Searching in an Array

- We can have other recursive formulations
- Search1: search (a, start, end, key)
  - Search key between a[start]…a[end]

```
if start > end, return 0;
if a[start] == key, return 1;
return search(a, start+1, end, key);
```

Searching in an Array

- One more recursive formulations
- Search2: search (a, start, end, key)
  - Search key between a[start]…a[end]

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
return search(a, start, mid-1, key)
            || search(a, mid+1, end, key);
```

Estimating the Time taken

- Two types of operations
  - Function calls
  - Other operations (call them simple operations)
- Assume each simple operation takes fixed amount of time (1 unit) to execute
  - Really a very crude assumption, but will simplify calculations
- Time taken by a function call is proportional to the number of operations performed by the call before returning.

Estimating the Time taken

- Search1

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

  - Let T(n) denote the time taken by search on an array of size n.
  - Line 1 takes 1 unit (or 2 units if you consider if check and return as two operations)
  - Line 2 takes 1 unit (or 3 units if you consider if check, array access and return as three operations)
  - But what about line 3?

Estimating the Time taken

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - What about line 3?
  - Remember the assumption: *Let T(n) denote the time taken by search on an array of size n.*
  - Line 3 is searching in n-1 sized array => takes T(n-1) units
  - But what about the value of T(n) ?

Estimating the Time taken

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - But what about the value of T(n) ?
  - Looking at the body of search, and the information we gathered on previous slides, we can come up with a recurrence relation:

$$T(n) = T(n-1) + C$$

  - We need to solve the recurrence to get the estimate of time

Estimating the Time taken

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - Solution to the recurrence?

$$T(n) = T(n-1) + C \,,\; T(0) = C$$

$$T(n) = Cn$$

  - The worst case run time of Search1 is proportional to the size of array
    - Bigger the array, slower the search
  - What is the best case run time?
  - Which one is more important to consider?

Estimating the Time taken

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
return search(a, start, mid-1, key)
          || search(a, mid+1, end, key);
```

- Search2
  - Recurrence?

$$T(n) \leq T(n/2) + T(n/2) + C$$

  - Solution?

$$T(n) \propto n$$

  - The **worst case** run time of Search2 is also proportional to the size of array
    - Can we do better?

Can we search Faster?

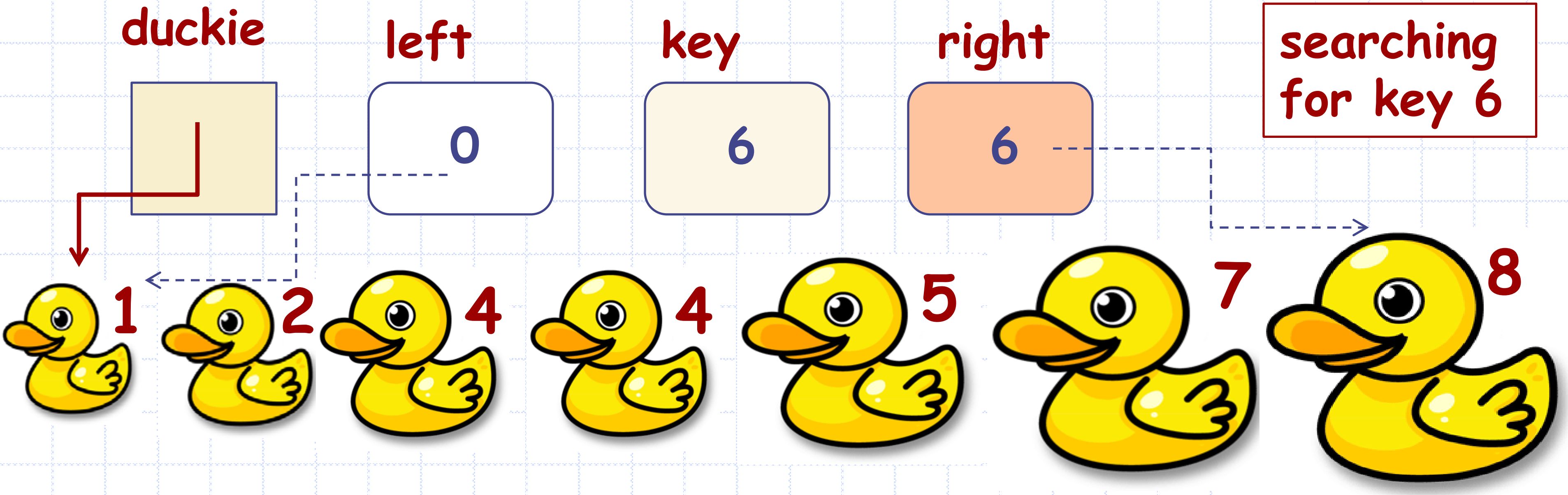
- Yes, provided the elements in the array are sorted
  - in either ascending or descending order

Let us take an example. We have an array of numbers, sorted in non-descending order.

```
int duckie [] = {1,2,4,4,5,6,7};
```

some numbers can be repeated, like 4 in duckie[]

To illustrate the idea, consider searching for the number 6 in the array.



