Recursion with Searching and Sorting

Binary Search

Quicksort

Linear search:

http://www.cosc.canterbury.ac.nz/mukundan/dsal/LSearch.html

Linear search:

http://www.cosc.canterbury.ac.nz/mukundan/dsal/LSearch.html

What could we change to get a better searching algorithm?

Linear search:

http://www.cosc.canterbury.ac.nz/mukundan/dsal/LSearch.html

Binary search:

http://www.cosc.canterbury.ac.nz/mukundan/dsal/BSearch.html

Binary Search Algorithm

```
input: a sorted list of data
input: a value to find
output: index of location of value, or -1

set left to 0, right to listSize-1

while (left <= right):
    set mid to (left + ((right - left) / 2))
    if (item at mid) < (value to find):
        left = mid + 1
    otherwise if (item at mid) > (value to find):
        right = mid - 1
    otherwise:
        return mid

return -1
```

Binary Search Algorithm

```
input: a sorted list of data
input: a value to find
output: index of location of value, or -1

set left to 0, right to listSize-1

while (left <= right):
    set mid to (left + ((right - left) / 2))
    if (item at mid) < (value to find):
        left = mid + 1
    otherwise if (item at mid) > (value to find):
        right = mid - 1
    otherwise:
        return mid

return -1
```

This algorithm is iterative because we use a loop

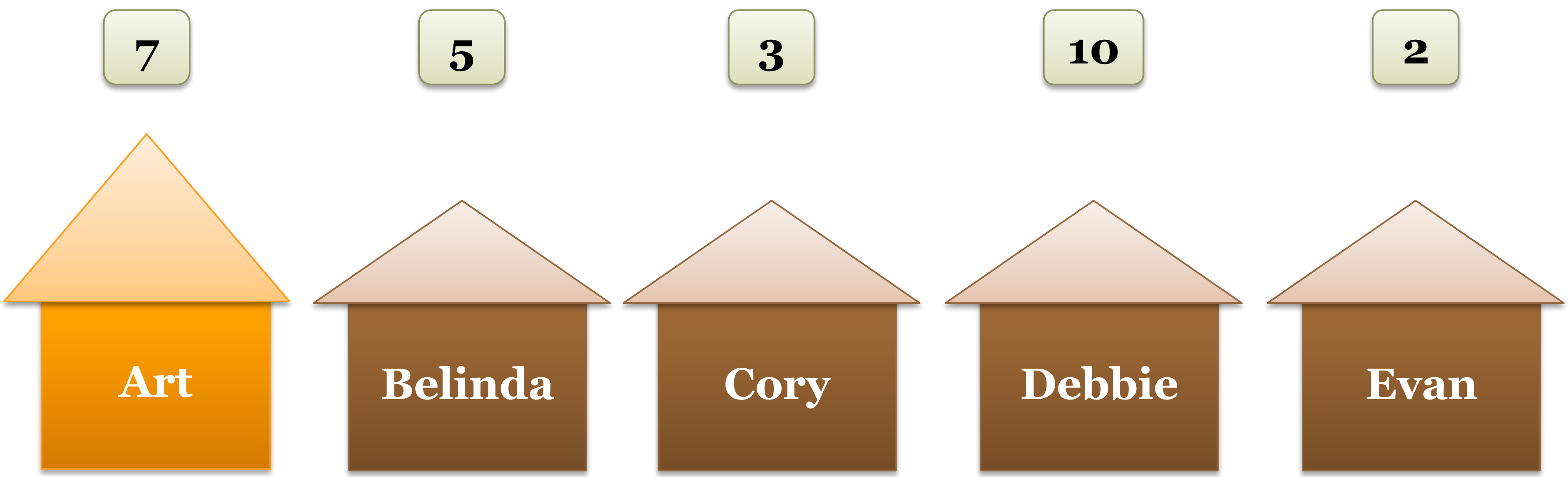
Recursion

Functions calling themselves!

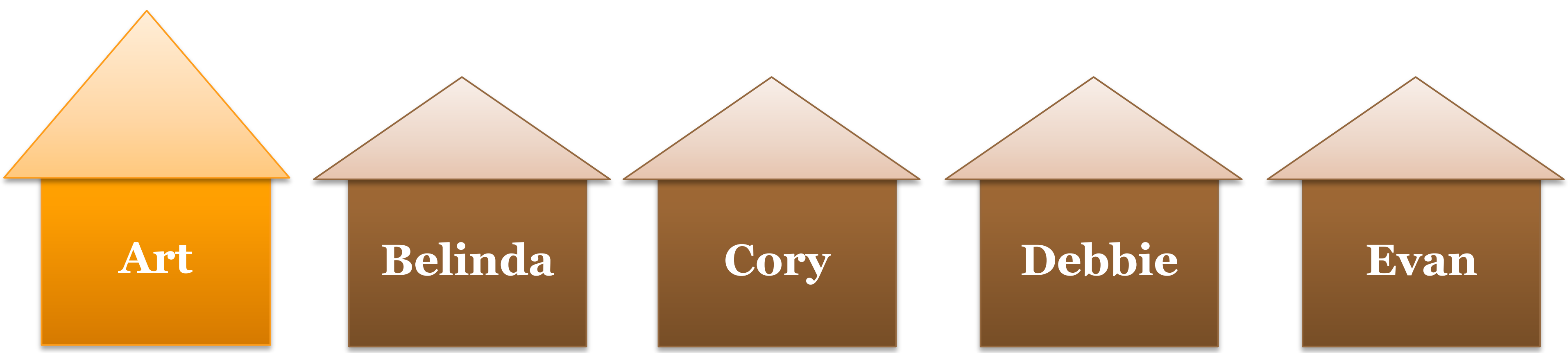
Passengers on the Tropical Paradise Railway (TPR) look forward to seeing dozens of colorful parrots from the train windows. Because of this, the railway takes a keen interest in the health of the local parrot population and decides to take a tally of the number of parrots in view of each train platform along the main line.

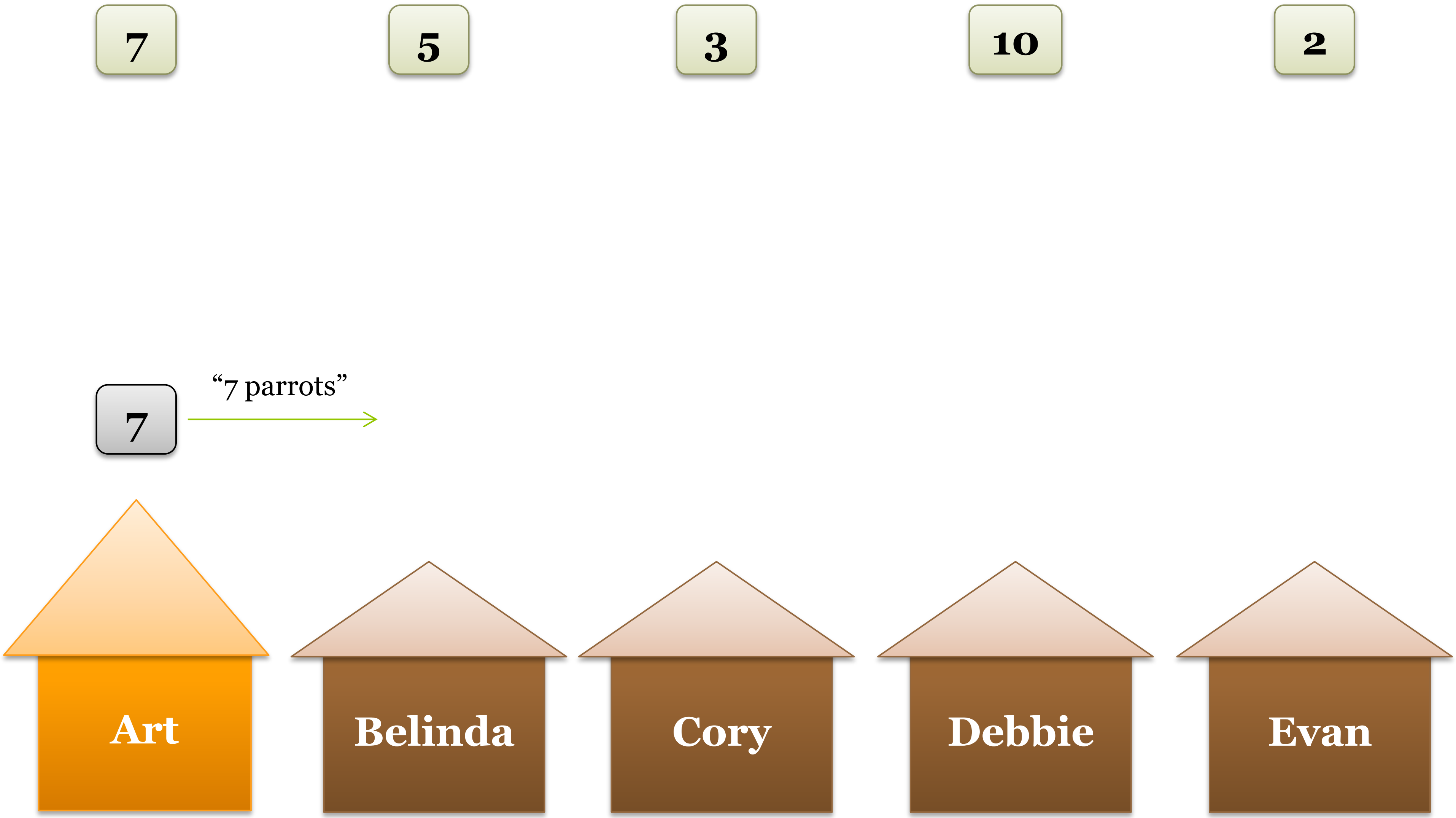
Each platform is staffed by a TPR who is certainly capable of counting parrots. Unfortunately, the job is complicated by the primitive telephone system. Each platform can call only its immediate neighbors.

How do we get the parrot total at the main line terminal?

