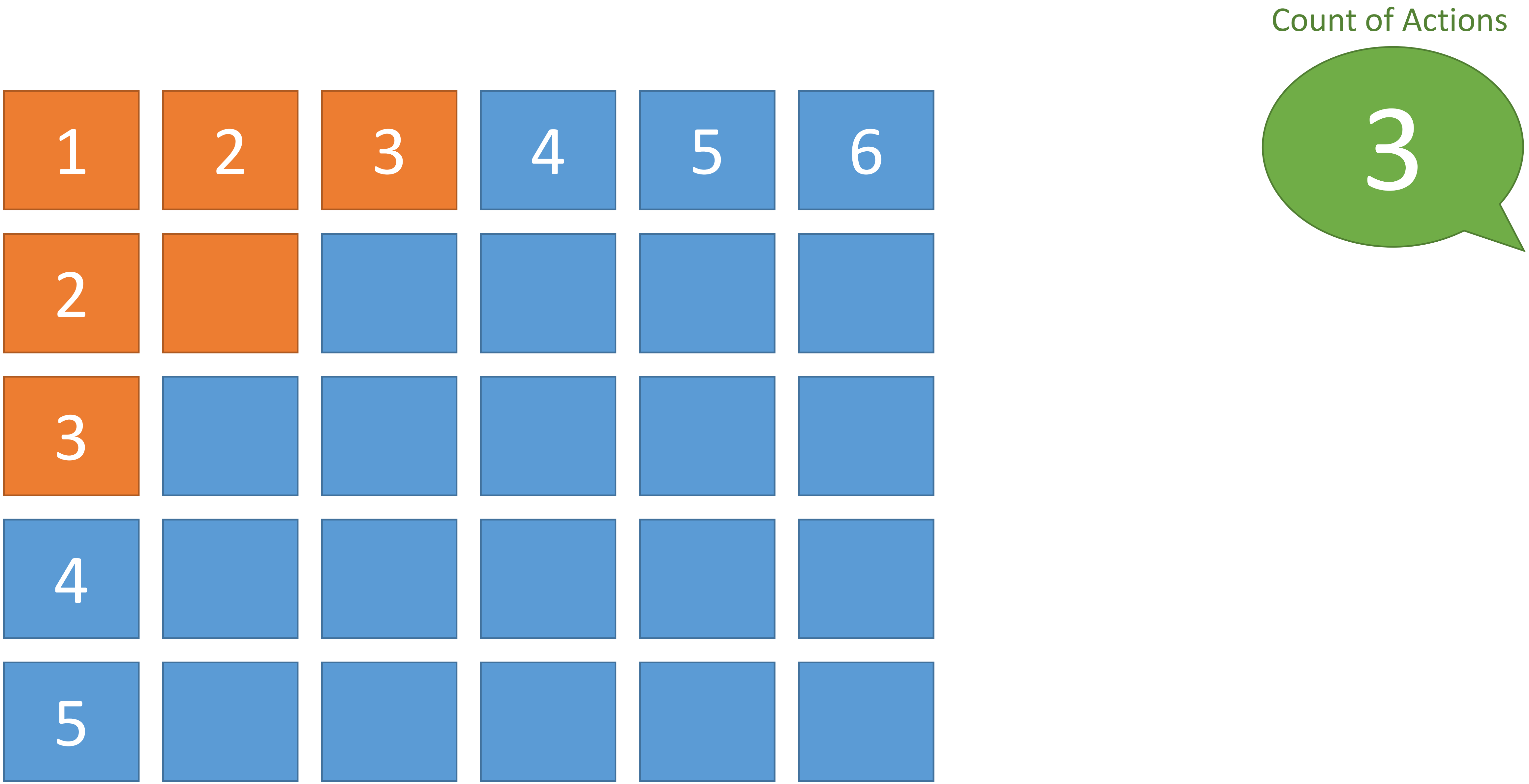
2

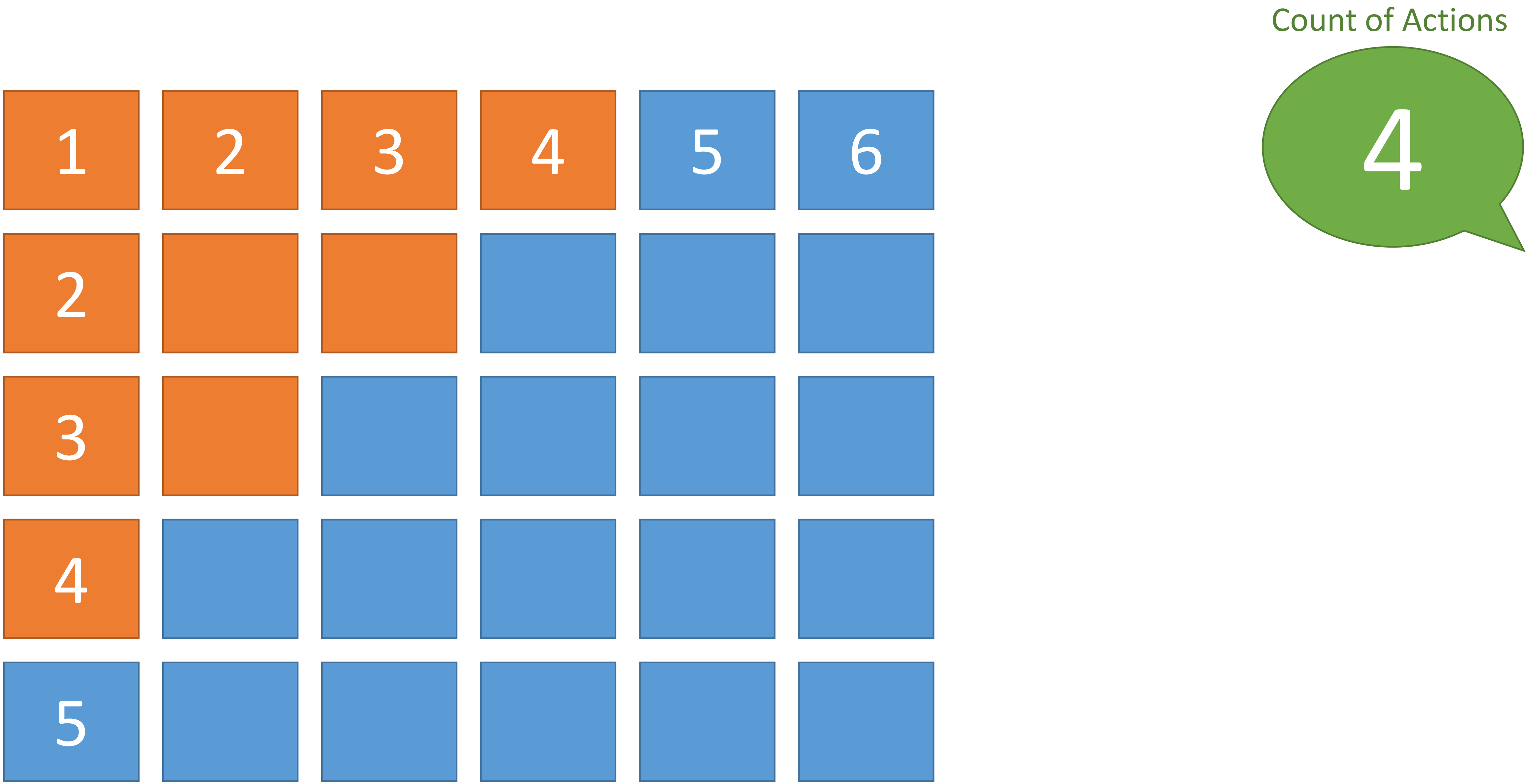
Hand out one pile to each row
Each student passes the pile back.