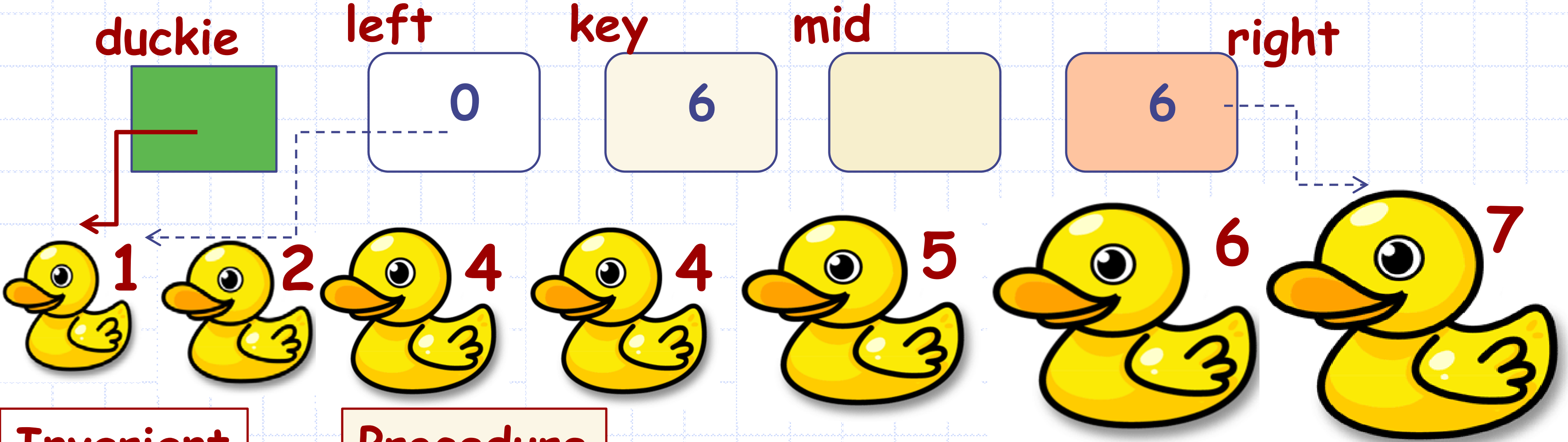
searching for key 6

Initialization

Keep two indices, left and right.
Initially left is 0 and right is the rightmost index in the array. Here right is 6.

Invariant

The key that is being searched for lies in between the indices left and right in the array duckie[] (both ends included), if at all it is in the array.



BINARY SEARCH

Invariant

The key lies between the indices left and right in the array duckie[], if at all it is in the array.

Procedure

Calculate the middle index of left and right

mid =(left + right)/2

Compare duckie[mid] with key. There are 3 possible outcomes.

duckie[mid] == key

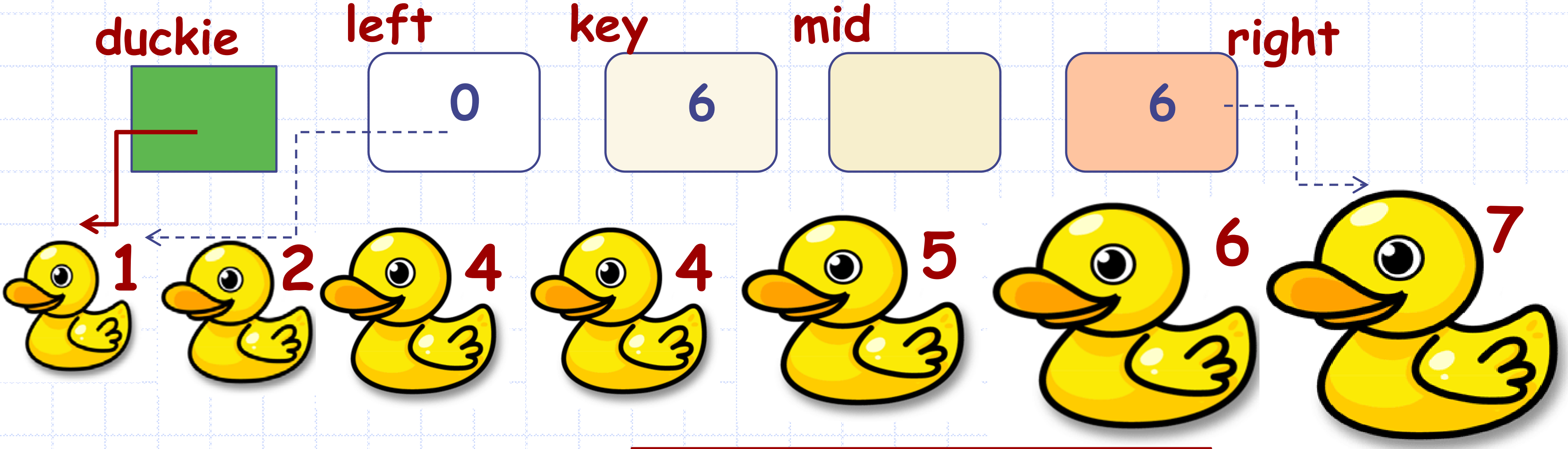
duckie[mid] == key
Key is found.
return mid.

duckie[mid] < key

Key may lie between duckie[mid+1] and duckie[right].

duckie[mid] > key

Key may lie between duckie[left] and duckie[mid-1].