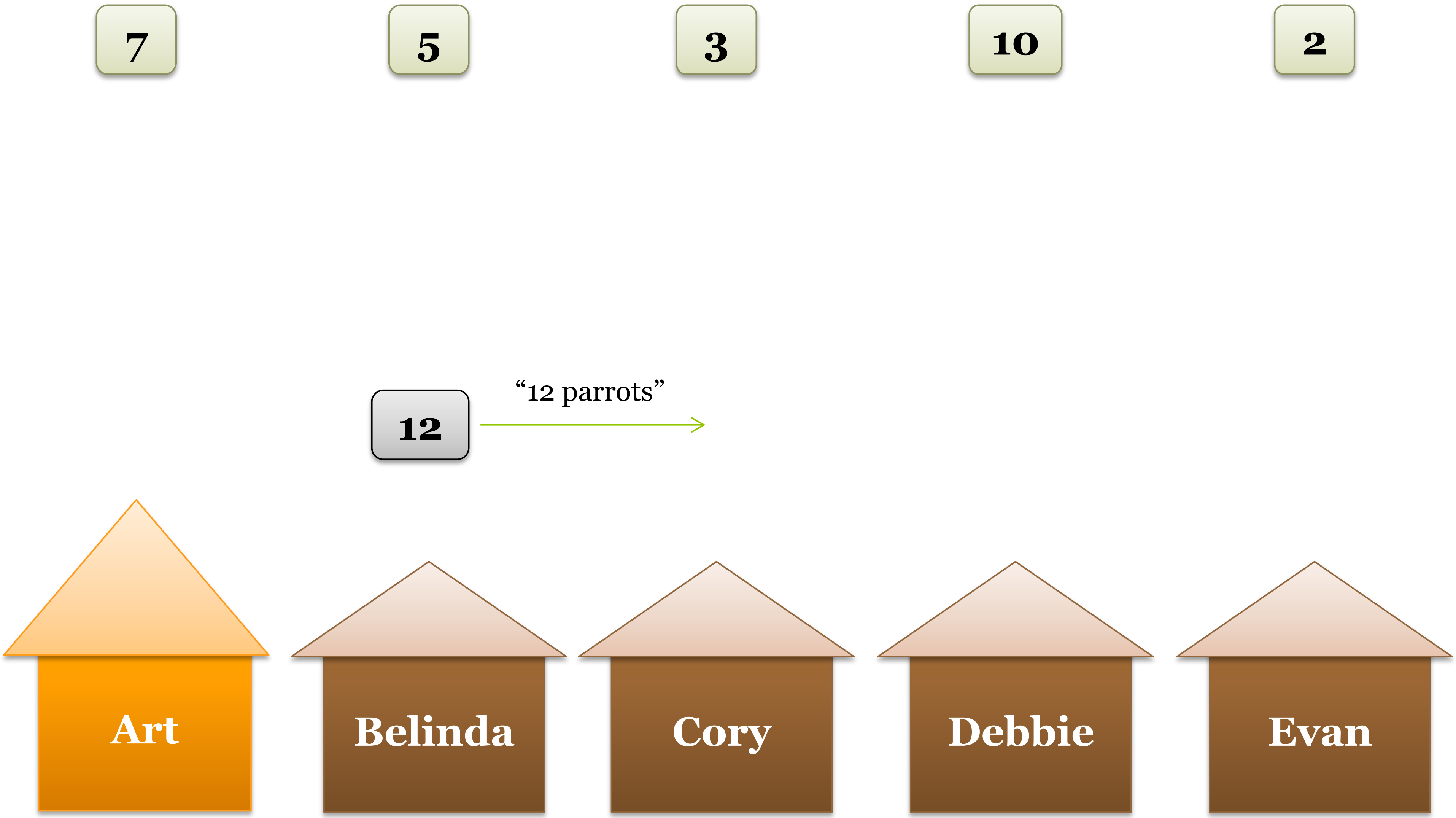

7
5
3
10
2
Art
Belinda
Cory
Debbie
Evan

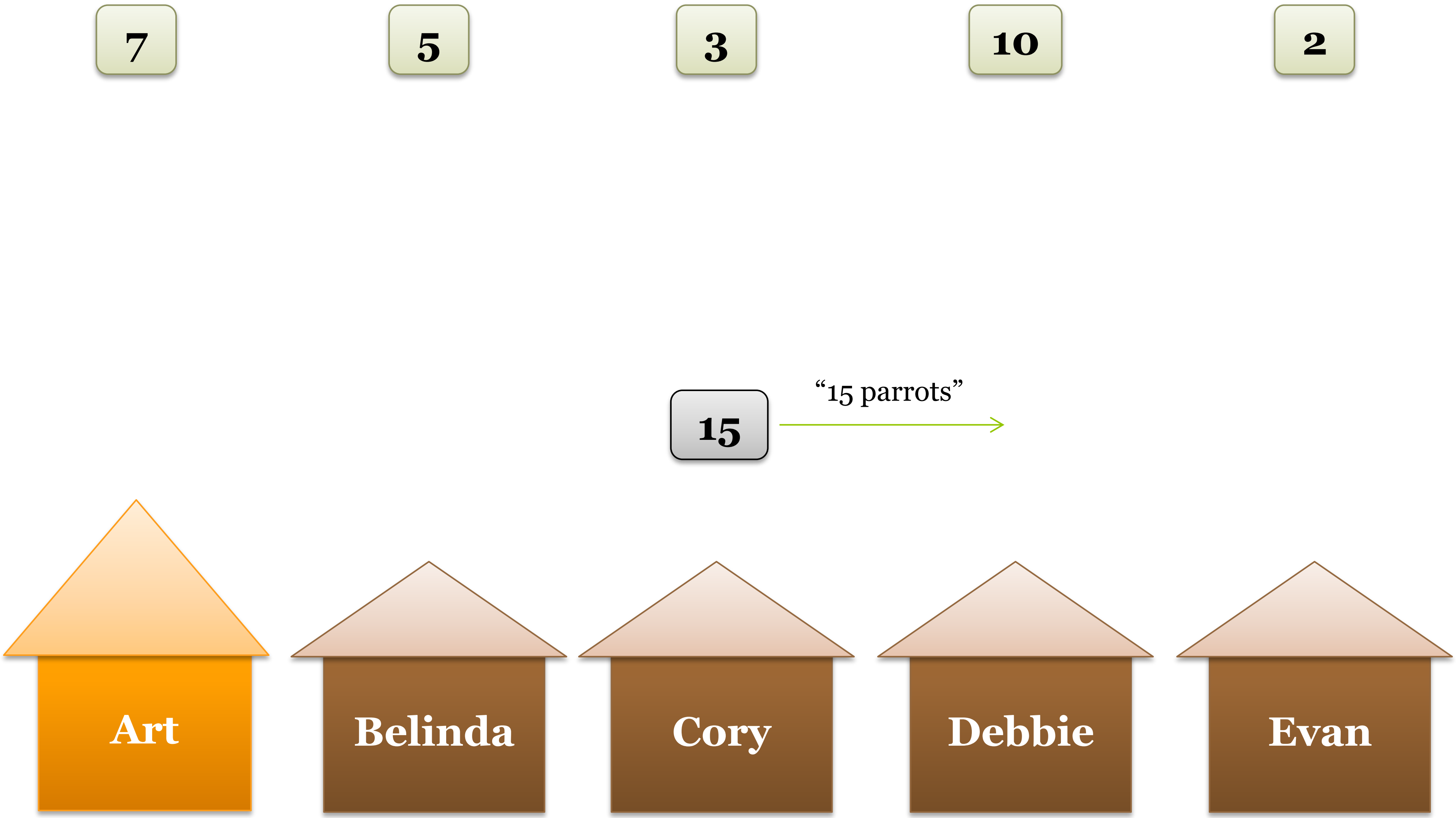
Solution 1: keep a running total of parrots as we go