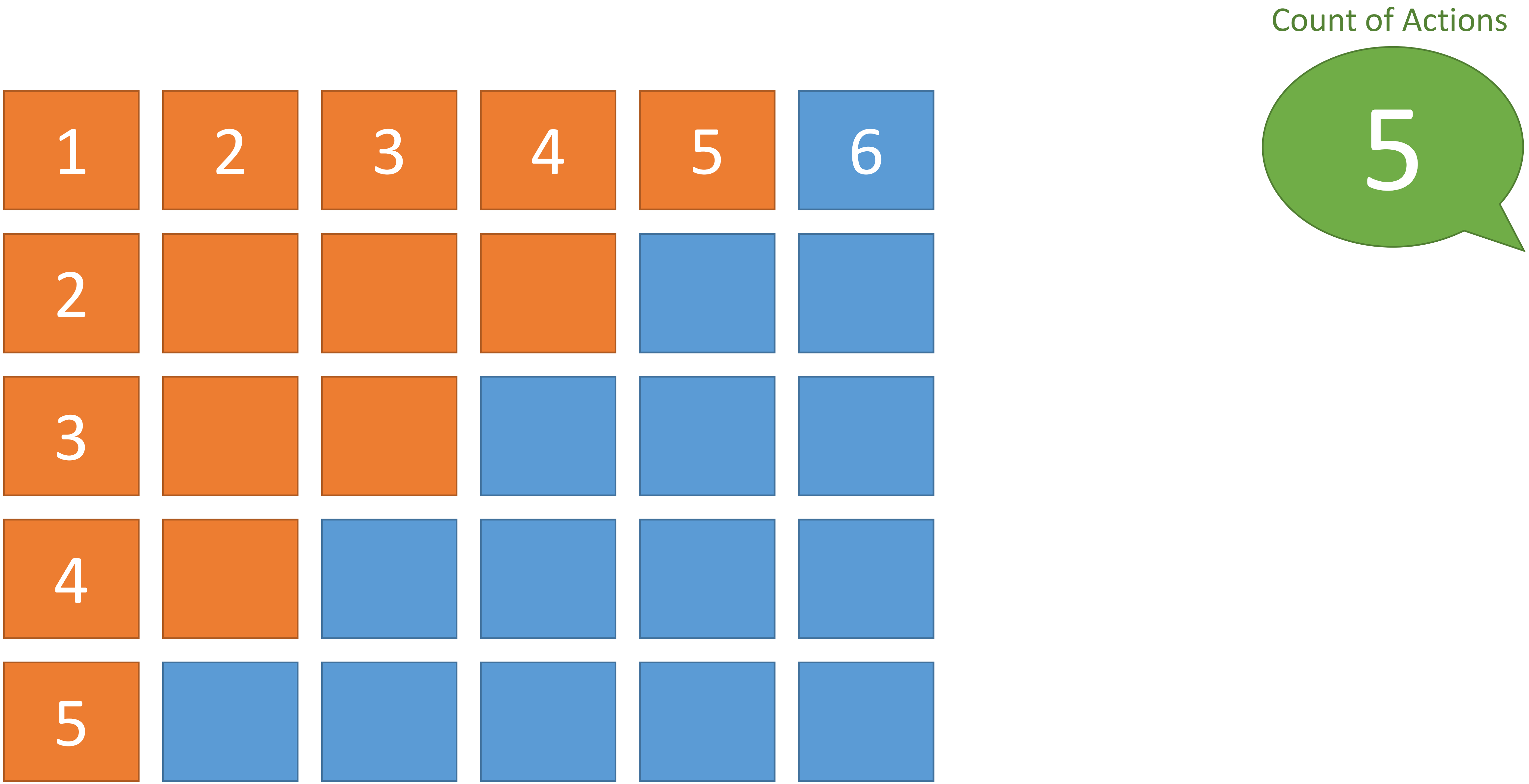
1	2	3	4	5	6
2					
3					
4					
5					


Count of Actions
1
1
2
3
4
5
6
2
3
4
5

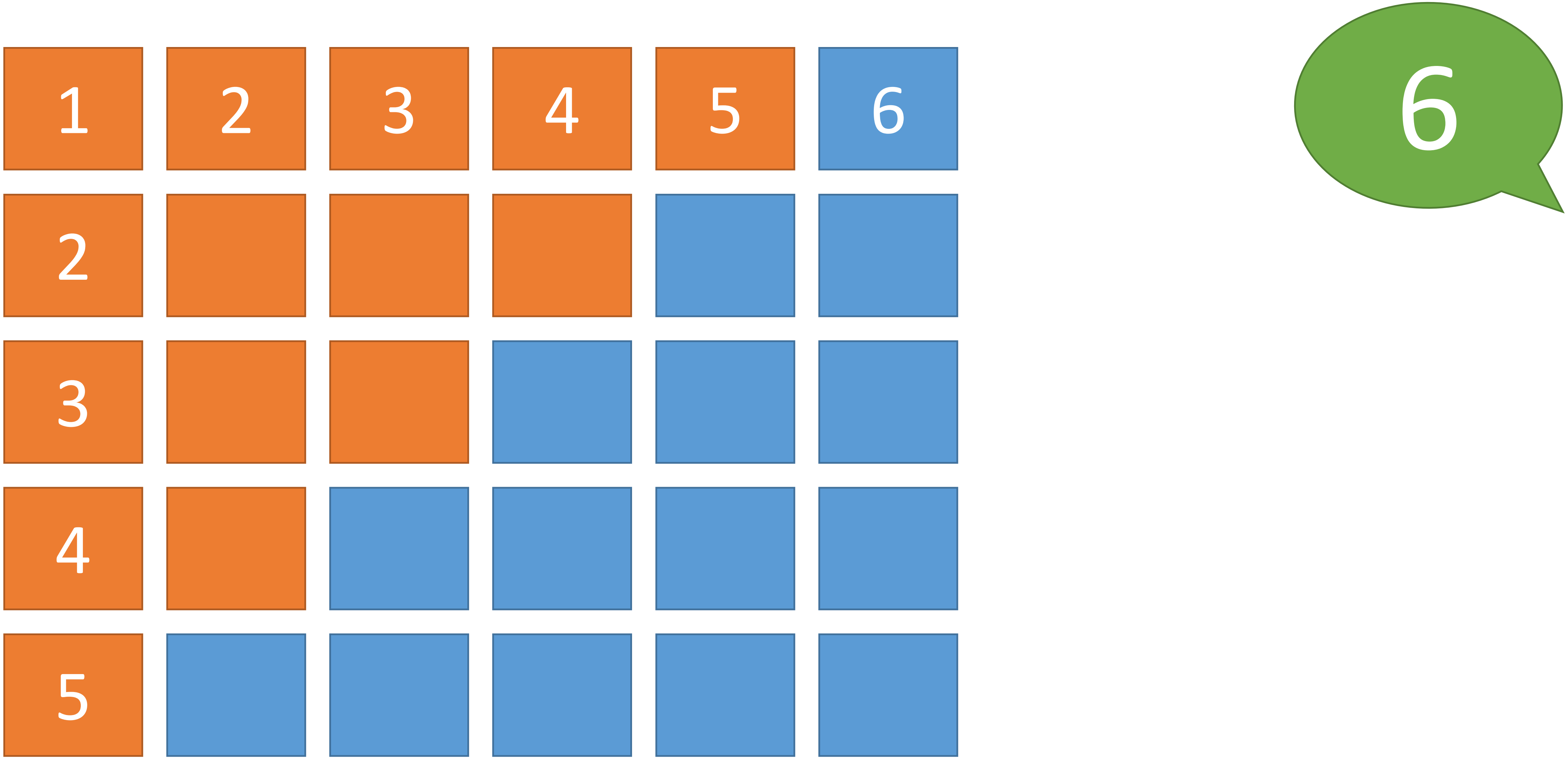

Count of Actions
2
1
2
3
4
5
6
2
3
4
5

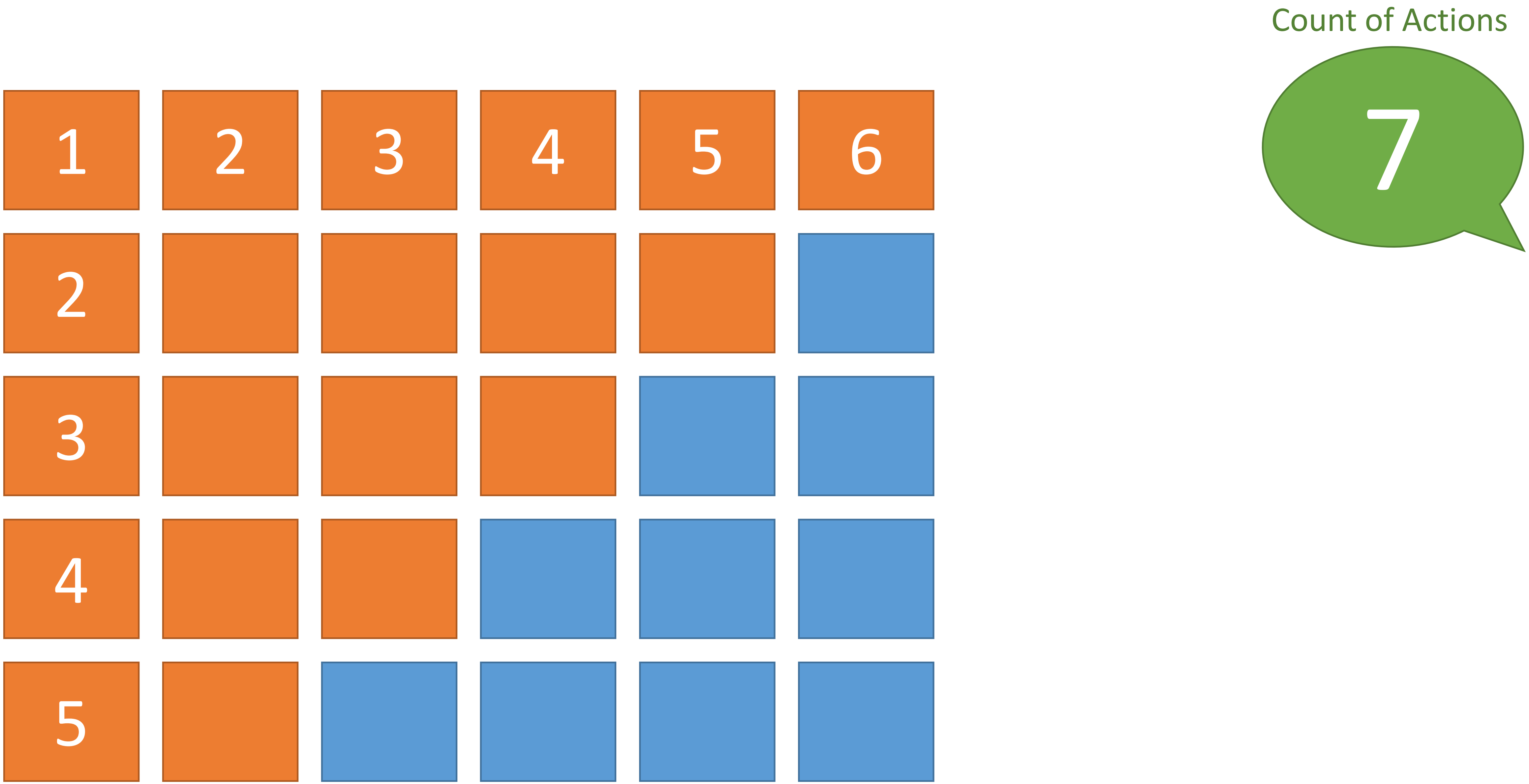

Count of Actions
3
1
2
3
4
5
6
2
3
4
5