Let us trace the procedure on the duckie array

BINARY SEARCH

Calculate the middle index of left and right

mid =(left + right)/2

Compare duckie[mid] with key. There are 3 possible outcomes.

duckie[mid] == key

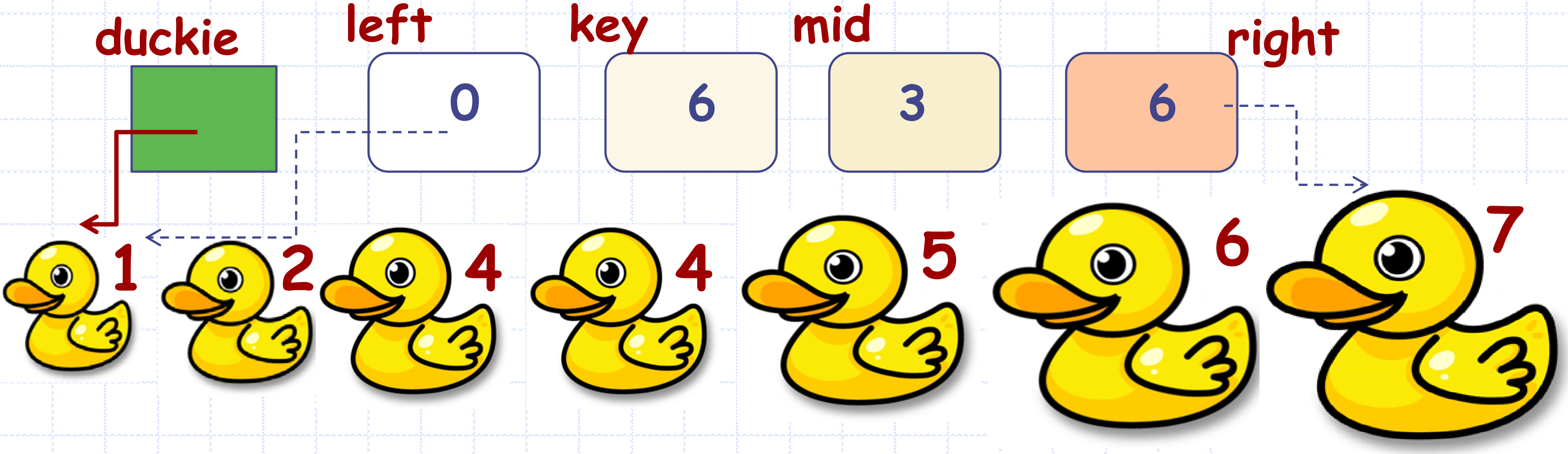
duckie[mid] == key
Key is found.
return mid.

duckie[mid] < key

Key may lie between duckie[left] and duckie[mid-1].

duckie[mid] > key

Key may lie between duckie[mid+1] and duckie[right].



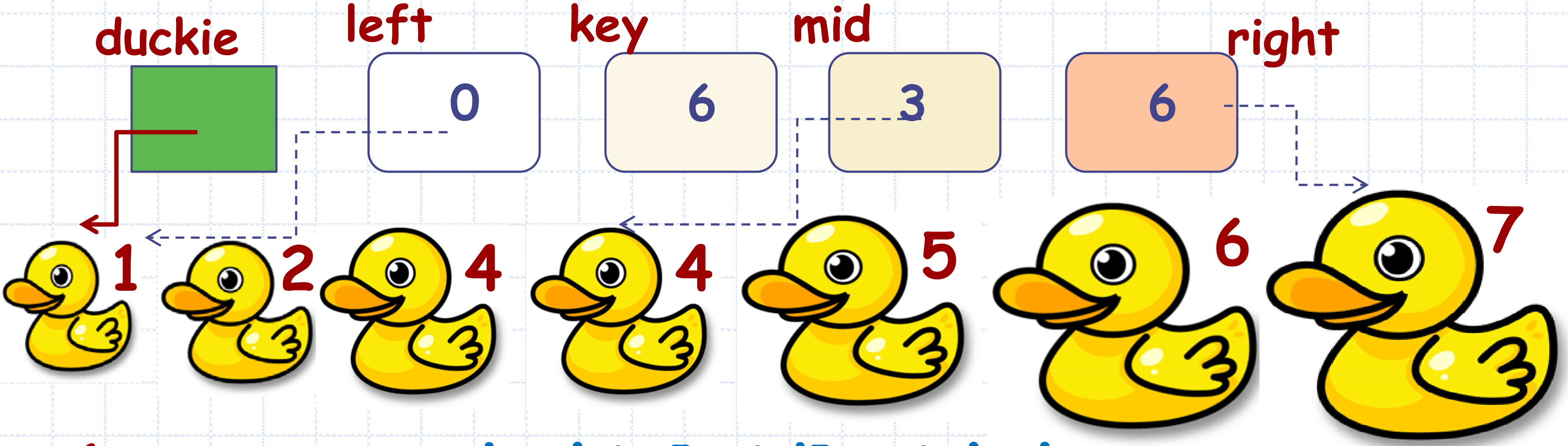
BINARY SEARCH

Calculate mid:

mid = (0 + 6)/2

mid is 3

Compare duckie[3] with key.