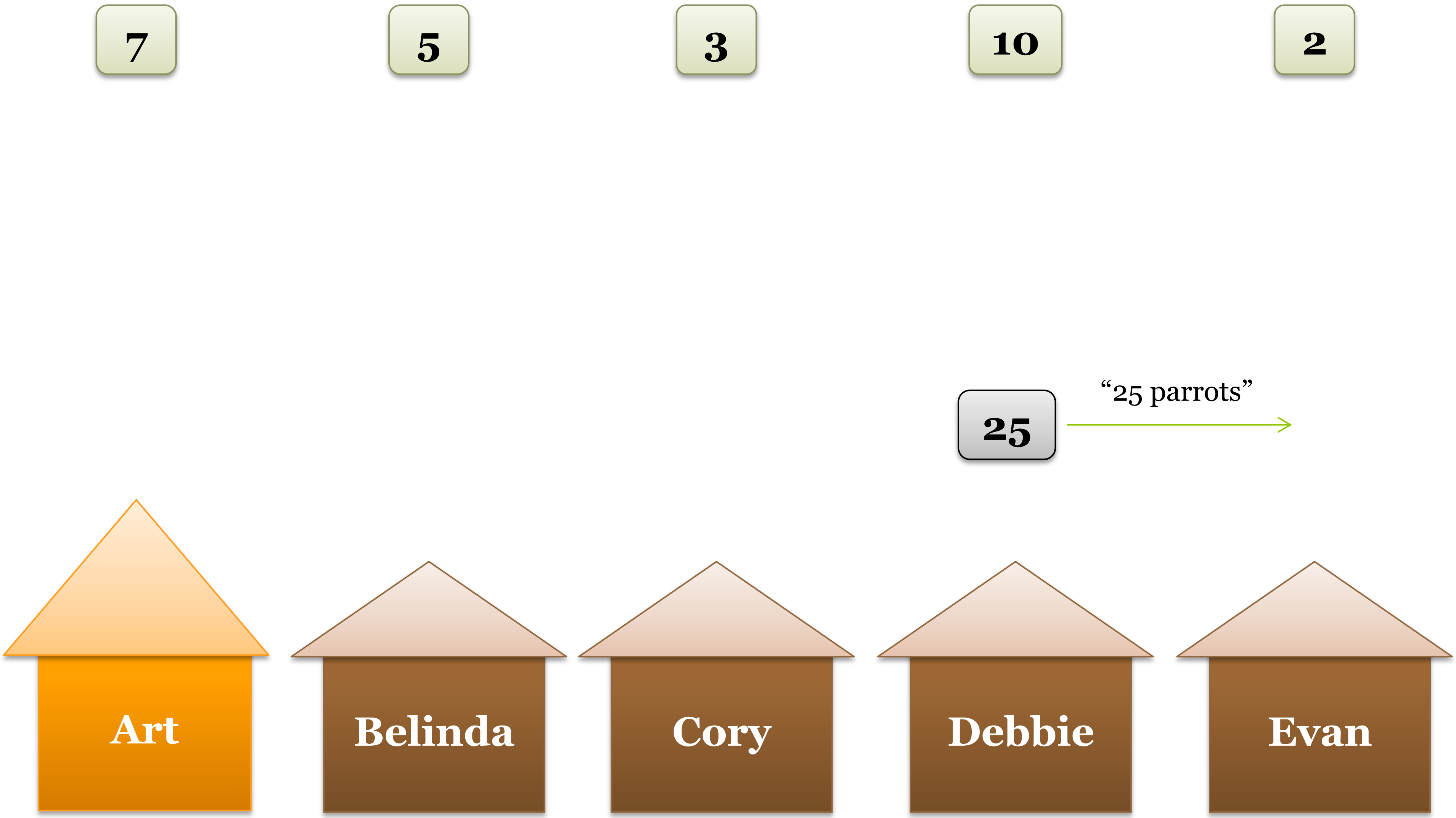


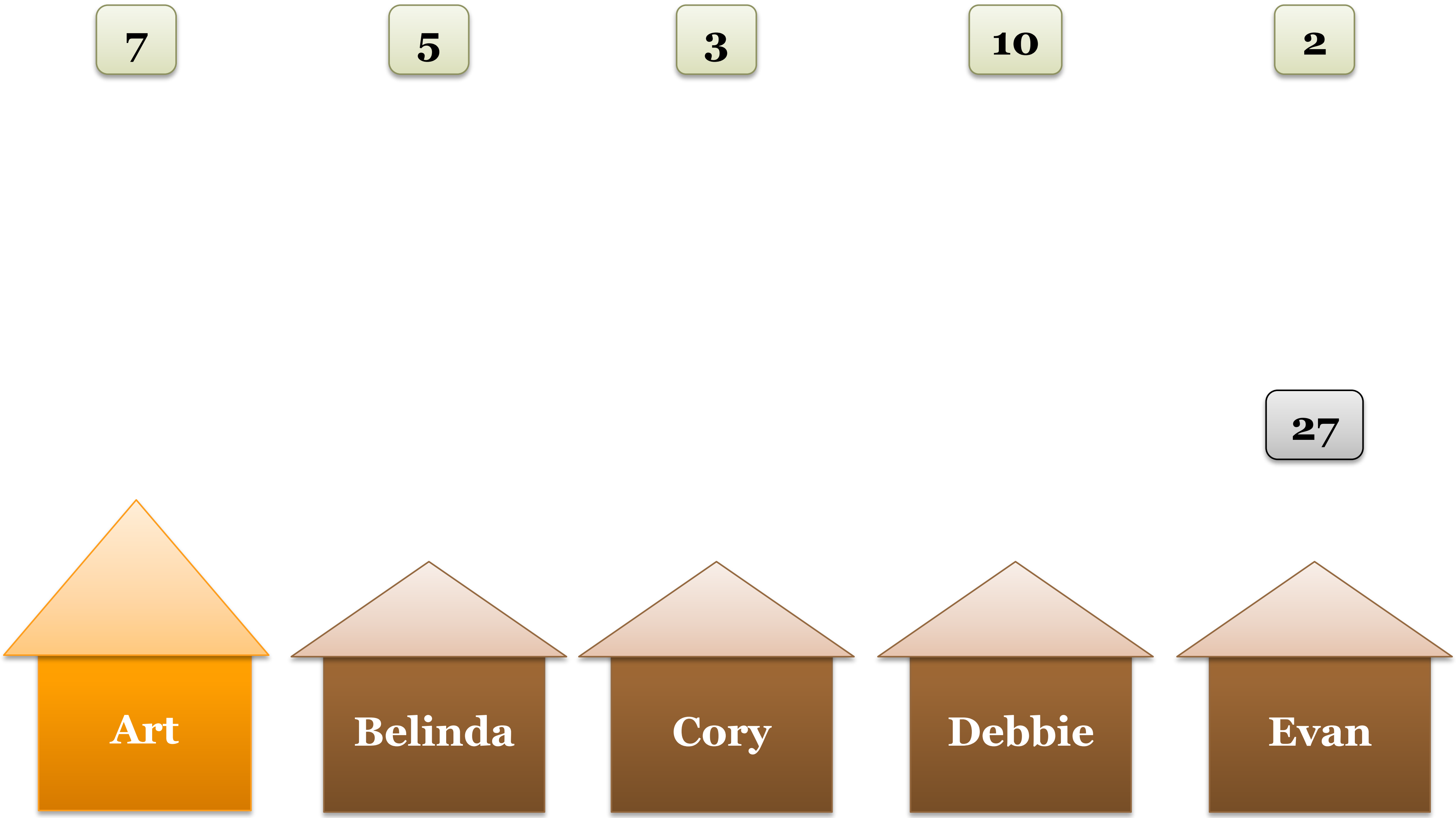


7
5
3
10
2
7
"7 parrots"
Art
Belinda
Cory
Debbie
Evan

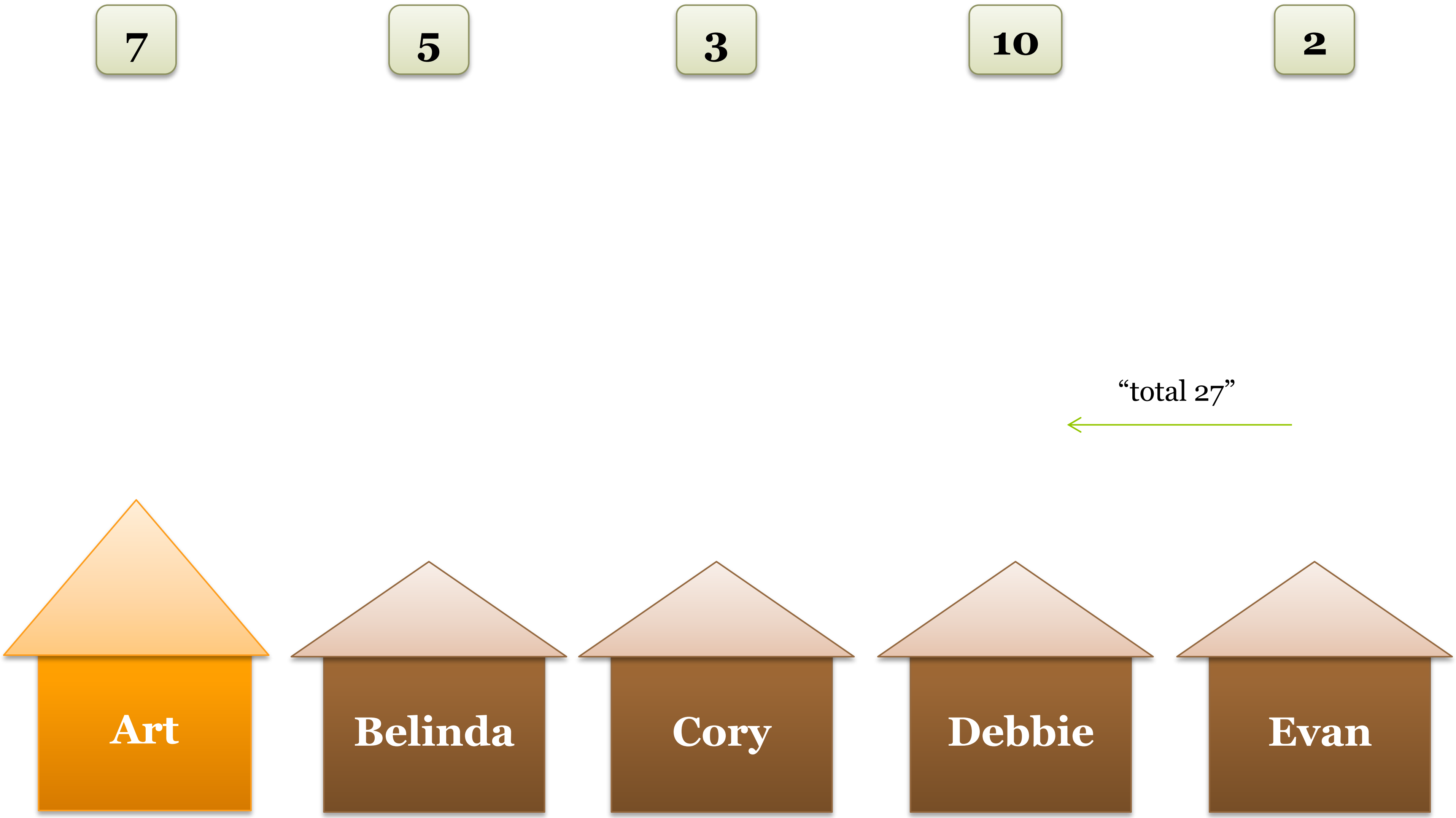

7
5
3
10
2
12
“12 parrots”
Art
Belinda
Cory
Debbie
Evan

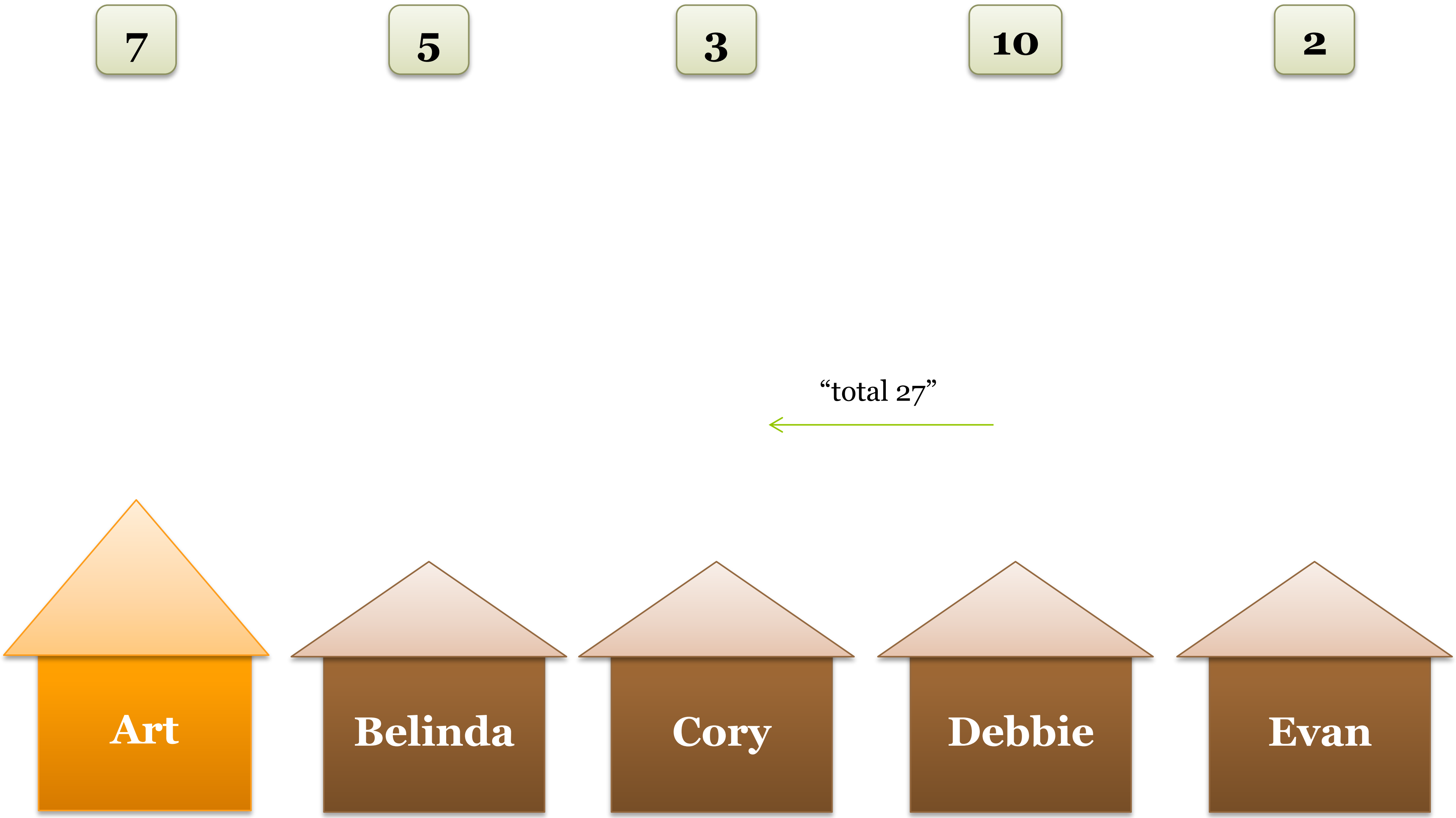

7