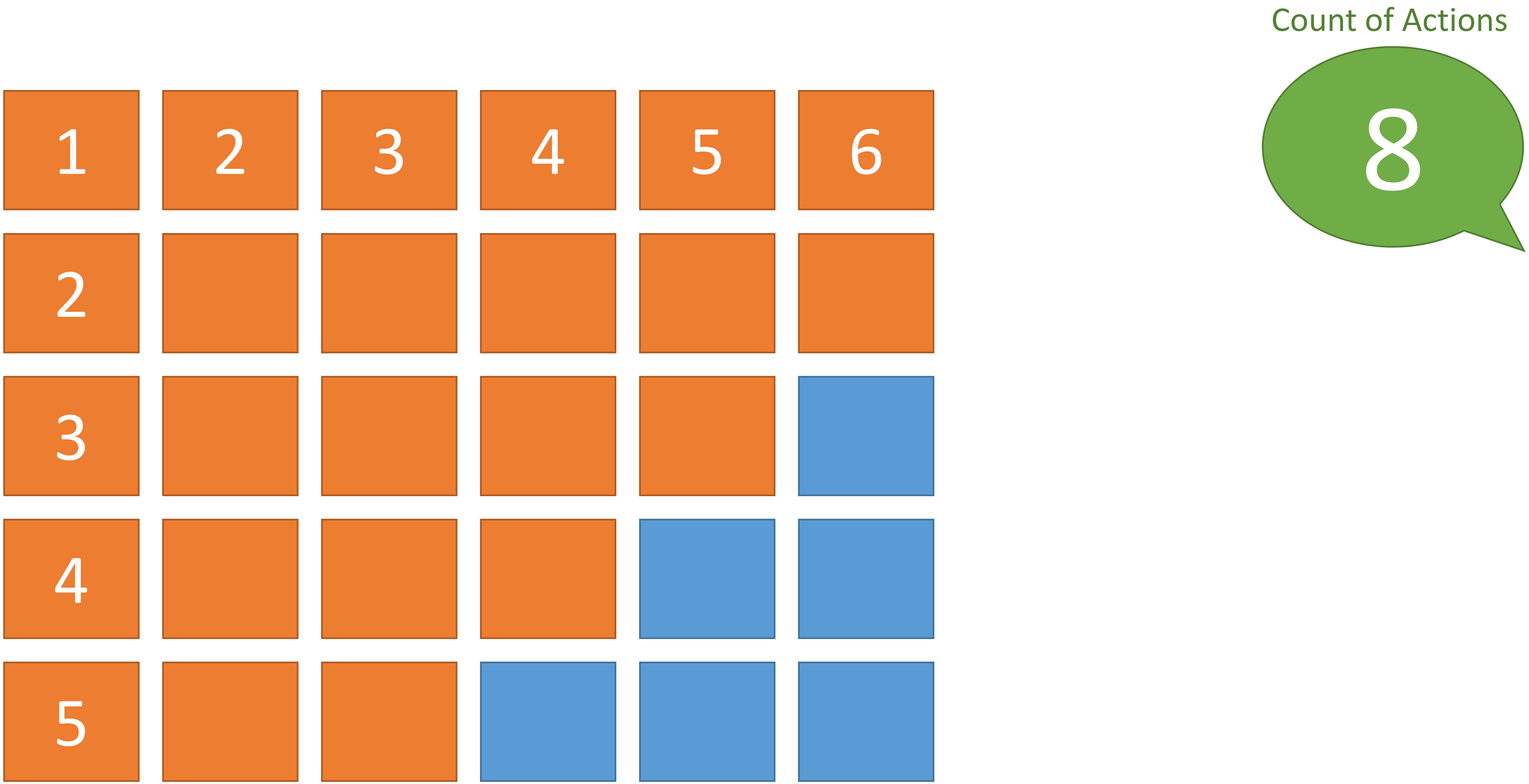

Count of Actions
4
1
2
3
4
5
6
2
3
4
5

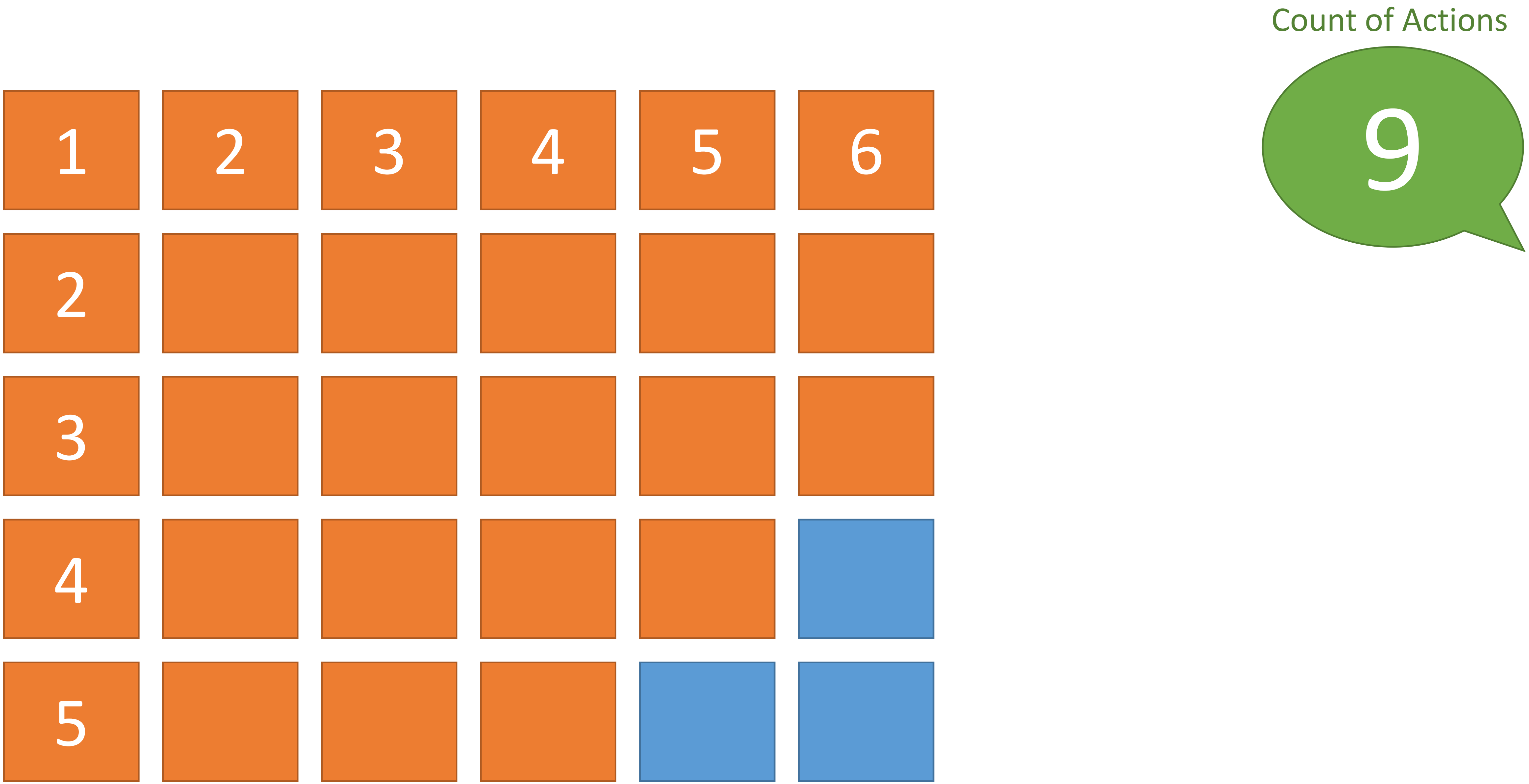

1
2
3
4
5
6
2
3
4
5
Count of Actions
5

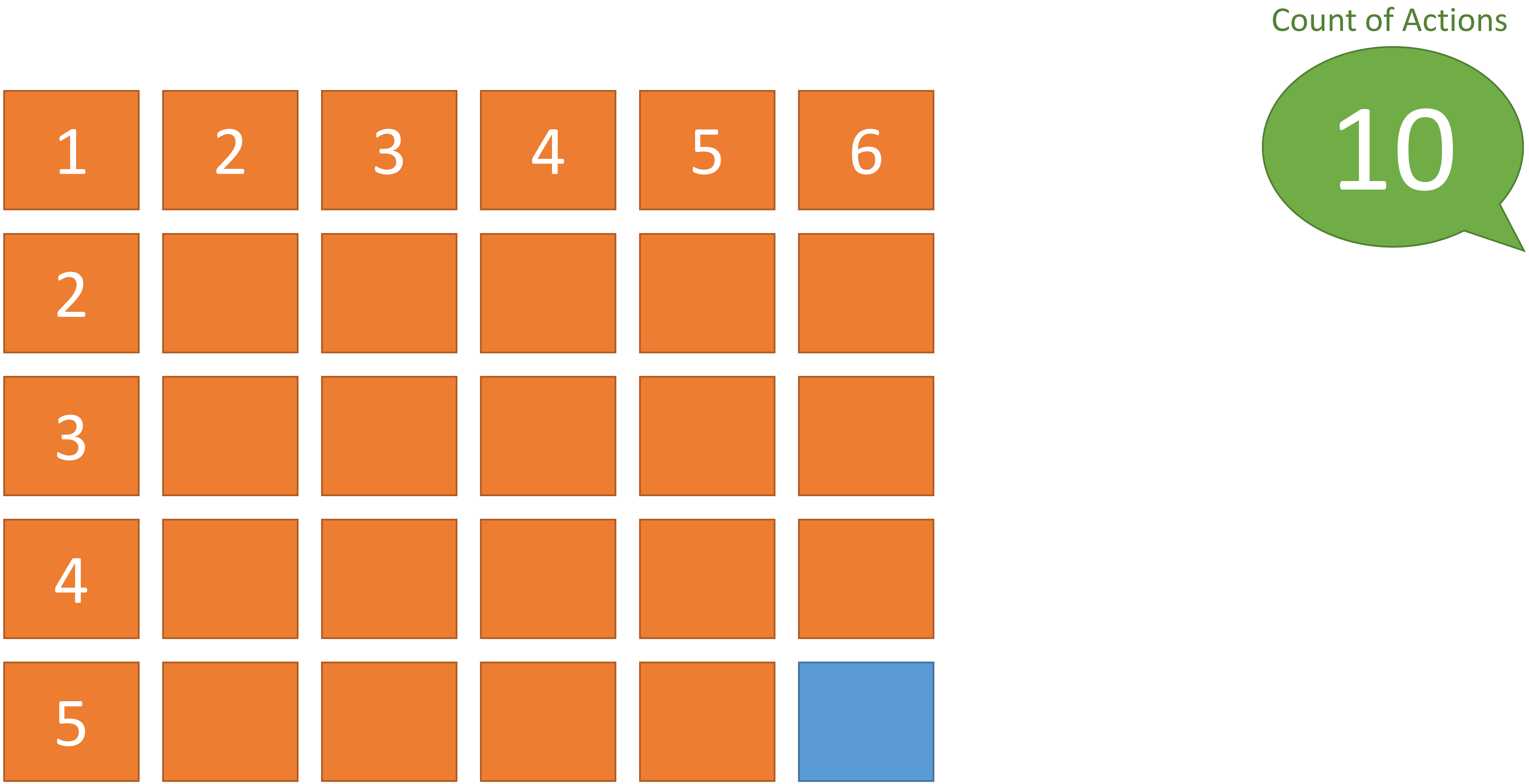
Count of Actions

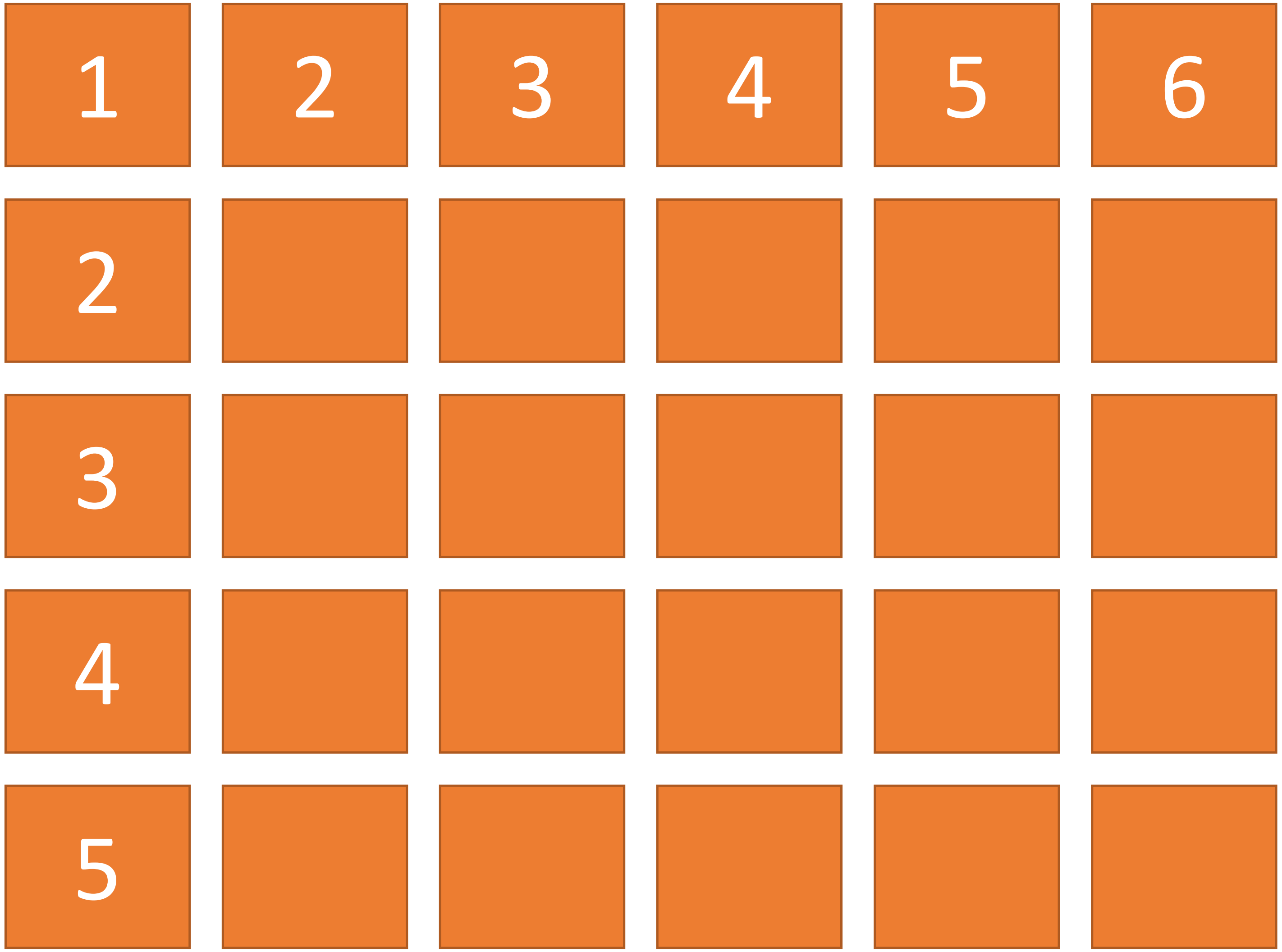



Count of Actions