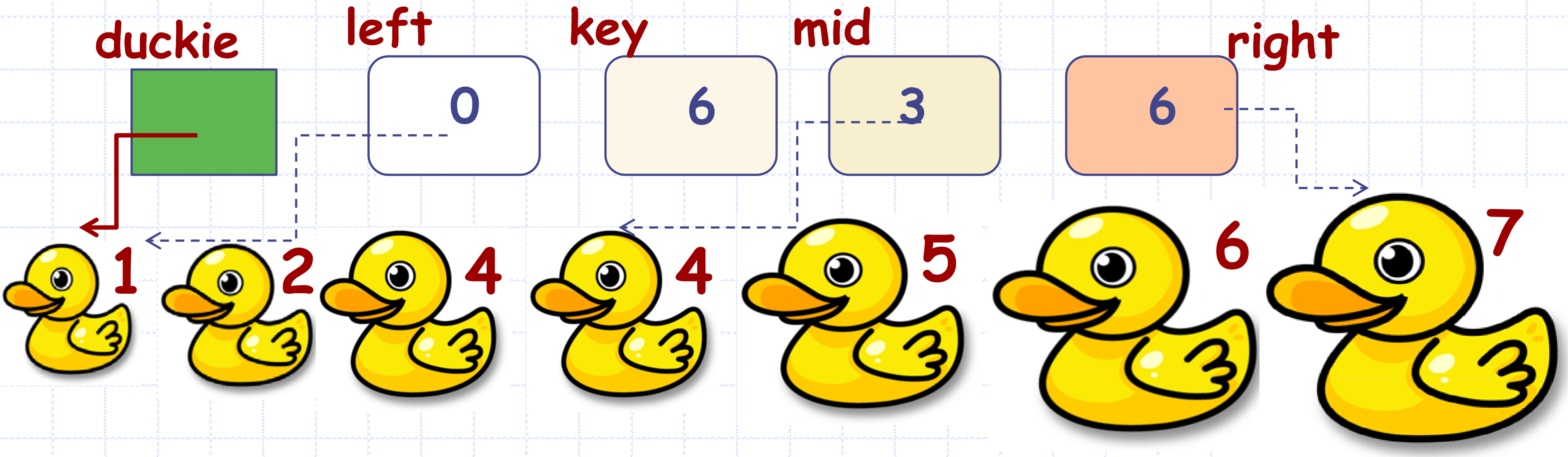
1. compare duckie[mid] with key.

BINARY SEARCH

2. duckie[mid] is 4, key is 6.

3. 4 < 6 so key may only lie among duckie[mid+1] to duckie[right]

What to do now?