5
3
10
2
15
“15 parrots”
Art
Belinda
Cory
Debbie
Evan

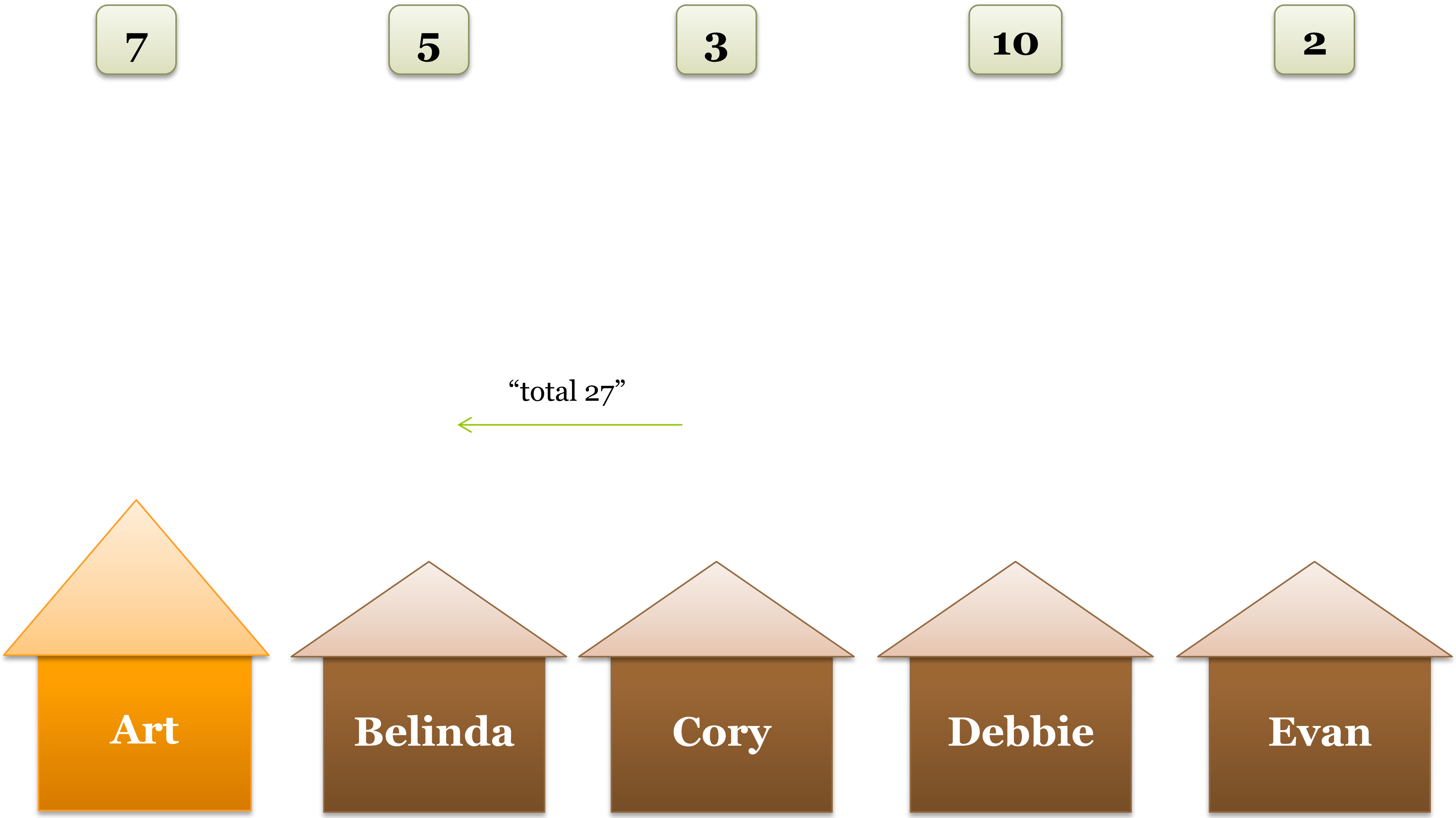

7
5
3
10
2
25
“25 parrots”
Art
Belinda
Cory
Debbie
Evan

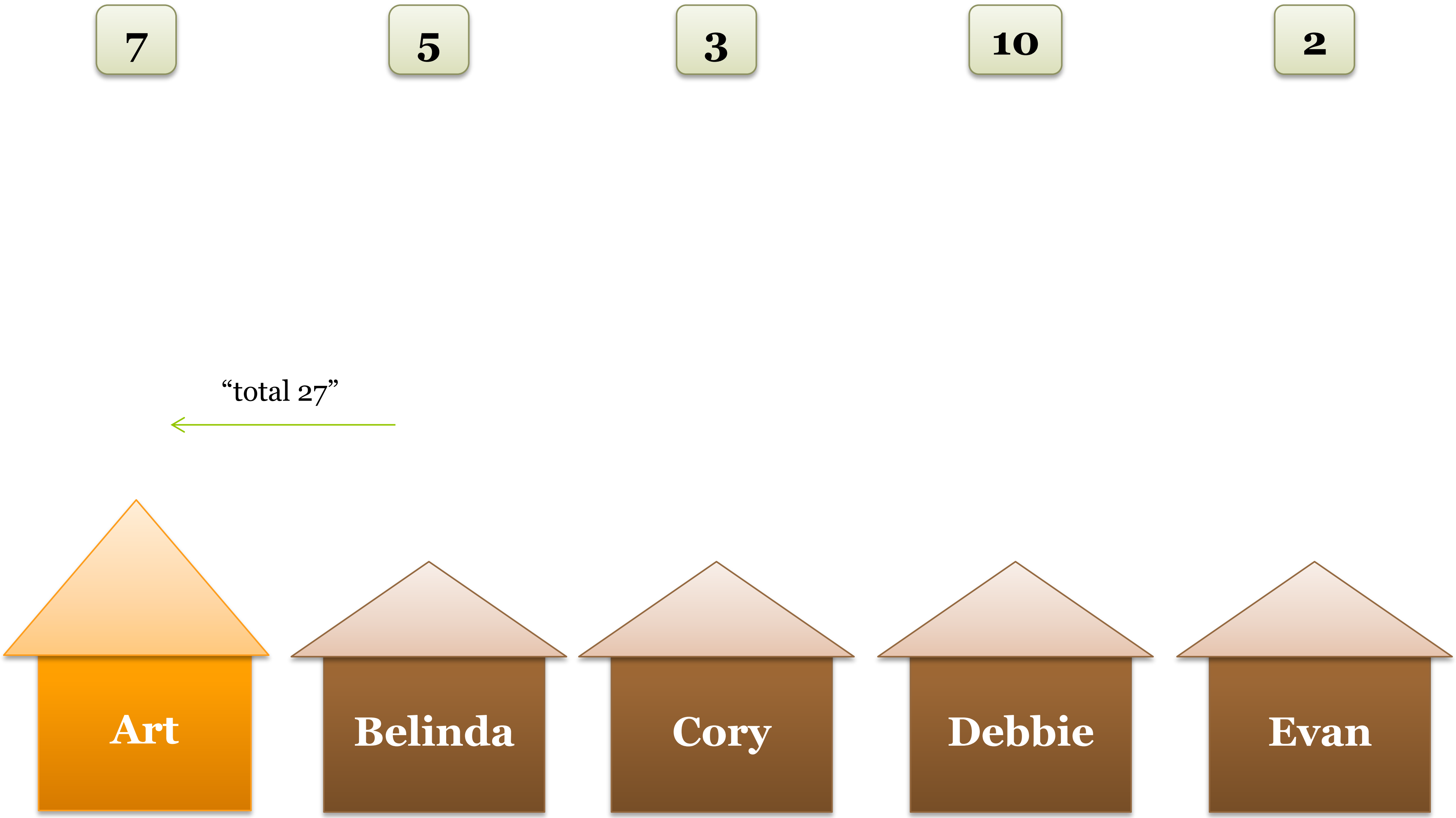


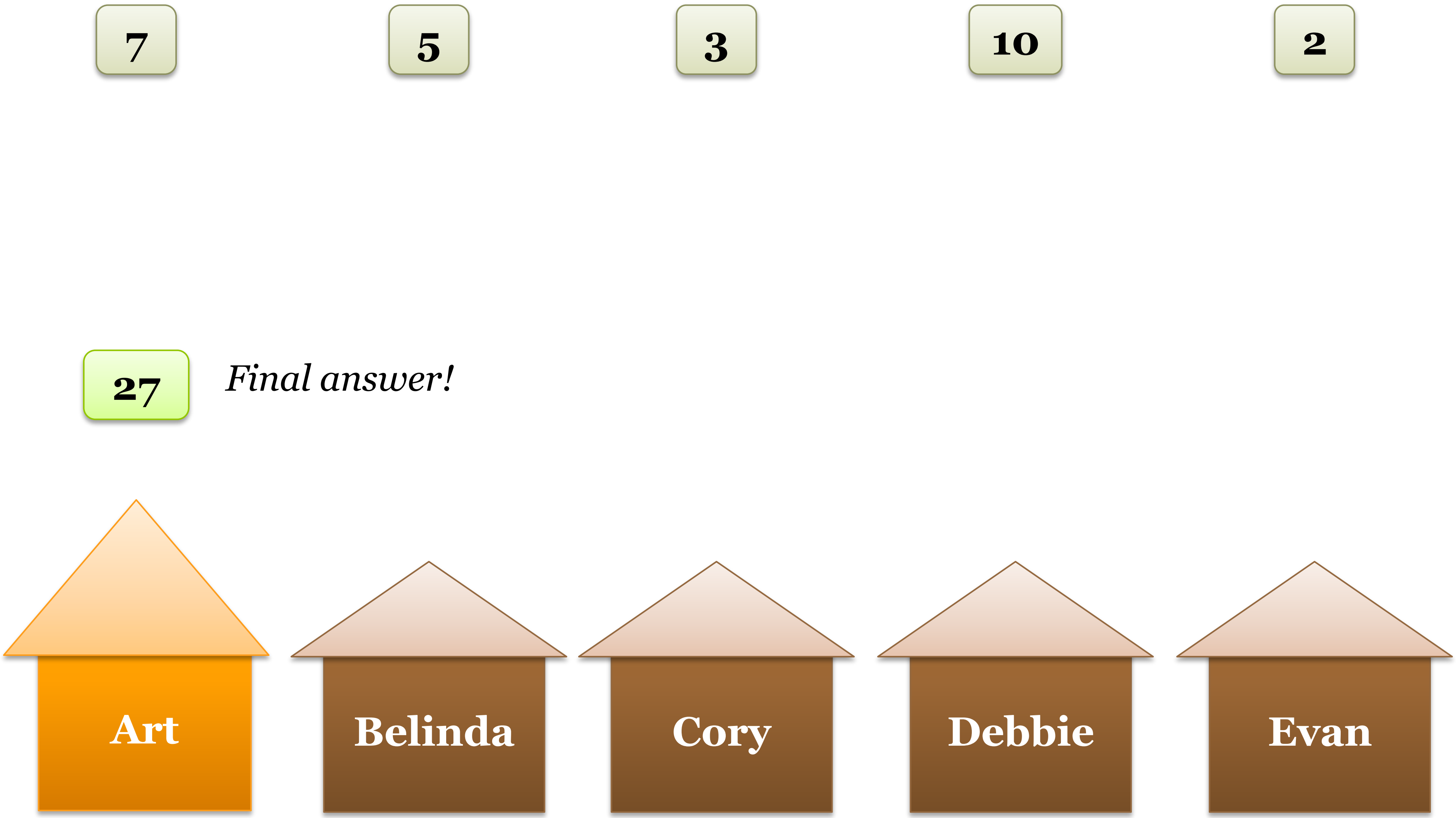
7
5
3
10
2
27
Art
Belinda
Cory
Debbie
Evan