7
1
2
3
4
5
6
2
3
4
5

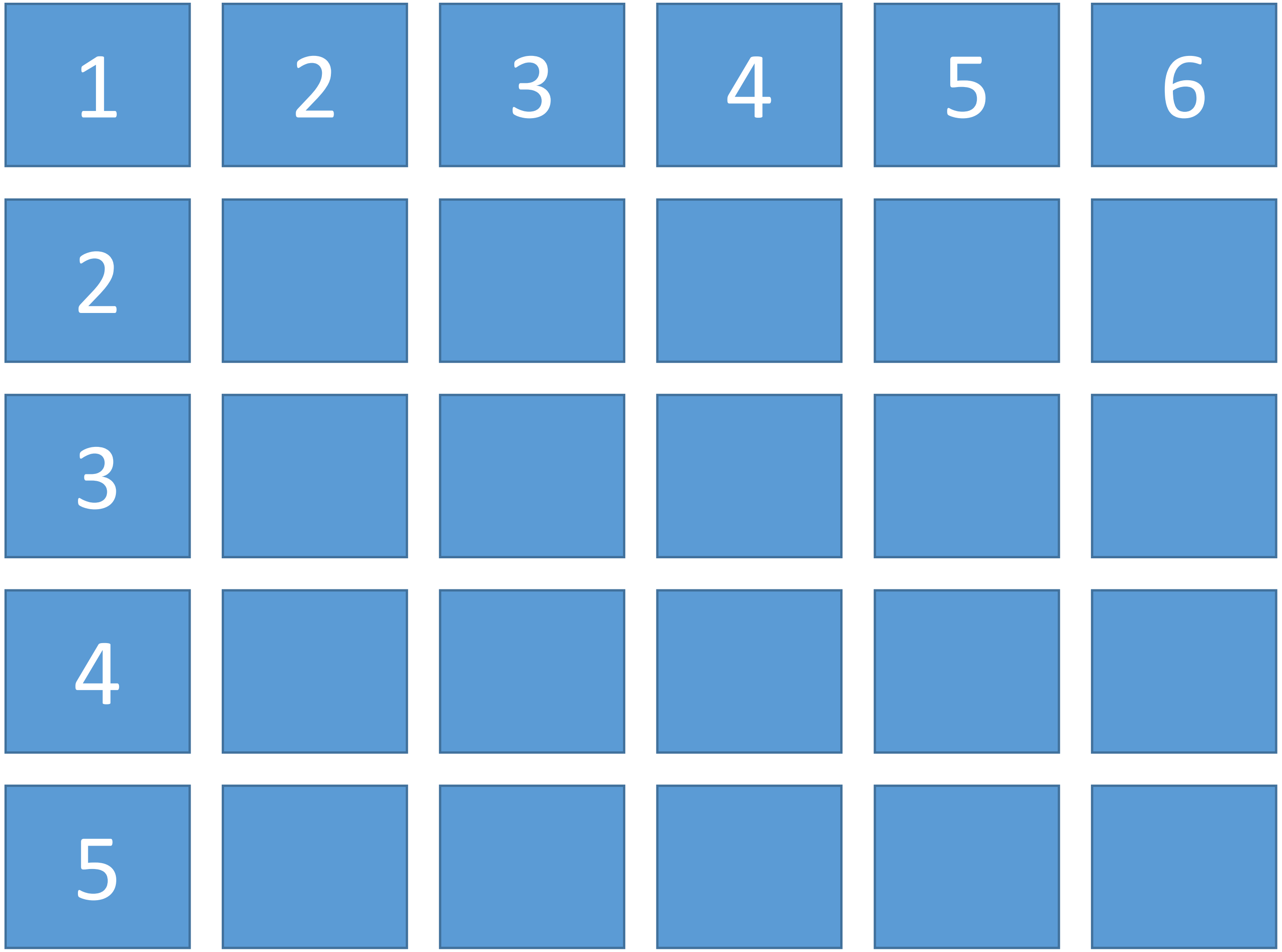

Count of Actions
8
1
2
3
4
5
6
2
3
4
5


Count of Actions
9
1
2
3
4
5
6
2
3
4
5


Count of Actions
10
1
2
3
4
5
6
2
3
4
5



Count of Actions



Count of Actions

One to each row

n/5 rows + (n/6) pass back

= 6 + 5