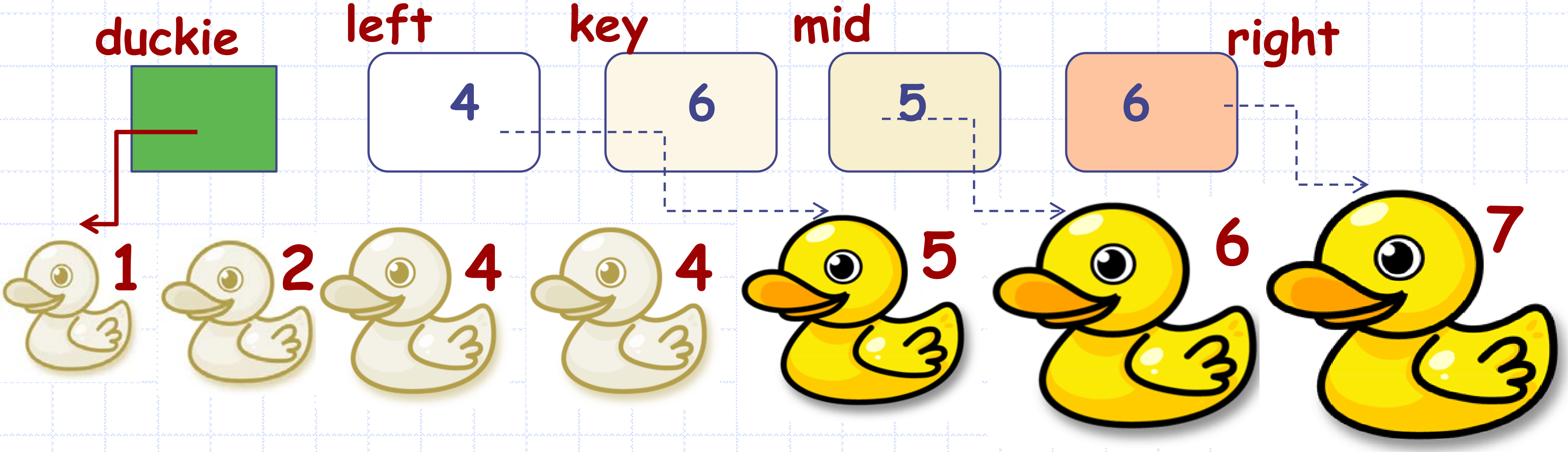


1. We know for sure that key does not lie in the index range 0..mid.

BINARY SEARCH

2. So we can set left to mid+1.

3. Continue the loop…

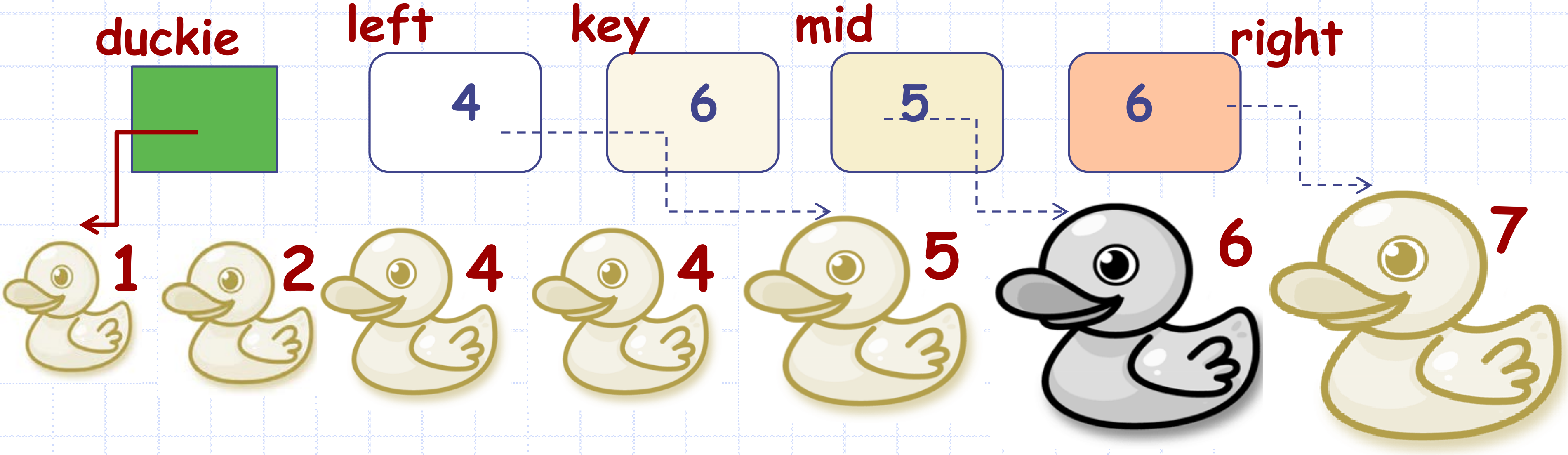


1. Continuing…

2. mid = (left+right)/2. So mid is 5.