7
5
3
10
2
“total 27”
Art
Belinda
Cory
Debbie
Evan


7
5
3
10
2
“total 27”
Art
Belinda
Cory
Debbie
Evan


7
5
3
10
2
“total 27”
Art
Belinda
Cory
Debbie
Evan

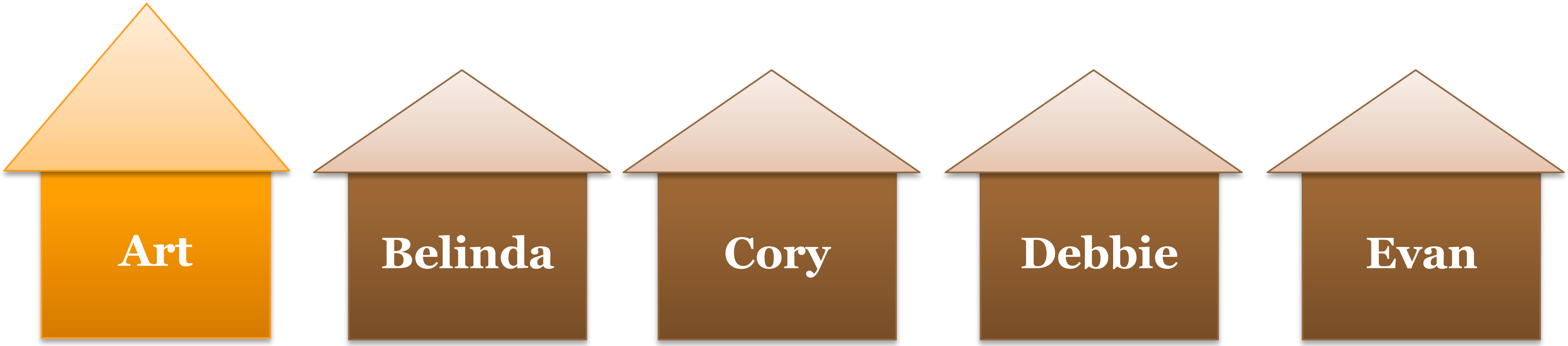

7
5
3
10
2
“total 27”
Art
Belinda
Cory
Debbie
Evan

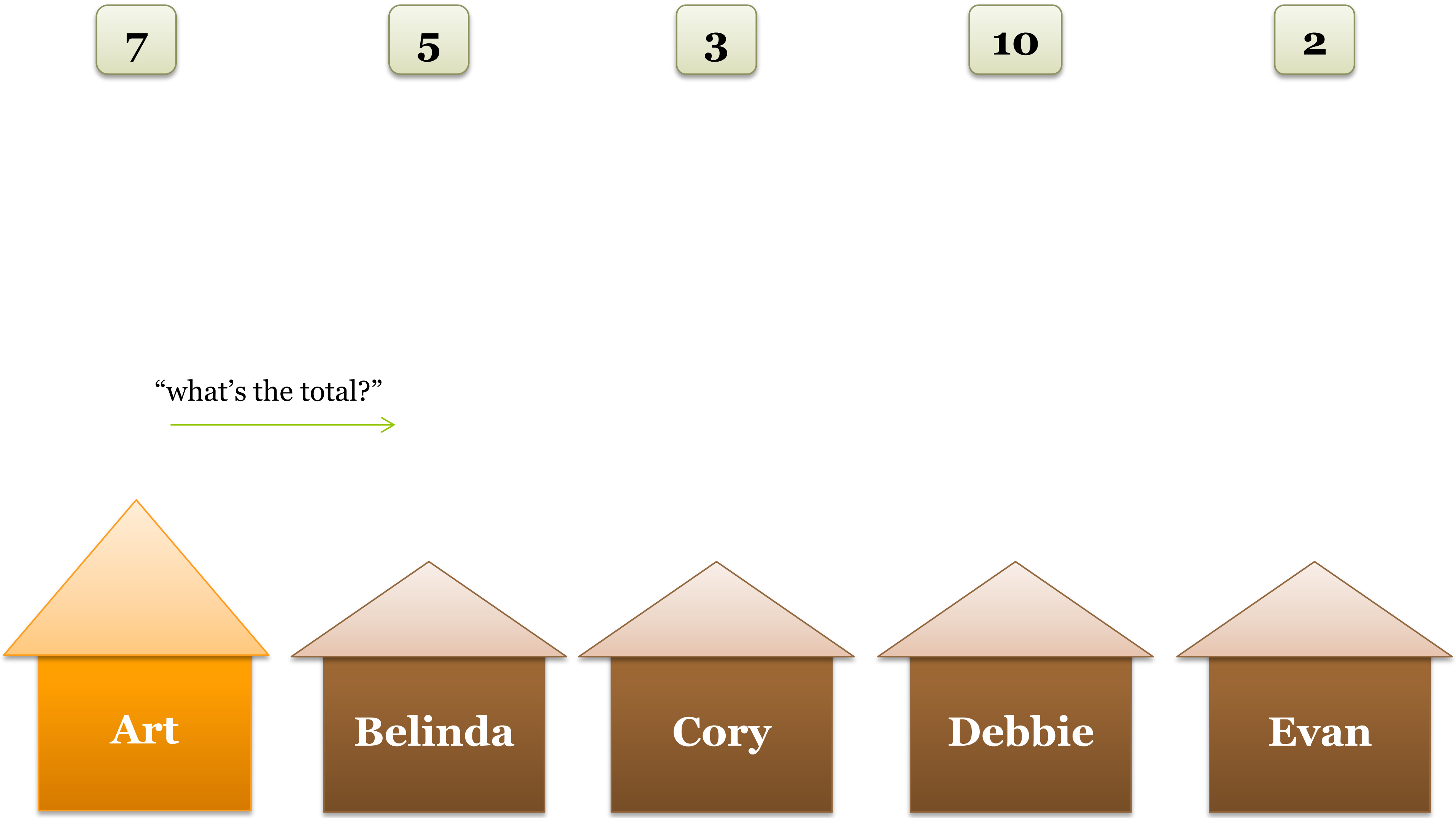

7
5
3
10
2
27
Final answer!
Art
Belinda
Cory
Debbie
Evan

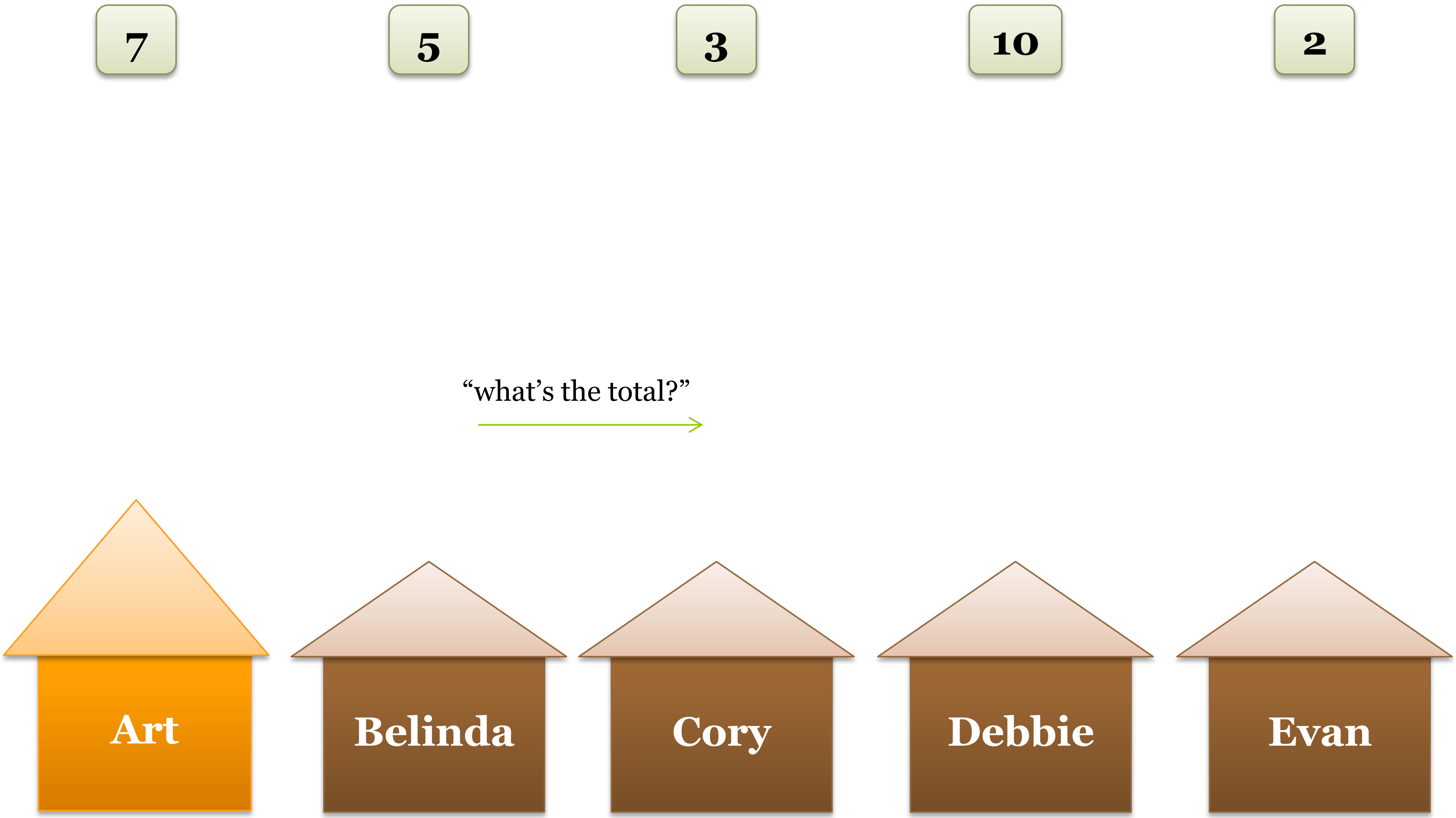
What did we do?

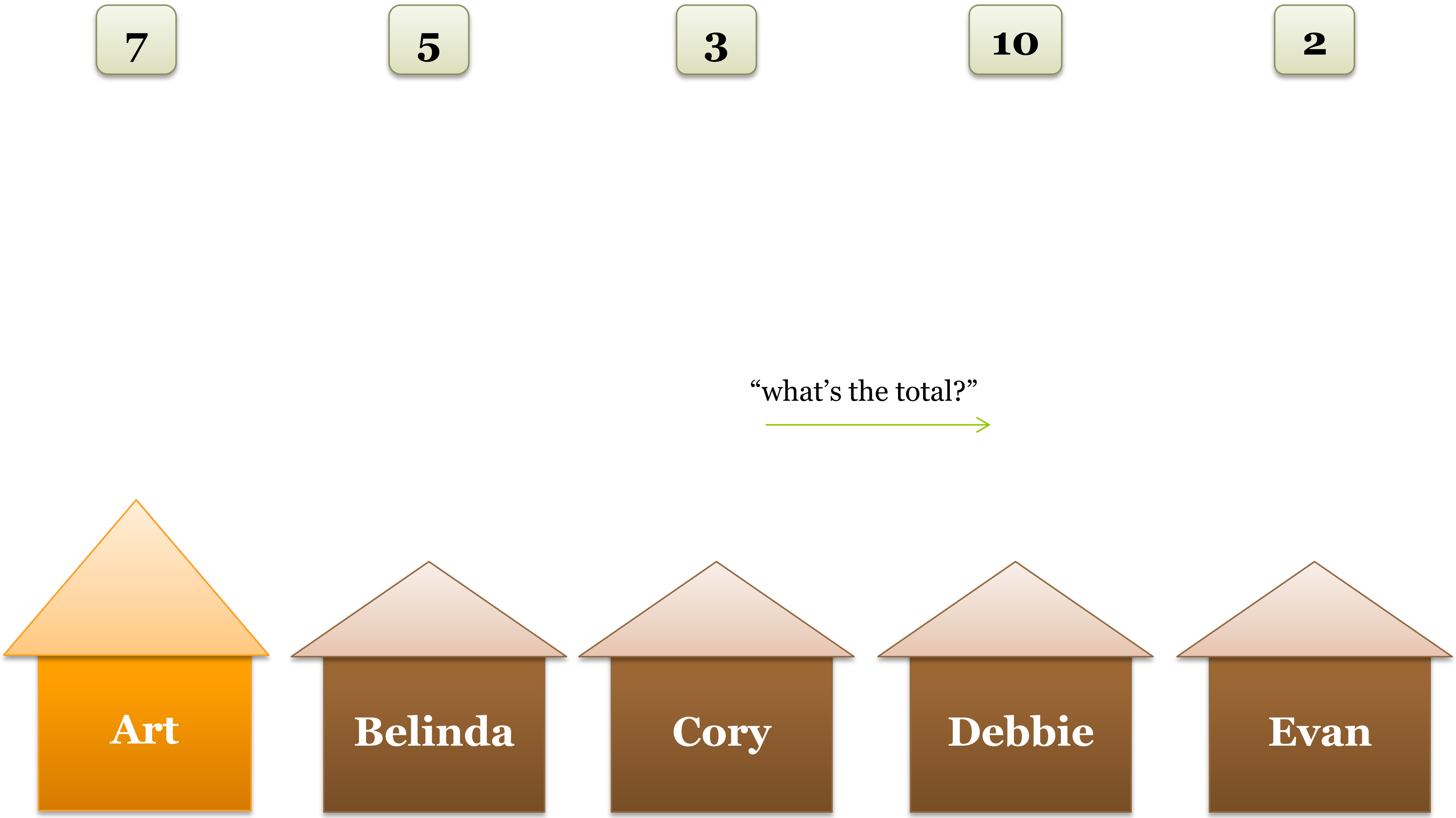
1. Count parrots at platform
2. Add value to the total given by previous station
3. Call the next station with the new total
4. Wait for the next station to return with the total, then pass that along to the previous station