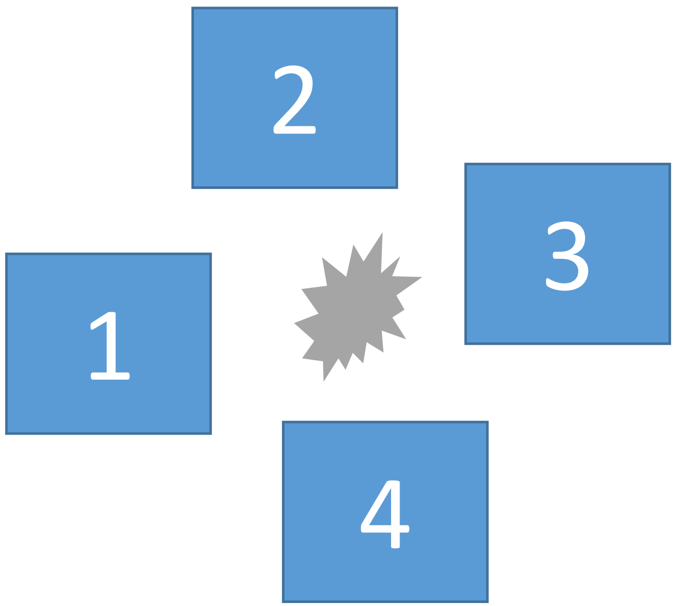
= 11

Algorithm

3

Throw the papers in the air

The student shuffle in to grab them



Throw in the air….

= n/4

= 8 + time to shuffle out….

= 8+ n*20 ?!?

= 608

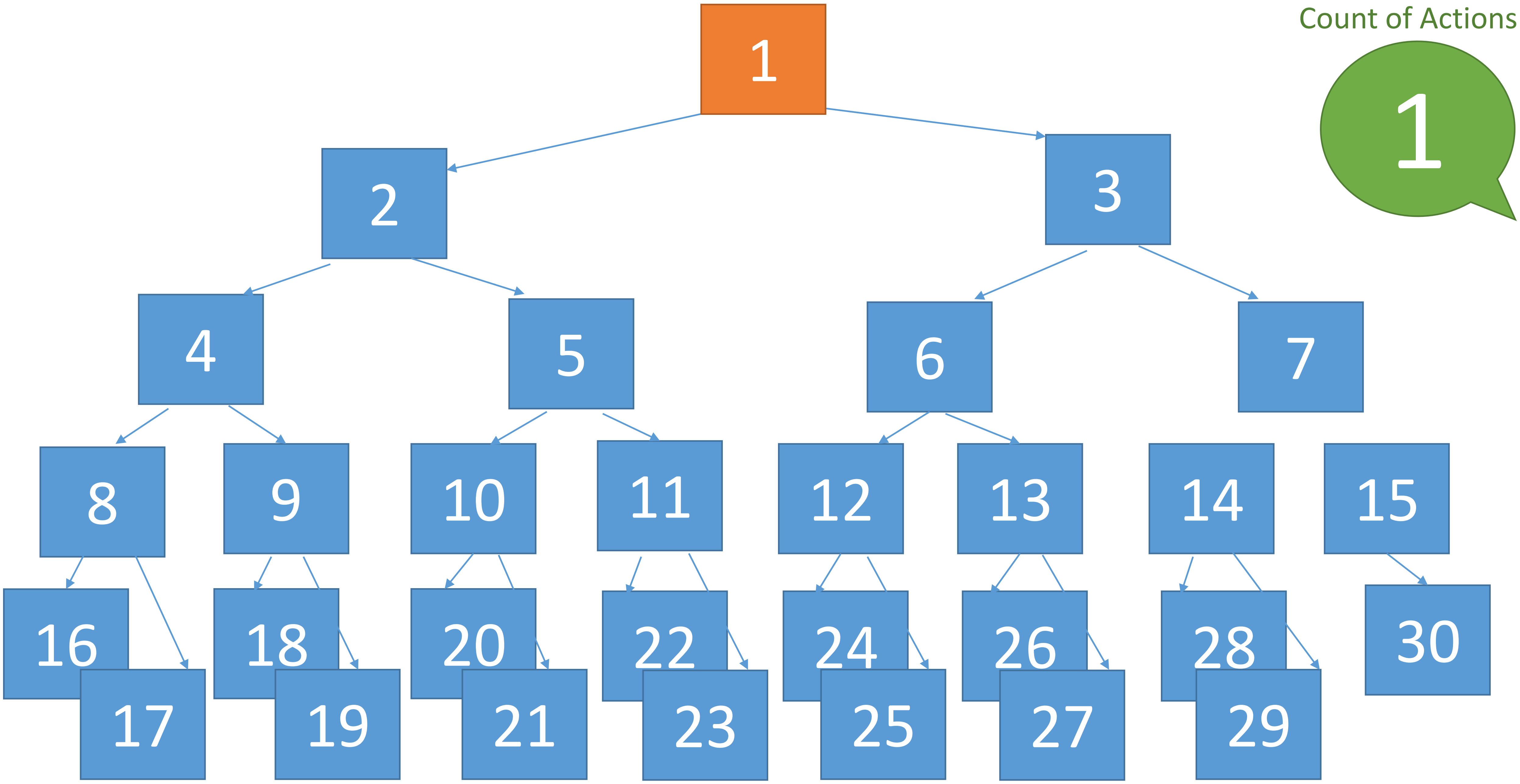
Additional considerations…

It’s chaos….