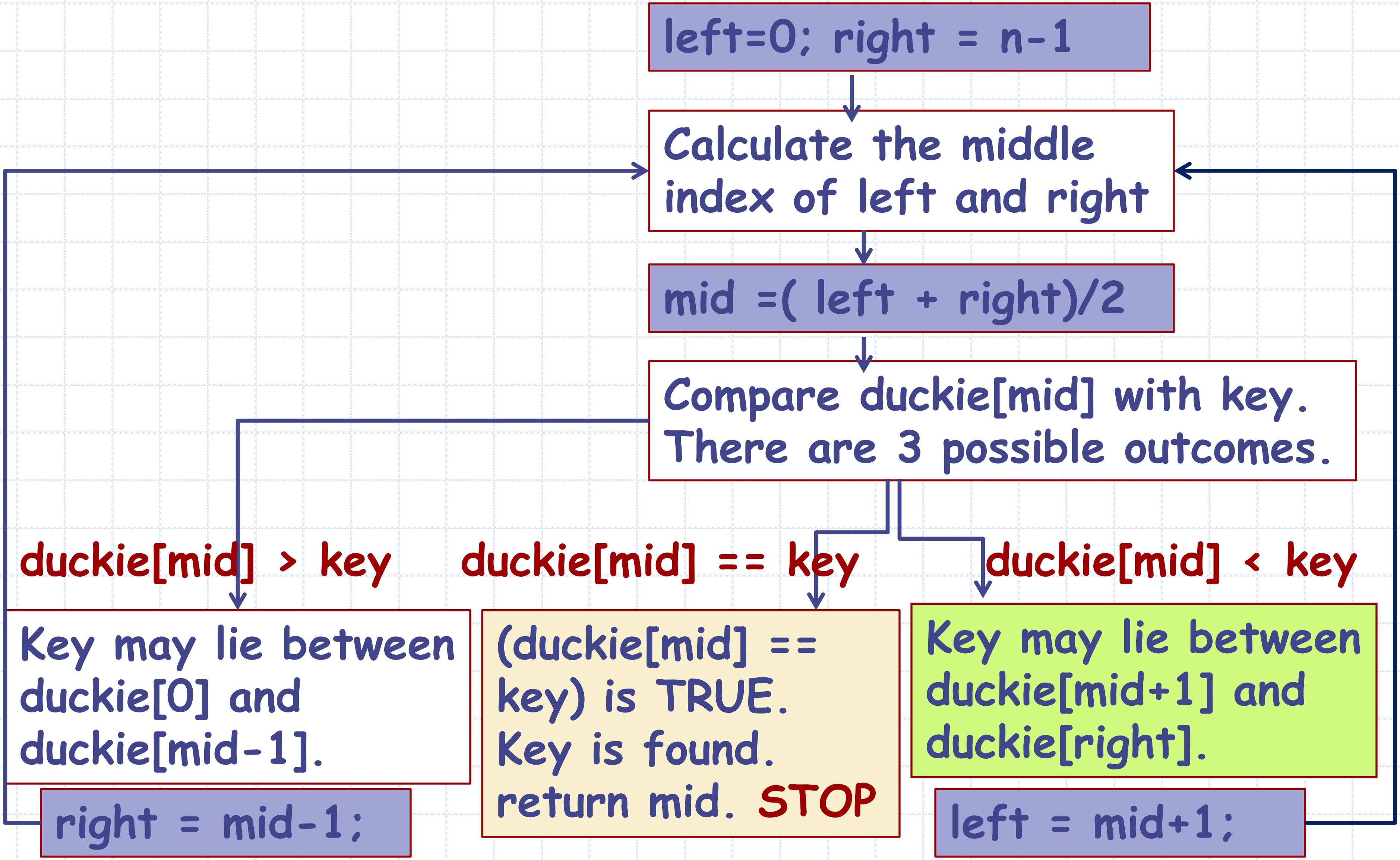
3. Now duckie[5] is 6, so we have found the key.

BINARY SEARCH



BINARY SEARCH

1. We have found the key, so we can return the index 5.
2. Let us complete the flow of procedure.



left=0; right = n-1

Calculate the middle index of left and right

mid =(left + right)/2

Compare duckie[mid] with key. There are 3 possible outcomes.

duckie[mid] > key

Key may lie between duckie[0] and duckie[mid-1].

right = mid-1;

duckie[mid] == key

(duckie[mid] == key) is TRUE. Key is found. return mid. **STOP**

duckie[mid] < key

Key may lie between duckie[mid+1] and duckie[right].

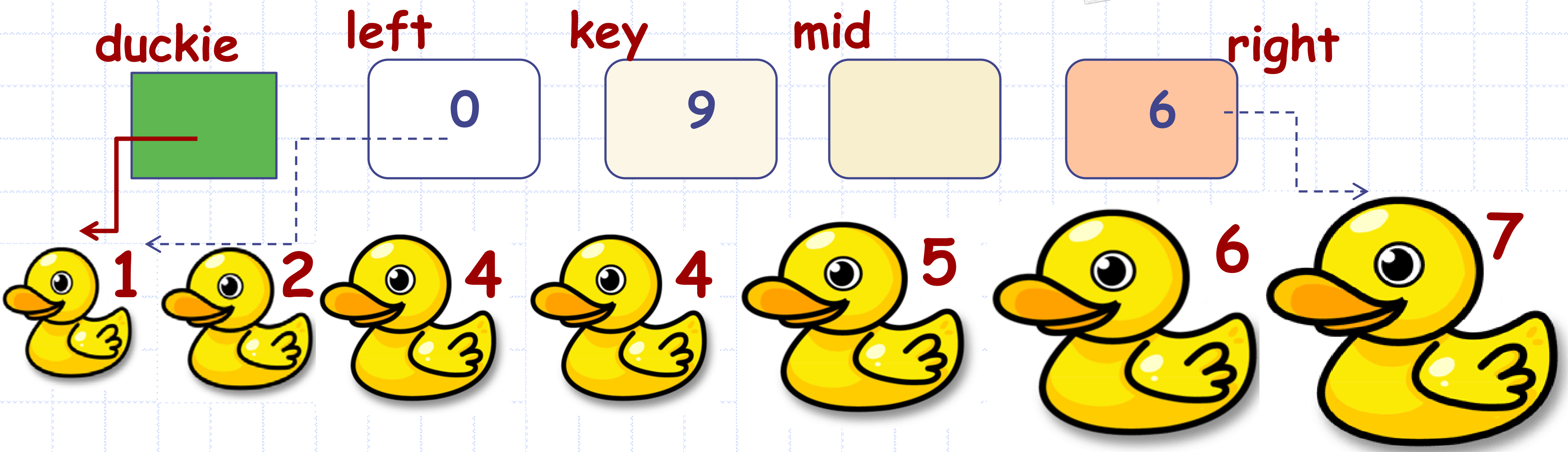
left = mid+1;