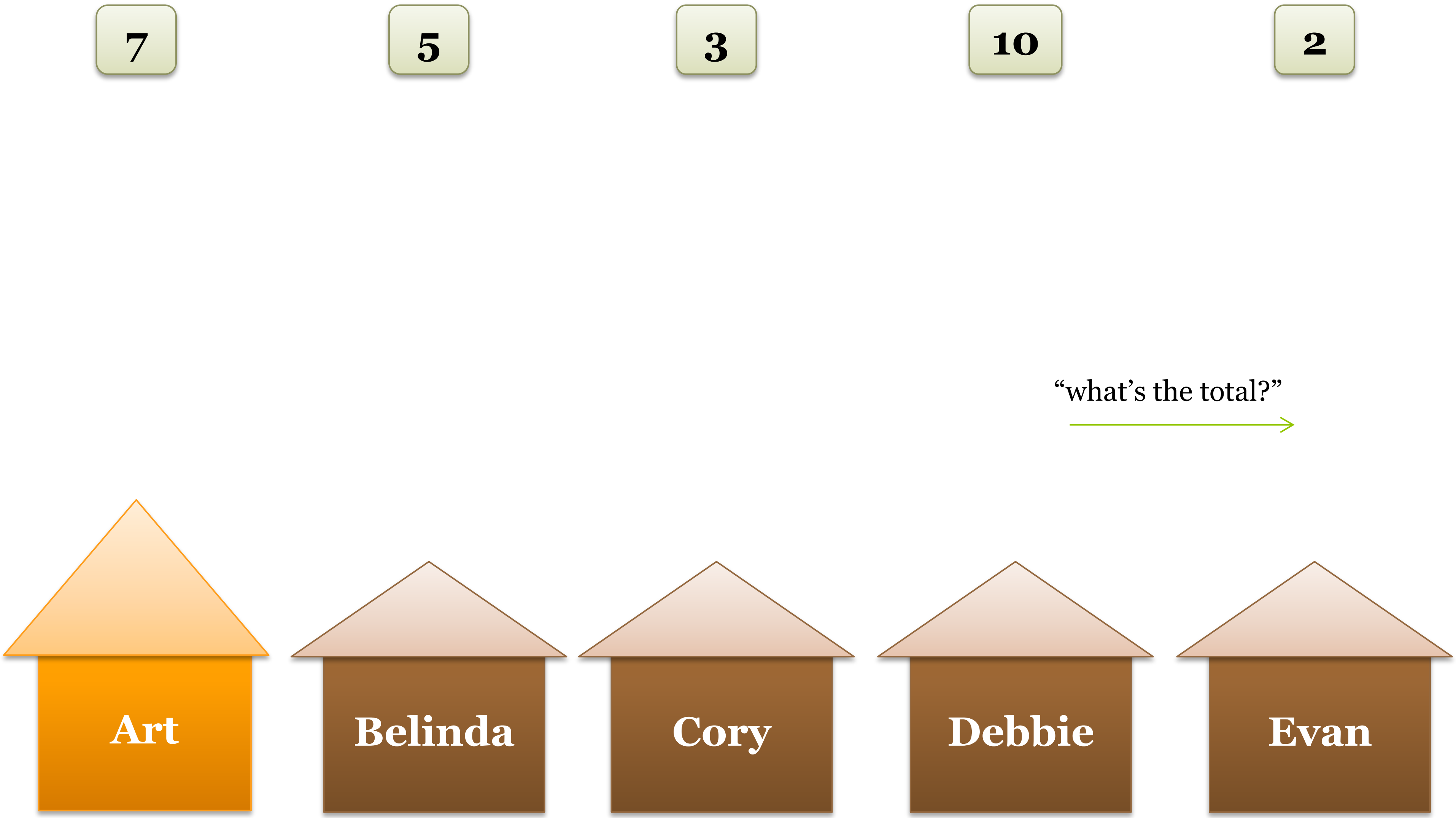
Solution 2: sum the count from the other end