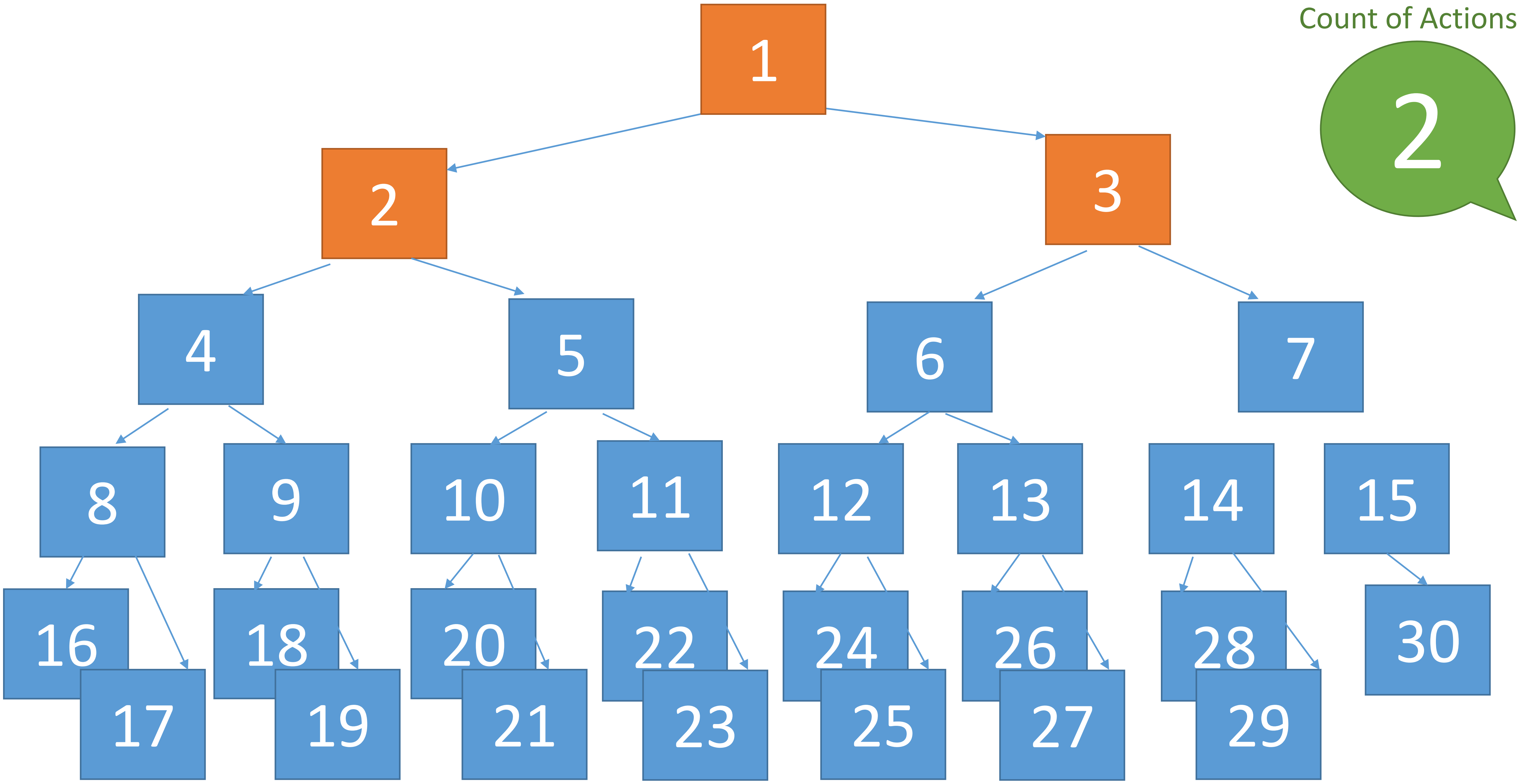
Algorithm

4

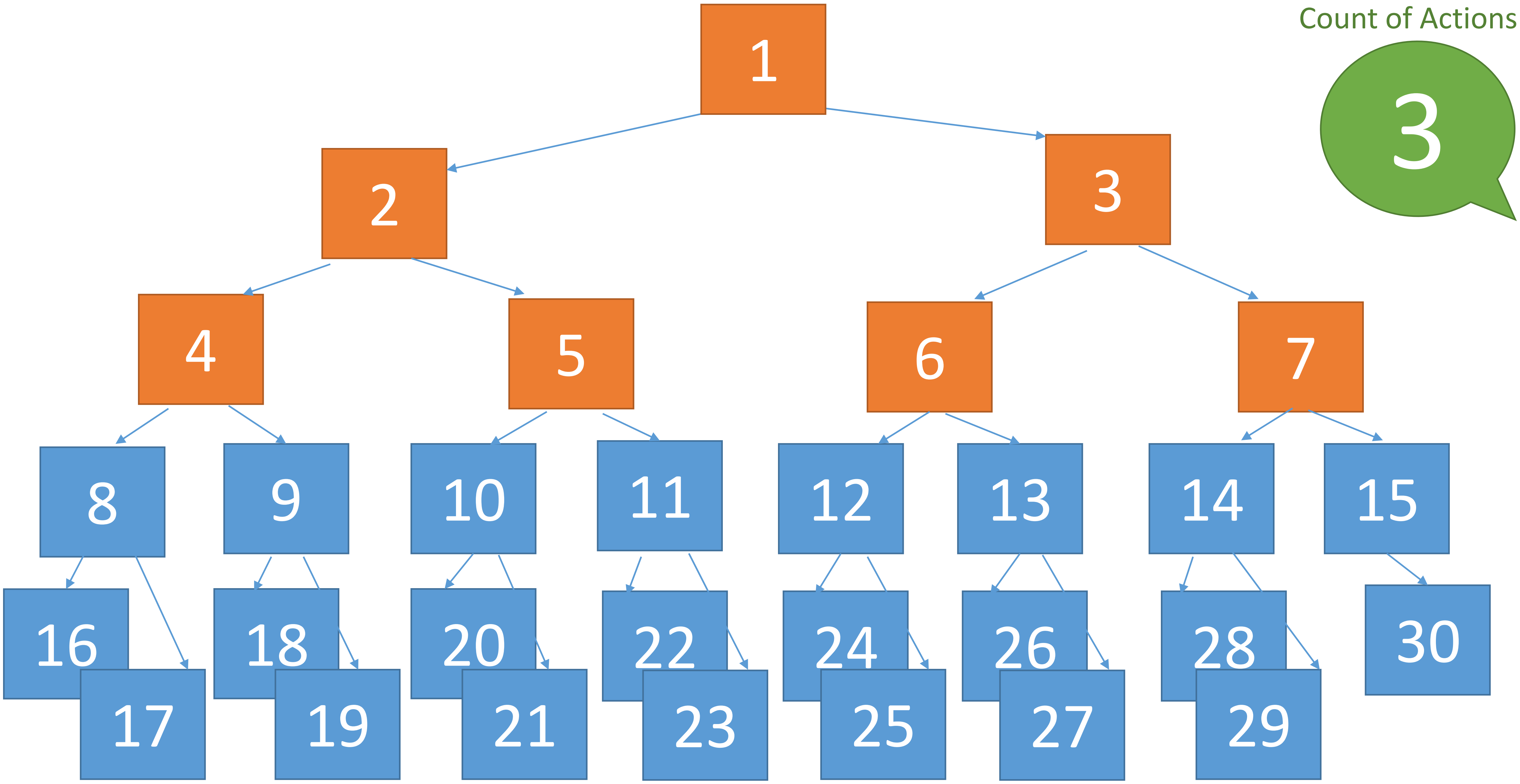
Take one yourself.
Find two people who don’t have the sheet, give each of them half the pile.