Something is still missing…

BINARY SEARCH

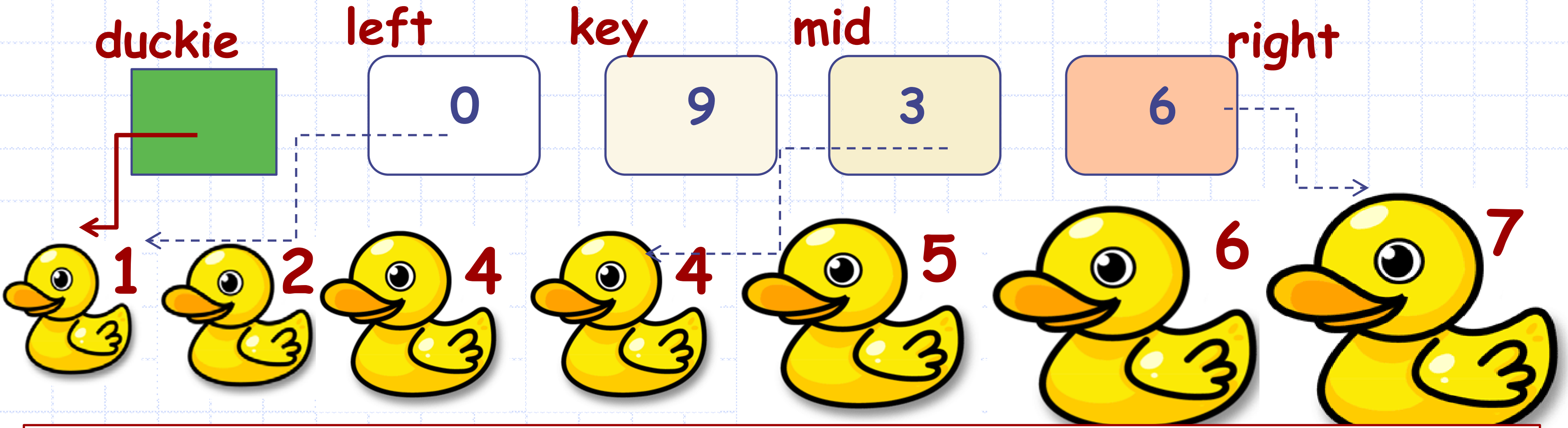
Let us search for 9 in this array.

an example of a search that does not succeed



Key is 9
Initial values: left is 0,
mid is undefined,
right is 6.

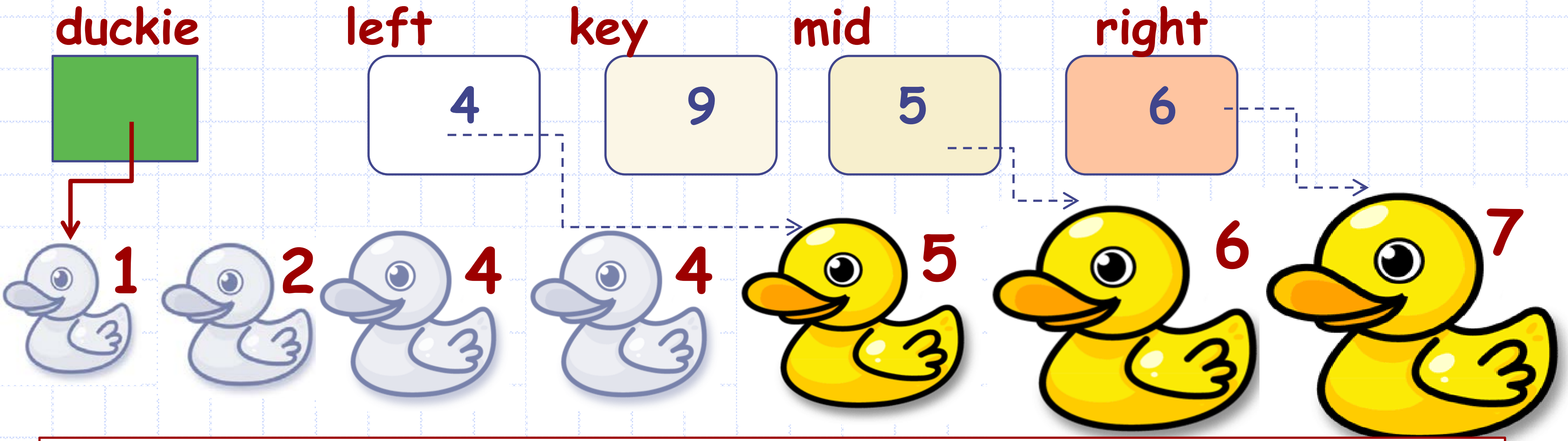
searching for 9.



Set mid to mid = (left+right)/2.
mid is 3. Compare duckie[mid] with key.
duckie[mid] is 4, key is 9 and 4<9.

So we have to move right, meaning left is set to mid+1.
So left will be 4.