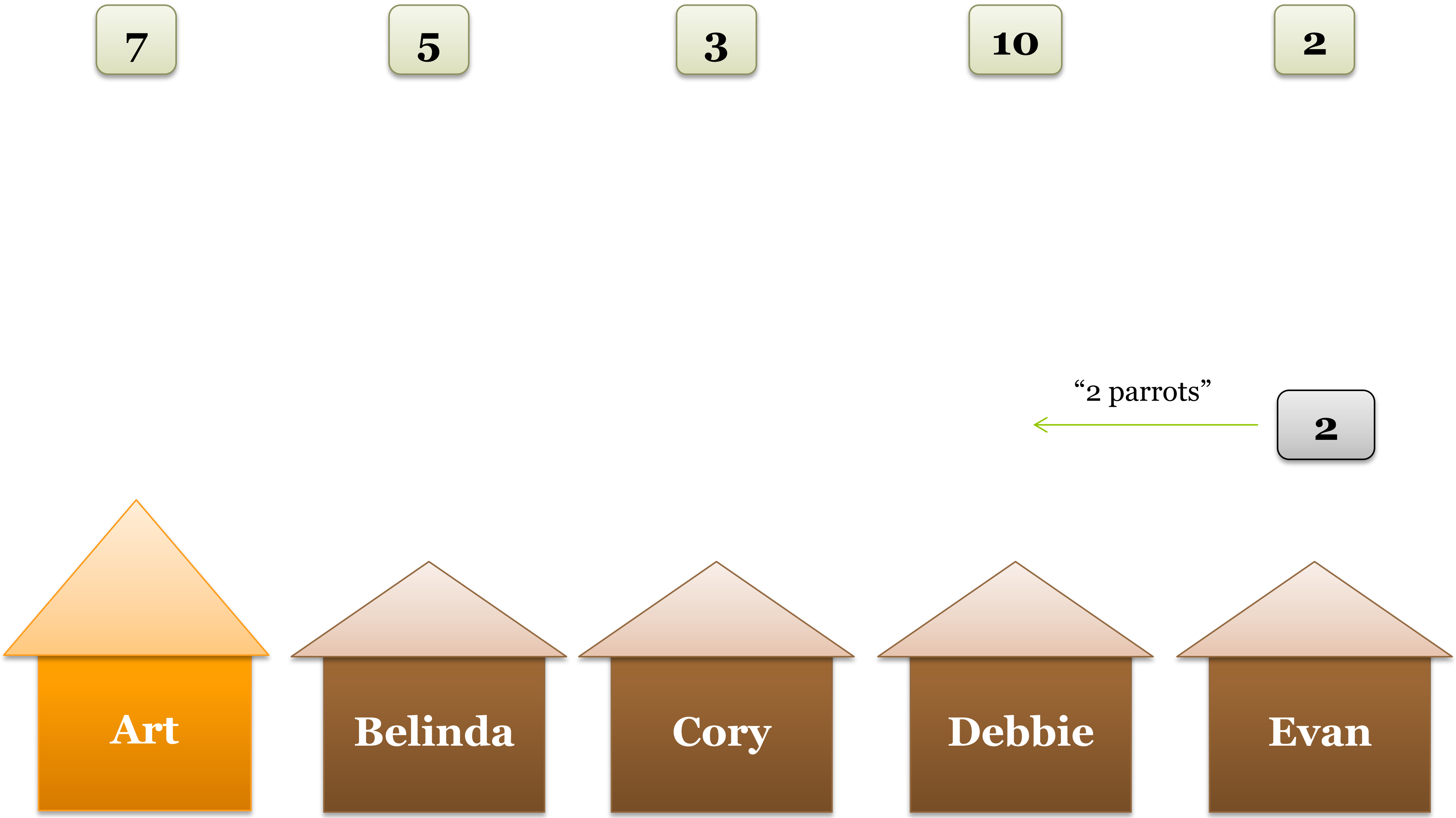


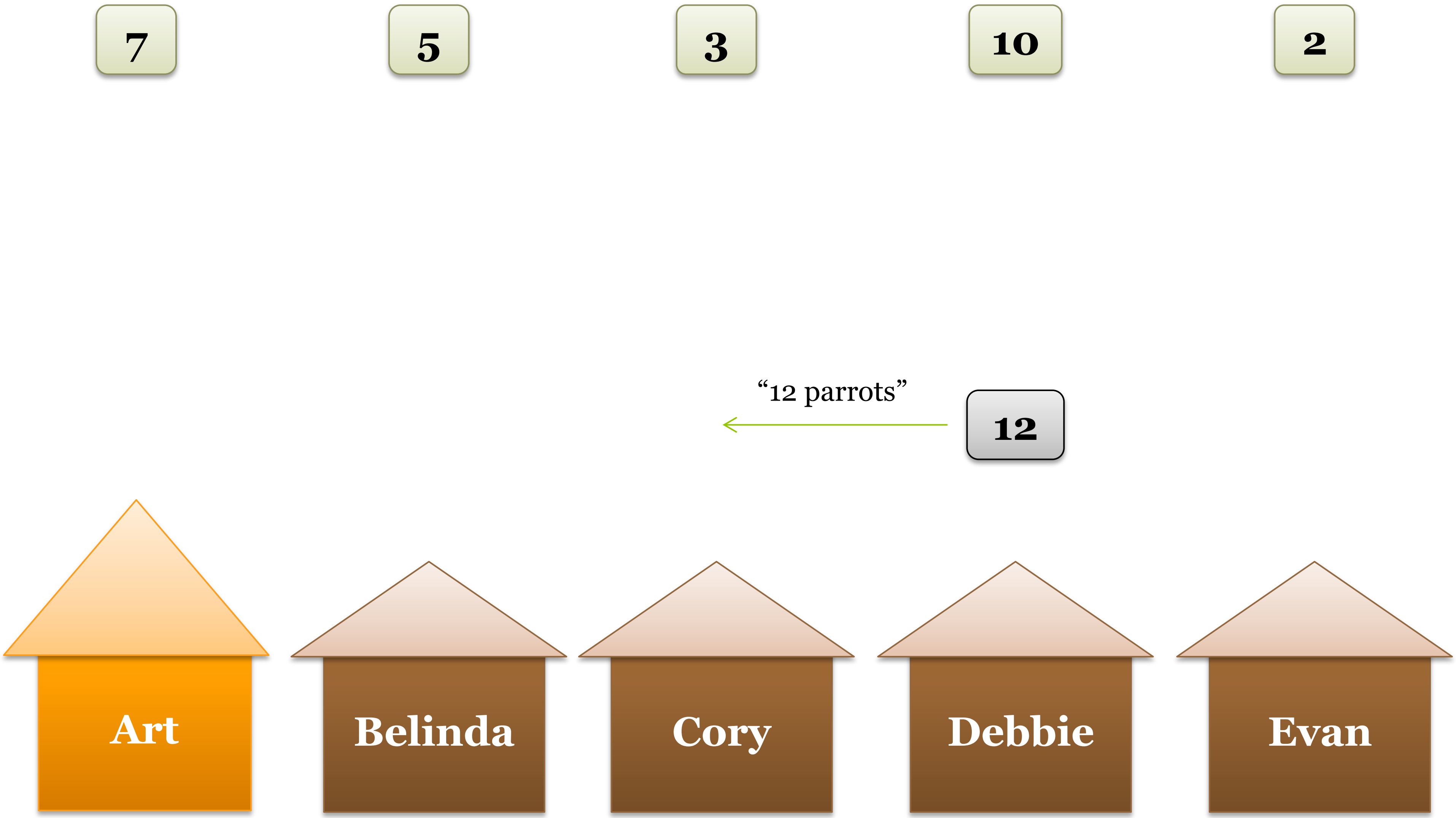

7
5
3
10
2
“what’s the total?”
Art
Belinda
Cory
Debbie
Evan

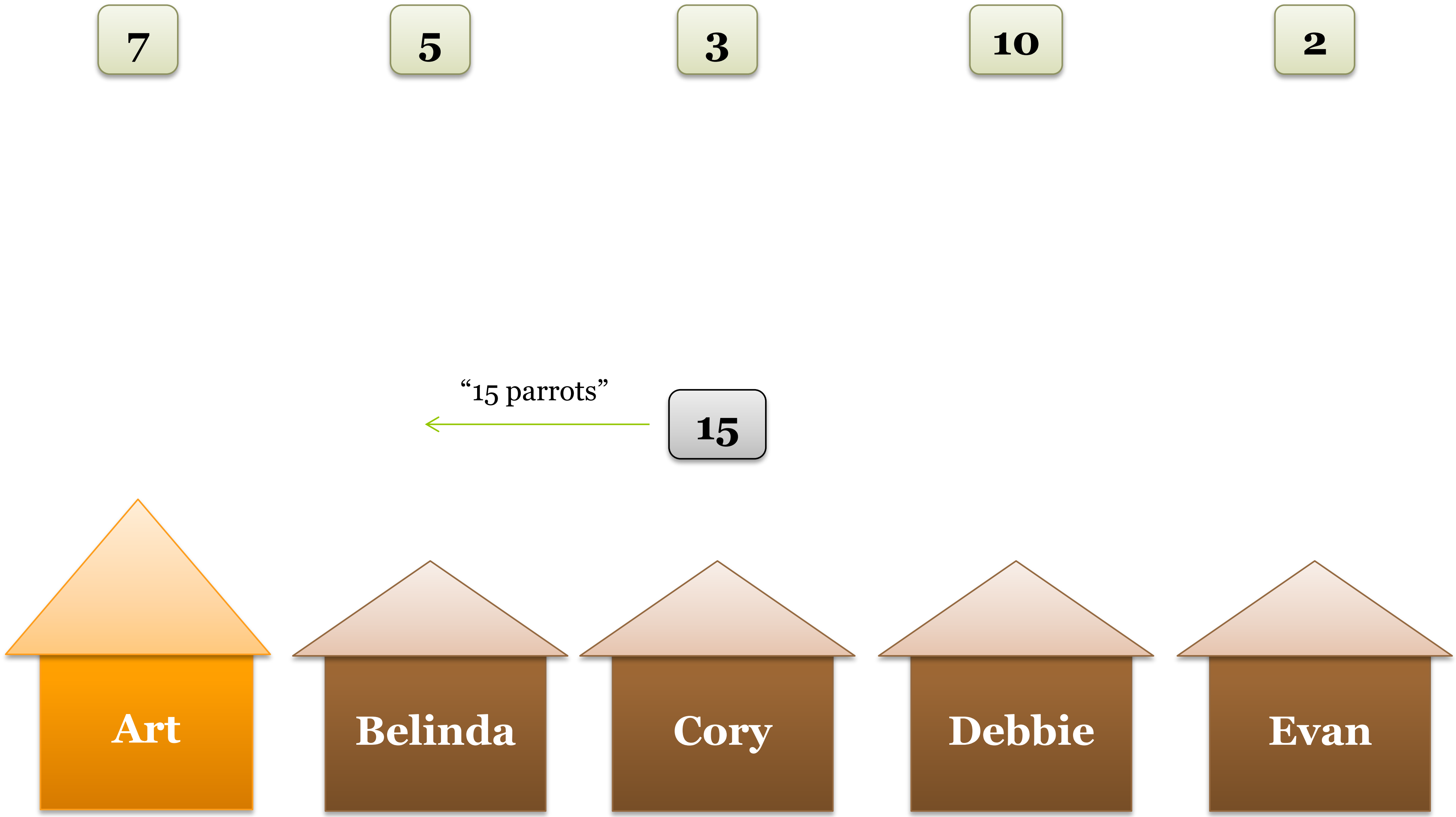

7
5
3
10
2
“what’s the total?”
Art
Belinda
Cory
Debbie
Evan

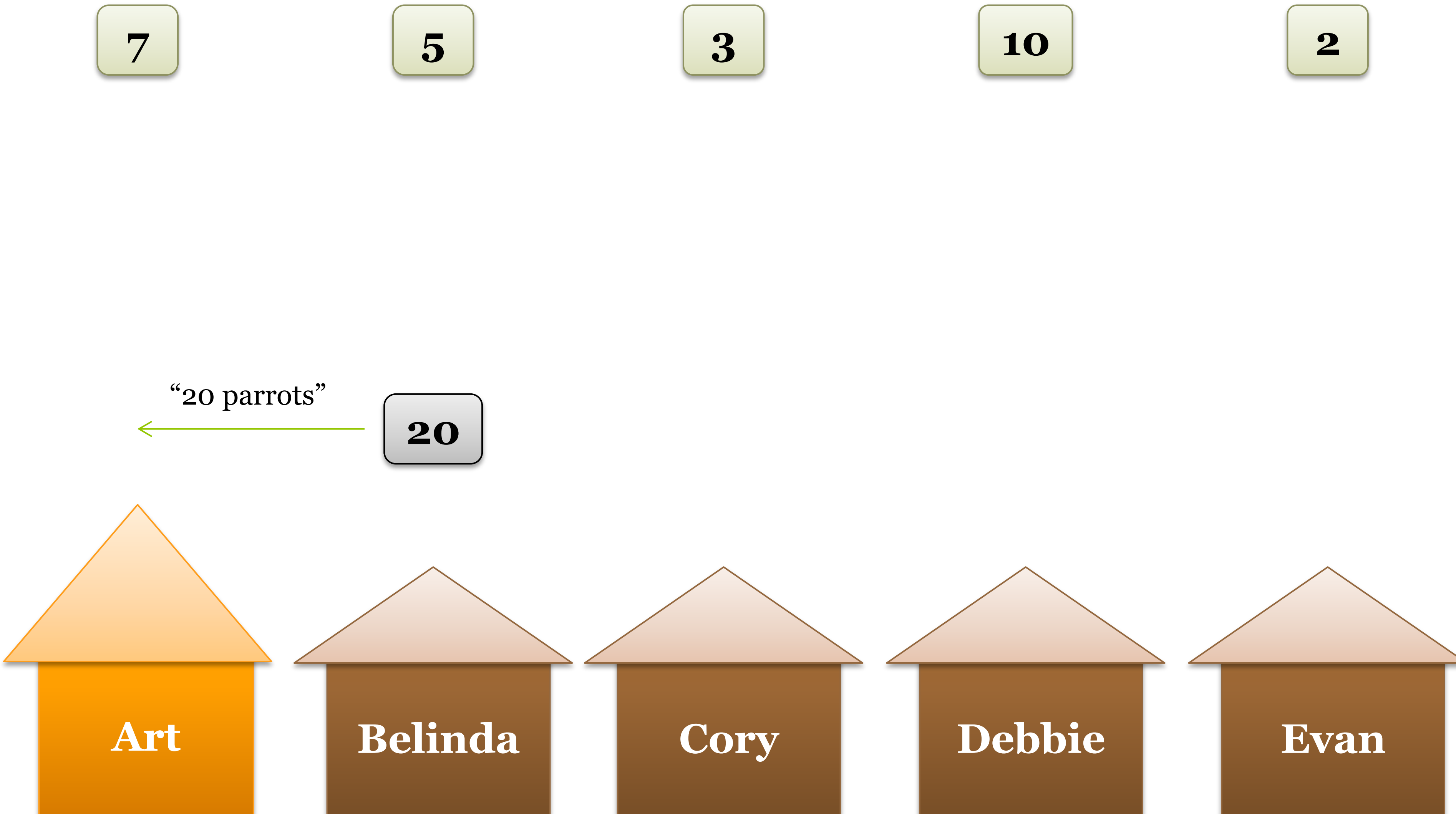

7
5
3
10
2
“what’s the total?”
Art
Belinda
Cory
Debbie
Evan

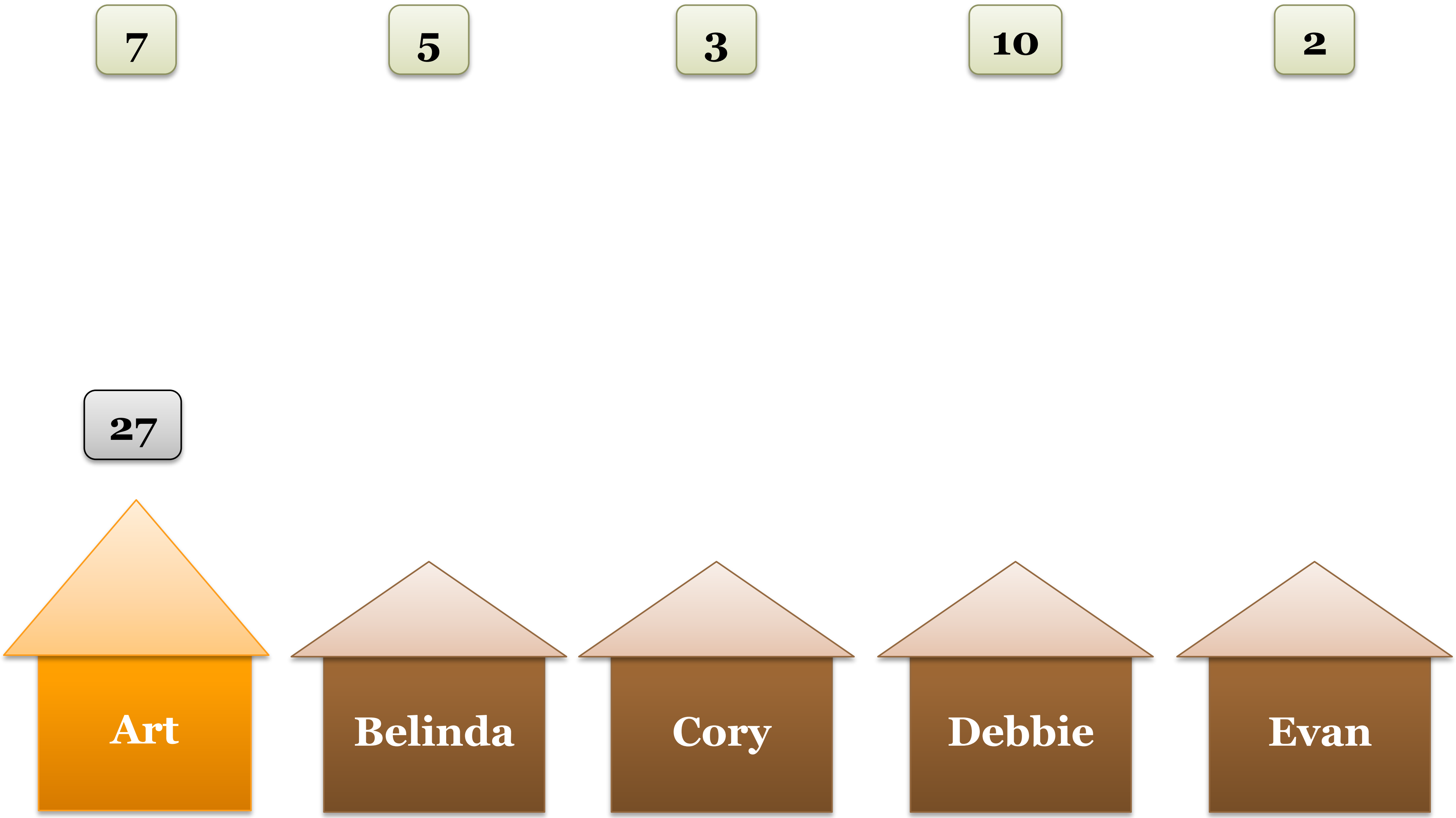

7
5
3
10
2
“what’s the total?”
Art
Belinda
Cory
Debbie
Evan


7
5
3
10
2
"2 parrots"
2
Art
Belinda
Cory
Debbie
Evan


7
5
3
10
2
“12 parrots”
12
Art
Belinda
Cory
Debbie
Evan


7
5
3
10
2
“15 parrots”
15
Art
Belinda
Cory
Debbie
Evan


7
5
3
10
2
“20 parrots”
20
Art
Belinda
Cory
Debbie
Evan


7
5
3
10
2
27
Art
Belinda
Cory
Debbie
Evan

7 5 3 10 2

27 *Final answer!*

Art Belinda Cory Debbie Evan

What did we do?

1. Call next station.
2. Count the parrots at platform.
3. Add this count to the total given by the next station.
4. Pass resulting sum up to previous station.

Solution 1

1. Count parrots at platform
2. Add value to the total given by previous station
3. Call the next station with the new total
4. Wait for the next station to return with the total, then pass that along to the previous station

Solution 2

1. Call next station.
2. Count the parrots at platform.
3. Add this count to the total given by the next station.
4. Pass resulting sum up to previous station.