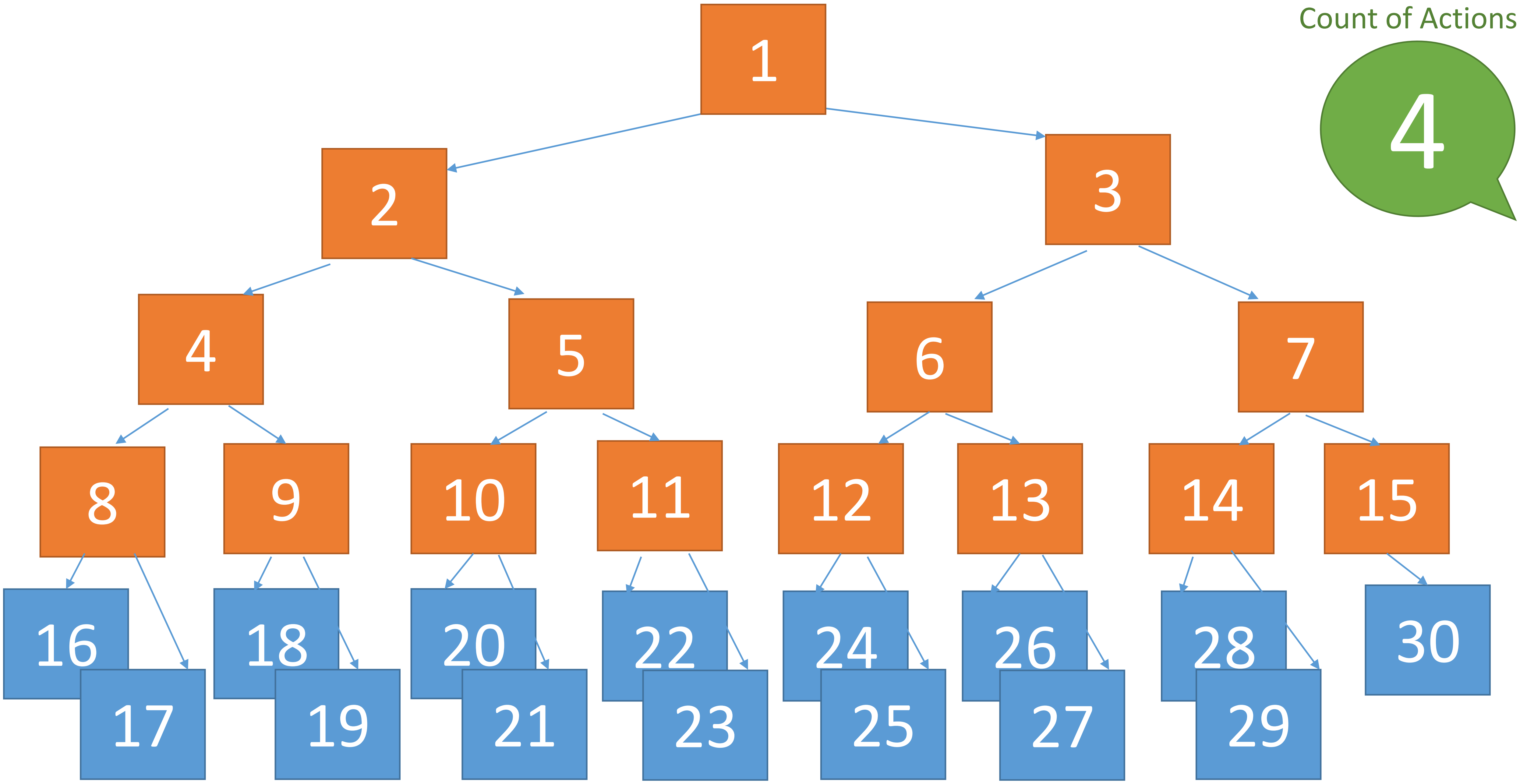


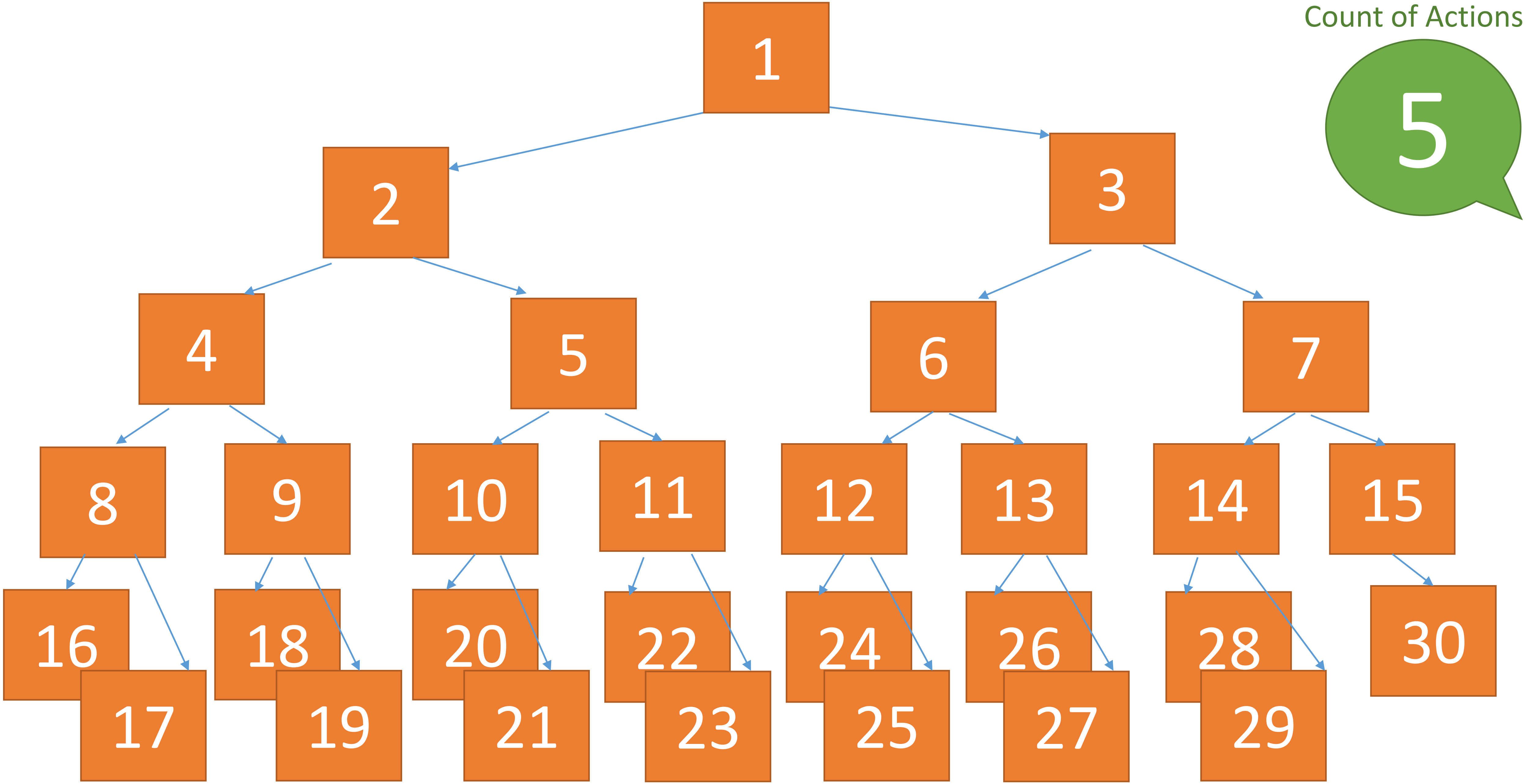
Count of Actions
1
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

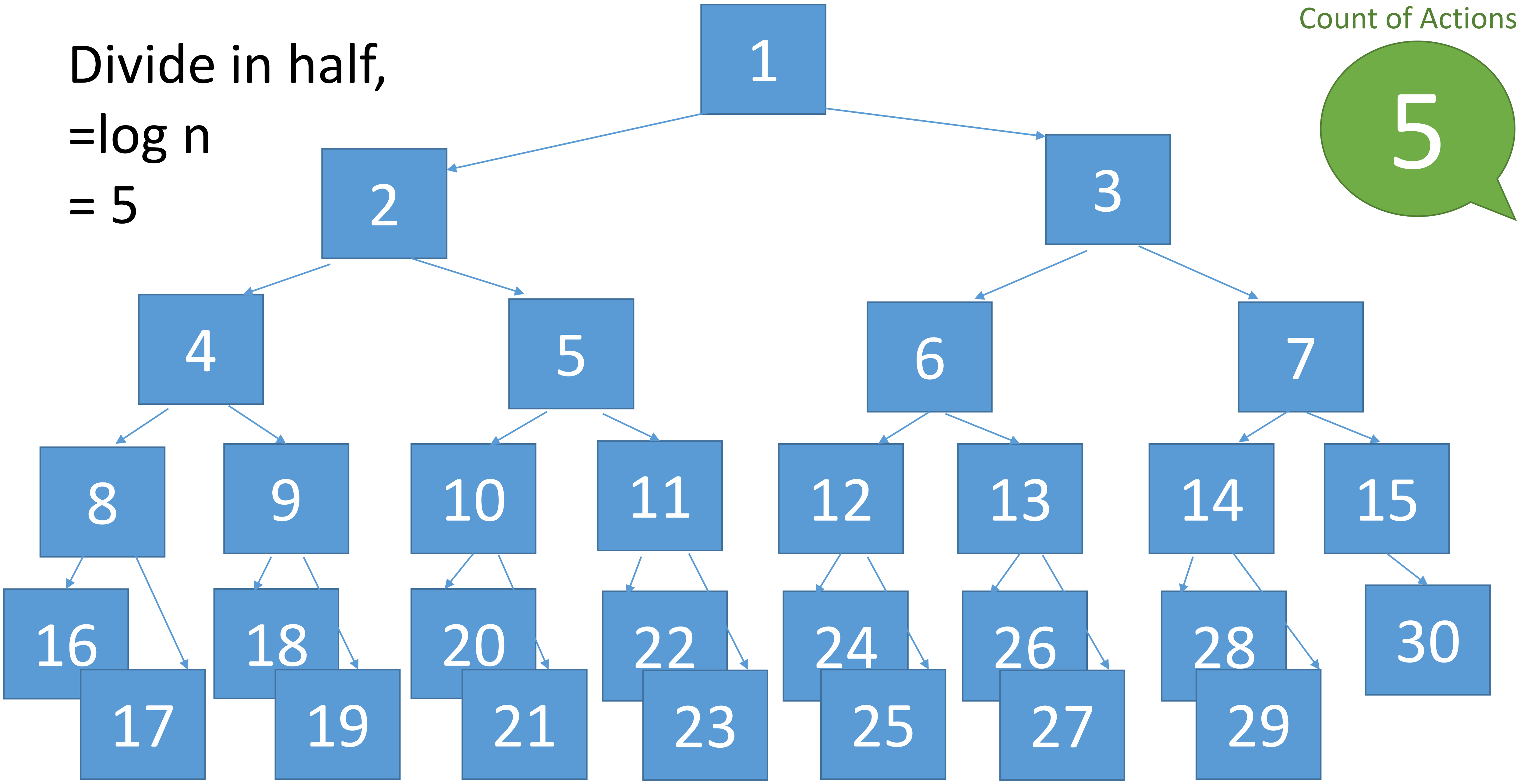


Count of Actions
2
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30


Count of Actions
3
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
Count of Actions
4


Count of Actions
5
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30


Divide in half,
=log n
= 5
Count of Actions
5
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

Big Oh Notation

- A way of measuring algorithm speed
- Uses a mathematical expression for the total number of operations that will be needed, based on the array size
- Meaning of the pieces:
    - O = order
    - n = number of elements in the array
- One loop is O(n)
- One loop inside another is O(n2)

What speed is this?

```
for (int i = 0 ; i < DaysOfWeek.length ; i++)
{
    System.out.println (DaysOfWeek [i]);
}
```

Let’s say the array has 8 values.
array.length is 8.

Tracing the values of i

	A	B	C	D	E	F	G	H	I
1	i	1	2	3	4	5	6	7	8

```
for (int i = 0 ; i < array.length ; i++)
{
    System.out.println (array [i]);
}
```

The loop runs 8 times.

Let’s say the array has 8 values.
array.length is 8.

Tracing the values of i

	A	B	C	D	E	F	G	H	I
1	i	1	2	3	4	5	6	7	8

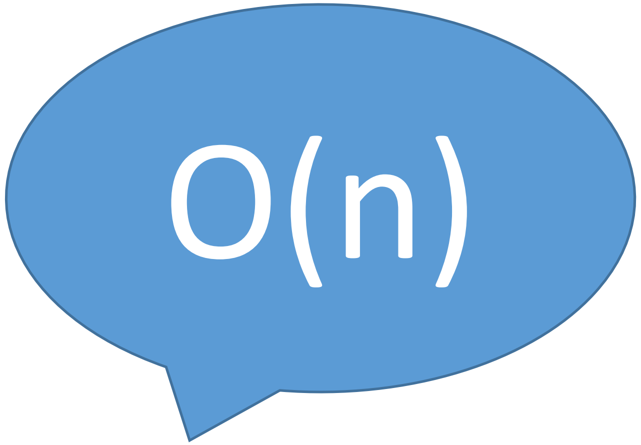
```
for (int i = 0 ; i < array.length ; i++)
{
     System.out.println (array [i]);
}
```

The loop runs 8 times.

O(n)

```
double min = price [0];
for (int i = 1 ; i < price.length ; i++)
{
    if (min > price [i])
        min = price [i];
}
System.out.println ("The lowest price is: $" + min);
```

```java
double min = price [0];
for (int i = 1 ; i < price.length ; i++)
{
    if (min > price [i])
        min = price [i];
}
System.out.println ("The lowest price is: $" + min);
```



```
for (int i = 0 ; i < a.length - 1 ; i++)
{
    for (int j = 0 ; j < a.length - 1 - i ; j++)
    { // compare the two neighbours
        if (a [j + 1] < a [j])
        { //swap the neighbours if necessary
            int temp = a [j];
            a [j] = a [j + 1];
            a [j + 1] = temp;
        }
    }
}
```

The outer loop runs roughly n times

The inner loop runs roughly n times.

Thus, this code is run n*n times

Let's say the array has 8 values.
array.length is 8.

Tracing the values of i and j

	A	B	C	D	E	F	G	H	I
1	i	j	j	j	j	j	j	j	j
2	1	1	2	3	4	5	6	7	8
3	2	1	2	3	4	5	6	7	8
4	3	1	2	3	4	5	6	7	8
5	4	1	2	3	4	5	6	7	8
6	5	1	2	3	4	5	6	7	8
7	6	1	2	3	4	5	6	7	8
8	7	1	2	3	4	5	6	7	8
9	8	1	2	3	4	5	6	7	8

```
for (int i = 0 ; i < a.length - 1 ; i++)
{
    for (int j = 0 ; j < a.length - 1 - i ; j++)
    { // compare the two neighbours
        if (a [j + 1] < a [j])
        { //swap the neighbours if necessary
            int temp = a [j];
            a [j] = a [j + 1];
            a [j + 1] = temp;
        }
    }
}
```

The inner loop runs roughly 64 times.

Tracing the values of i and j

	A	B	C	D	E	F	G	H	I
1	i	j	j	j	j	j	j	j	j
2	1	1	2	3	4	5	6	7	8
3	2	1	2	3	4	5	6	7	8
4	3	1	2	3	4	5	6	7	8
5	4	1	2	3	4	5	6	7	8
6	5	1	2	3	4	5	6	7	8
7	6	1	2	3	4	5	6	7	8
8	7	1	2	3	4	5	6	7	8
9	8	1	2	3	4	5	6	7	8

Let’s say the array has 8 values.
array.length is 8.

```
for (int i = 0 ; i < a.length - 1 ; i++)
{
    for (int j = 0 ; j < a.length - 1 - i ; j++)
    { // compare the two neighbours
        if (a [j + 1] < a [j])
        { //swap the neighbours if necessary
            int temp = a [j];
            a [j] = a [j + 1];
            a [j + 1] = temp;
        }
    }
}
```

The inner loop runs roughly 64 times.

$O(n^2)$

Algorithm speeds

(in order from fastest to slowest)

1. $O(1)$, constant time
2. $O(\log n)$, logarithmic time
3. $O(n)$, linear time
4. $O(n \log n)$
5. $O(n^2)$, quadratic time
6. $O(n^3)$, cubic time
7. $O(n^4)$
8. $O(2^n)$, exponential time

Where would $O(n^2 \log n)$ go?

Algorithm speeds
(in order from fastest to slowest)

1. $O(1)$, constant time
2. $O(\log n)$, logarithmic time
3. $O(n)$, linear time
4. $O(n \log n)$
5. $O(n^2)$, quadratic time
6. $O(n^3)$, cubic time
7. $O(n^4)$
8. $O(2^n)$, exponential time

Where would $O(n^2 \log n)$ go?

	A	B	C	D	E	F	G	H
1	n	O(1)	O(logn)	O(n)	O(nlogn)	O(n^2)	O(n^3)	O(2^n)
2	4	1	2	4	8	16	64	16
3	10	1	3.3219	10	33.2193	100	1000	1024
4	16	1	4	16	64	256	4096	65536
5	100	1	6.6439	100	664.386	10000	1000000	1.26765E+30
6	1000	1	9.9658	1000	9965.78	1000000	1000000000	1.0715E+301
7	10000	1	13.288	10000	132877	100000000	1E+12	#NUM!

The Grade 11 algorithms and their speeds:

Speed	Algorithms
$O(1)$	Swap, add, finding the length
$O(\log n)$	Binary search
$O(n)$	print, min, max, sum, average, delete, linear search, Bin sort
$O(n \log n)$	Quicksort, Mergesort
$O(n^2)$	Selection sort, Bubblesort

How fast is the min algorithm?

How fast is the swap algorithm?

How fast is the min algorithm?

O(n)

linear

How fast is the swap algorithm?

How fast is the min algorithm?

O(n)

linear

How fast is the swap algorithm?

O(1)

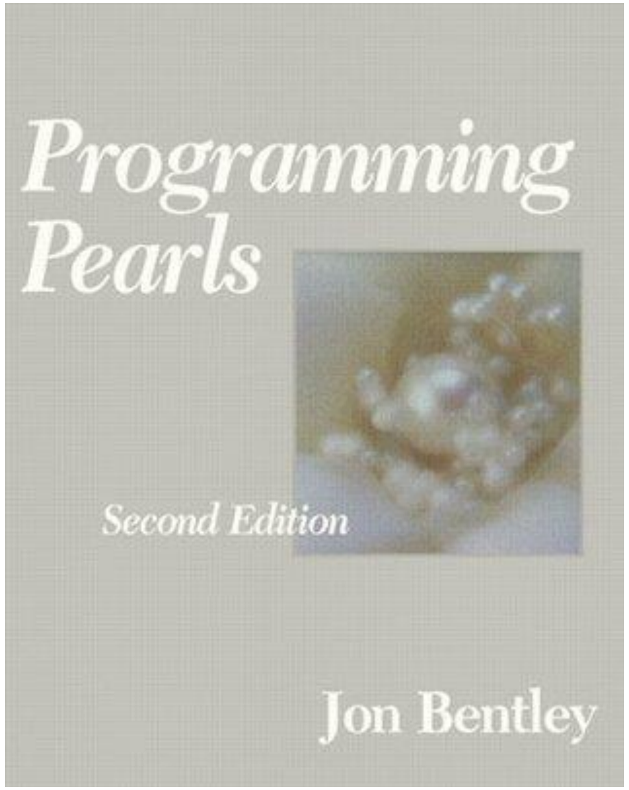
Constant time

Does it really work?

Jon Bentley describes an experiment in *Programming Pearls*, p. 75. The problem is to take a list of N real numbers and return the maximum sum found in any *contiguous* sublist. For example:

31	-41	59	26	-53	58	97	-93	-23	84

He describes four algorithms to solve the problem. They are $O(n^3)$, $O(n^2)$, $O(n \lg n)$, and $O(n)$. To prove that constant factors don't matter much, he deliberately tried to make the constant factors of the $O(n^3)$ and $O(n)$ algorithms differ by as much as possible.



$O(n)$ on a TRS-80 hobbyist computer

$O(n^3)$ on a Cray supercomputer

$O(n^3)$ algorithm: Cray-1, finely-tuned Fortran, $3.0n^3$ nanoseconds

$O(n)$ algorithm: TRS-80, interpreted Basic, $19.5n$ milliseconds = $19,500,000n$ nanoseconds

N	Cray – great hw, 0(n^3) – awful sw	TRS-80 – bad hw, O(n) – great sw
10	0.000003 sec	0.2 sec
100		
1000		
2500		
10,000		
100,000		
1,000,000		

N	Cray – great hw, 0(n^3) – awful sw	TRS-80 – bad hw, O(n) – great sw
10	0.000003 sec	0.2 sec
100	0.003 sec	2.0 sec
1000		
2500		
10,000		
100,000		
1,000,000		

N	Cray – great hw, 0(n^3) – awful sw	TRS-80 – bad hw, O(n) – great sw
10	0.000003 sec	0.2 sec
100	0.003 sec	2.0 sec
1000	3 sec	20 sec
2500		
10,000		
100,000		
1,000,000		

N	Cray – great hw, 0(n^3) – awful sw	TRS-80 – bad hw, O(n) – great sw
10	0.000003 sec	0.2 sec
100	0.003 sec	2.0 sec
1000	3 sec	20 sec
2500	47 sec	49 sec
10,000		
100,000		
1,000,000		

N	Cray – great hw, 0(n^3) – awful sw	TRS-80 – bad hw, O(n) – great sw
10	0.000003 sec	0.2 sec
100	0.003 sec	2.0 sec
1000	3 sec	20 sec
2500	47 sec	49 sec
10,000	50 min	3.25 min
100,000		
1,000,000		

N	Cray – great hw, 0(n^3) – awful sw	TRS-80 – bad hw, O(n) – great sw
10	0.000003 sec	0.2 sec
100	0.003 sec	2.0 sec
1000	3 sec	20 sec
2500	47 sec	49 sec
10,000	50 min	3.25 min
100,000	34.7 days	32.5 min
1,000,000		

N	Cray – great hw, 0(n^3) – awful sw	TRS-80 – bad hw, O(n) – great sw
10	0.000003 sec	0.2 sec
100	0.003 sec	2.0 sec
1000	3 sec	20 sec
2500	47 sec	49 sec
10,000	50 min	3.25 min
100,000	34.7 days	32.5 min
1,000,000	95	5.4

N	Cray – great hw, 0(n^3) – awful sw	TRS-80 – bad hw, O(n) – great sw
10	0.000003 sec	0.2 sec
100	0.003 sec	2.0 sec
1000	3 sec	20 sec
2500	47 sec	49 sec
10,000	50 min	3.25 min
100,000	34.7 days	32.5 min
1,000,000	95 years	5.4 hours

The Moral of Bentley's Example

Fast hardware cannot compensate for a slow algorithm.