Tail Recursion

Solution 1

1. Count parrots at platform
2. Add value to the total given by previous station
3. **Call the next station with the new total**
4. Wait for the next station to return with the total, then pass that along to the previous station

Solution 2

1. Call next station.
2. Count the parrots at platform.
3. Add this count to the total given by the next station.
4. Pass resulting sum up to previous station.

Tail Recursion

Solution 1

1. Count parrots at platform
2. Add value to the total given by previous station
3. Call the next station with the new total
4. Wait for the next station to return with the total, then pass that along to the previous station

Head Recursion

Solution 2

1. **Call next station**.
2. Count the parrots at platform.
3. Add this count to the total given by the next station.
4. Pass resulting sum up to previous station.

In each approach, who knows the total number of parrots (the final answer)?

What information is being passed to the recursive call in each approach?

Solution 1

1. Count parrots at platform
2. Add value to the total given by previous station
3. Call the next station with the new total
4. Wait for the next station to return with the total, then pass that along to the previous station

Solution 2

1. Call next station.
2. Count the parrots at platform.
3. Add this count to the total given by the next station.
4. Pass resulting sum up to previous station.

We will apply this style of recursion to binary searching…

Solution 1

1. Count parrots at platform
2. Add value to the total given by previous station
3. Call the next station with the new total
4. Wait for the next station to return with the total, then pass that along to the previous station

Solution 2

1. Call next station.
2. Count the parrots at platform.
3. Add this count to the total given by the next station.
4. Pass resulting sum up to previous station.

We'll check whether we've found the value, and recalculate the list bounds first…

1. Count parrots at platform
2. Add value to the total given by previous station
3. Call the next station with the new total
4. Wait for the next station to return with the total, then pass that along to the previous station

Solution 2

1. Call next station.
2. Count the parrots at platform.
3. Add this count to the total given by the next station.
4. Pass resulting sum up to previous station.

...then we'll recurse, passing along the information we've gathered...

3. Call the next station with the new total
4. Wait for the next station to return with the total, then pass that along to the previous station

Solution 2

1. Call next station.
2. Count the parrots at platform.
3. Add this count to the total given by the next station.
4. Pass resulting sum up to previous station.

Solution 1

...and finally we'll wait for an answer to come back, which we'll just return to whoever called us

4. Wait for the next station to return with the total, then pass that along to the previous station

Solution 2

1. Call next station.
2. Count the parrots at platform.
3. Add this count to the total given by the next station.
4. Pass resulting sum up to previous station.

Recursive Binary Search Algorithm

```
input: a sorted list of data
input: a value to find
input: minIndex of list
input: maxIndex of list
output: index of location of value, or -1

if (maxIndex < minIndex):
   return -1
else:
   set mid to (minIndex + ((maxIndex - minIndex) / 2))
   if (item at mid) < (value to find):
      return binarySearch(list, value to find,
                          mid+1, maxIndex)
   else if (item at mid) > (value to find):
      return binarySearch(list, value to find,
                          minIndex, midIndex-1)
   else:
      return mid
```

General Form of Recursion

1. Check if we're finished ("base case").
2. If not, break the problem into something smaller, and call the function again.

Gene rsion

For parrot counting, if we've reached the main terminal, we're done.

1. Check if we're finished ("base case").
2. If not, break the problem into something smaller, and call the function again.

General Form of Recursion

1. Check if we're finished ("base case").
2. If not, break the problem into something smaller, and call the function again.

Otherwise, find out the count of the terminals after the current terminal

Gen sion

For binary search, if we found the value or searched the entire list, we're done.

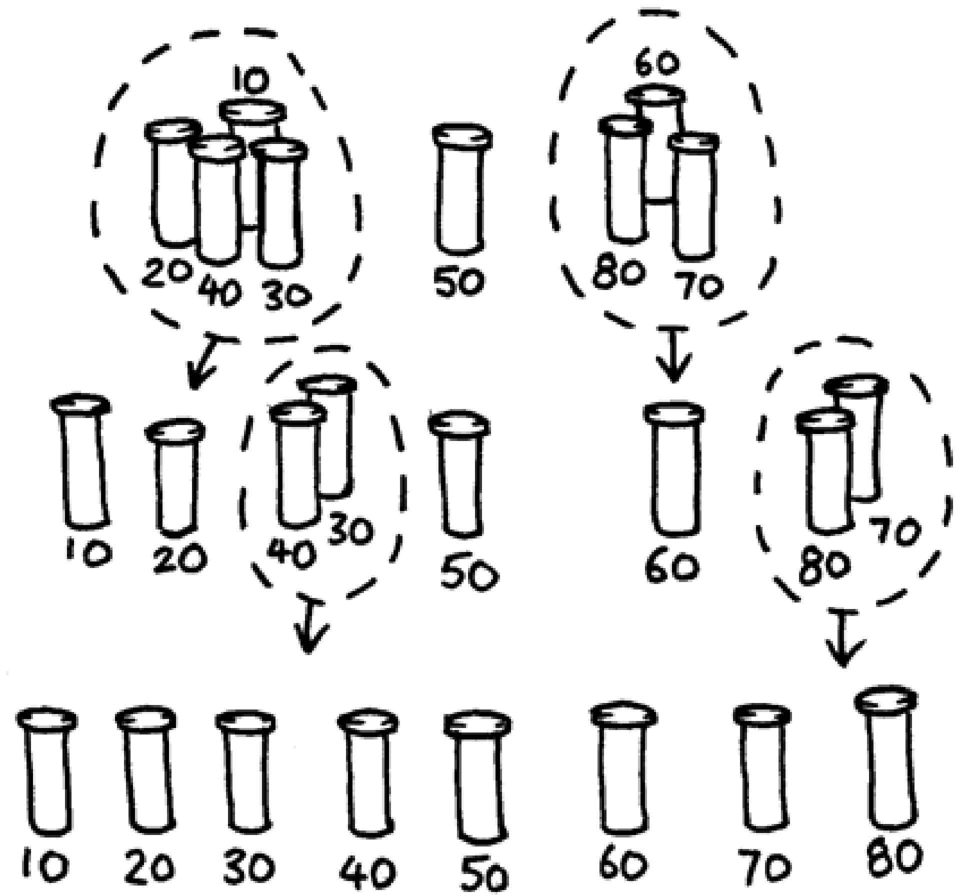
1. Check if we're finished ("base case").
2. If not, break the problem into something smaller, and call the function again.

General Form of Recursion

1. Check if we're finished ("base case").
2. If not, break the problem into something smaller, and call the function again.

Otherwise, keep searching in only the bottom or top half of the list, whichever the value can actually be in

Quick Sort



Quick Sort

http://www.sorting-algorithms.com/quick-sort

https://www.youtube.com/watch?v=cVMKXKoGu_Y

http://www.youtube.com/watch?v=FSyr8o8jjwM

Partition Algorithm

```
input: a list of data
input: leftIndex, rightIndex
output: index of pivot

set pivotIndex to random index between left and right
set pivotValue to value at pivotIndex

swap values at pivotIndex and rightIndex
set partitionIndex to leftIndex

for each value between leftIndex and rightIndex-1:
   if next list value < pivotValue:
      swap next list value and value at partitionIndex
      add 1 to partitionIndex

swap values at rightIndex and partitionIndex
return partitionIndex
```

Partition Algorithm

Choose a random pivot value to sort list items around

```
input: a list of data
input: leftIndex, rightInde
output: index of pivot

set pivotIndex to random index between left and right
set pivotValue to value at pivotIndex

swap values at pivotIndex and rightIndex
set partitionIndex to leftIndex

for each value between leftIndex and rightIndex-1:
   if next list value < pivotValue:
      swap next list value and value at partitionIndex
      add 1 to partitionIndex

swap values at rightIndex and partitionIndex
return partitionIndex
```

Partition Algorithm

```
input: a list of data
input: leftIndex, ri
output: index of piv

set pivotIndex to ra
set pivotValue to va

swap values at pivotIndex and rightIndex
set partitionIndex to leftIndex

for each value between leftIndex and rightIndex-1:
   if next list value < pivotValue:
      swap next list value and value at partitionIndex
      add 1 to partitionIndex

swap values at rightIndex and partitionIndex
return partitionIndex
```

Temporarily put the pivot to the far right, since we don't know where it will be positioned after partitioning

Partition Algorithm

```
input: a list of data
input: leftIndex, rightIndex
output: index of pivot

set pivotInde                                  right
set pivotValu

swap values a
set partitionIndex to leftIndex

for each value between leftIndex and rightIndex-1:
   if next list value < pivotValue:
      swap next list value and value at partitionIndex
      add 1 to partitionIndex

swap values at rightIndex and partitionIndex
return partitionIndex
```

Keep track of the last place we stored items that are smaller than the pivot

Partition Algorithm

```
input: a list of data
input: leftIndex, rightIndex
output: index of pivot

set pivotIndex to random index between left and right
set pivotValue to value at pivotIndex

swap values at piv
set partitionIndex
```

Process every item in range before the pivot

```
for each value between leftIndex and rightIndex-1:
    if next list value < pivotValue:
        swap next list value and value at partitionIndex
        add 1 to partitionIndex

swap values at rightIndex and partitionIndex
return partitionIndex
```

Partition Algorithm

```
input: a list of data
input: leftIndex, rightIndex
output: index of pivot

set pivotIndex to random index between left and right
set pivotValue to value at pivotIndex

swap values at pivot
set partitionIndex t

for each value between leftIndex and rightIndex-1:
   if next list value < pivotValue:
      swap next list value and value at partitionIndex
      add 1 to partitionIndex

swap values at rightIndex and partitionIndex
return partitionIndex
```

If the item is smaller than the pivot, we need to make sure it's with the other smaller items

Partition Algorithm

```
input: a list of data
input: leftIndex, rightIndex
output: index of pivot

set pivotIndex to random index between left and right
set pivotValue to value at pivotIndex

swap values at pivotIndex and rightIndex
set partitionIndex to leftIndex

for each value be
   if next list v
      swap next l                                           dex
      add 1 to pa

swap values at rightIndex and partitionIndex
return partitionIndex
```

After the loop, everything smaller than the pivot is before `lastStorageIndex`, so put the pivot there

Partition Algorithm

```
input: a list of data
input: leftIndex, rightIndex
output: index of pivot

set pivotIndex to random index between left and right
set pivotValue to value at pivotIndex

swap values at pivotIndex and rightIndex
set partitionIndex to leftIndex

for each value between leftIndex and rightIndex-1:
   if next list v
      swap next                                 dex
      add 1 to pa

swap values at rightIndex and partitionIndex
return partitionIndex
```

Finally, output the position of the pivot (could be anywhere in the range!)

Quick Sort Algorithm

```
input: a list of data
input: leftIndex
input: rightIndex

if leftIndex < rightIndex:

   set pivotIndex to result of
          partition(list, leftIndex, rightIndex)

   quickSort(list, leftIndex, pivotIndex-1)
   quickSort(list, pivotIndex+1, rightIndex)
```

Quick Sort Algorithm

```
input: a list of c
input: leftIndex
input: rightIndex
```

Base case happens when this is false – it means we are done sorting!

```
if leftIndex < rightIndex:

    set pivotIndex to result of
            partition(list, leftIndex, rightIndex)

    quickSort(list, leftIndex, pivotIndex-1)
    quickSort(list, pivotIndex+1, rightIndex)
```

Quick Sort Algorithm

```
input: a list of data
input: leftIndex
input: rightInde

if leftIndex < r

    set pivotIndex to result of
           partition(list, leftIndex, rightIndex)

    quickSort(list, leftIndex, pivotIndex-1)
    quickSort(list, pivotIndex+1, rightIndex)
```

Otherwise, we break the problem into two pieces around a random pivot value…

Quick Sort Algorithm

```
input: a list of data
input: leftIndex
input: rightIndex

if leftIndex < rightIndex:

   set pivotIndex to result of
          partition(list, leftIndex, rightIndex)

   quickSort(list, leftIndex, pivotIndex-1)
   quickSort(list, pivotIndex+1, rightIndex)
```

...and recurse on the smaller-sized problems.