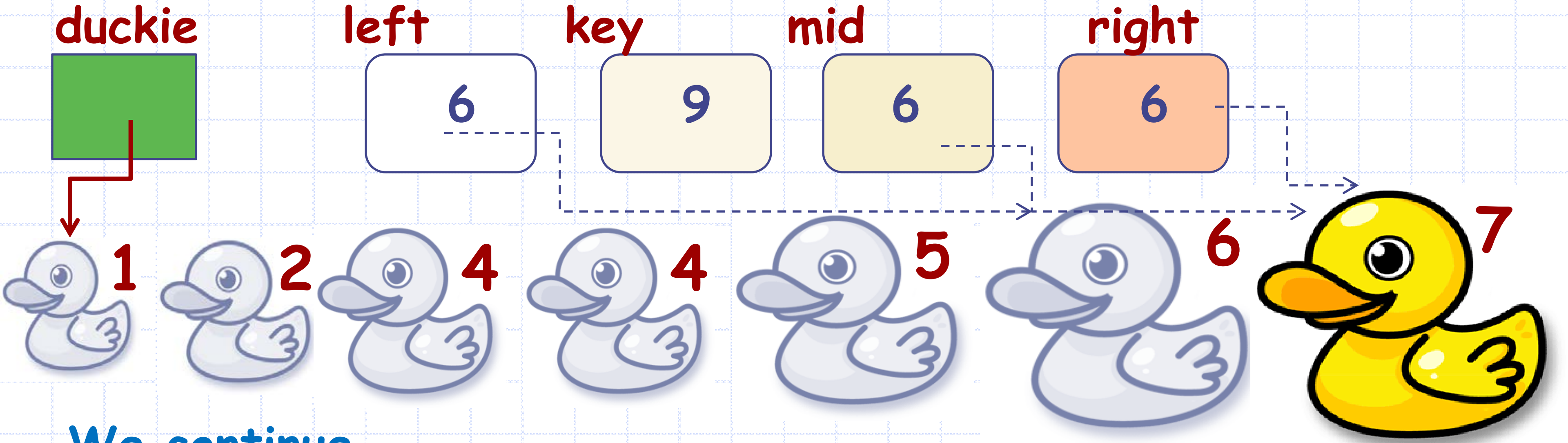
search for 9



Set mid to (left+right)/2. So mid is (4+6)/2 equals 5.
Compare duckie[mid] with key.
duckie[mid] is 6, key is 9 and 6<9.

So, we have to move right again.
Set left to mid +1, so left becomes 6.

search for 9



duckie

left 6

key 9

mid 6

right 6

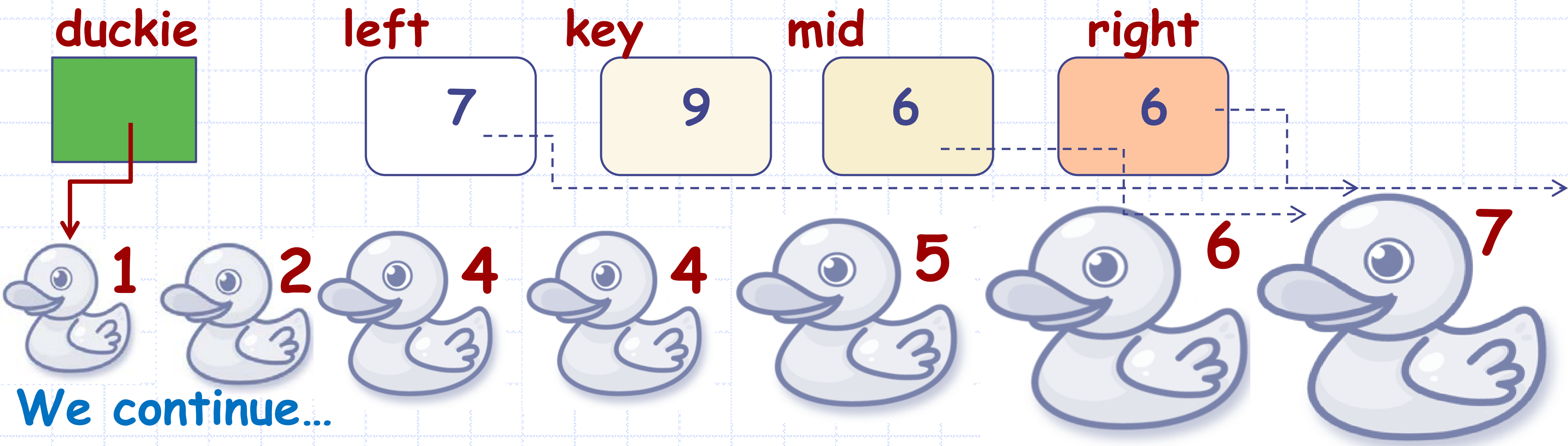
1 2 4 4 5 6 7

We continue…

Set mid to mid = (left+right)/2.
So mid is (6+6)/2 equals 6.
Compare duckie[mid] with key. duckie[mid] is 7, key is 9 and 7<9.

So, we have to move right again.
Set left to mid +1, so left becomes 7.

search for 9



We continue…

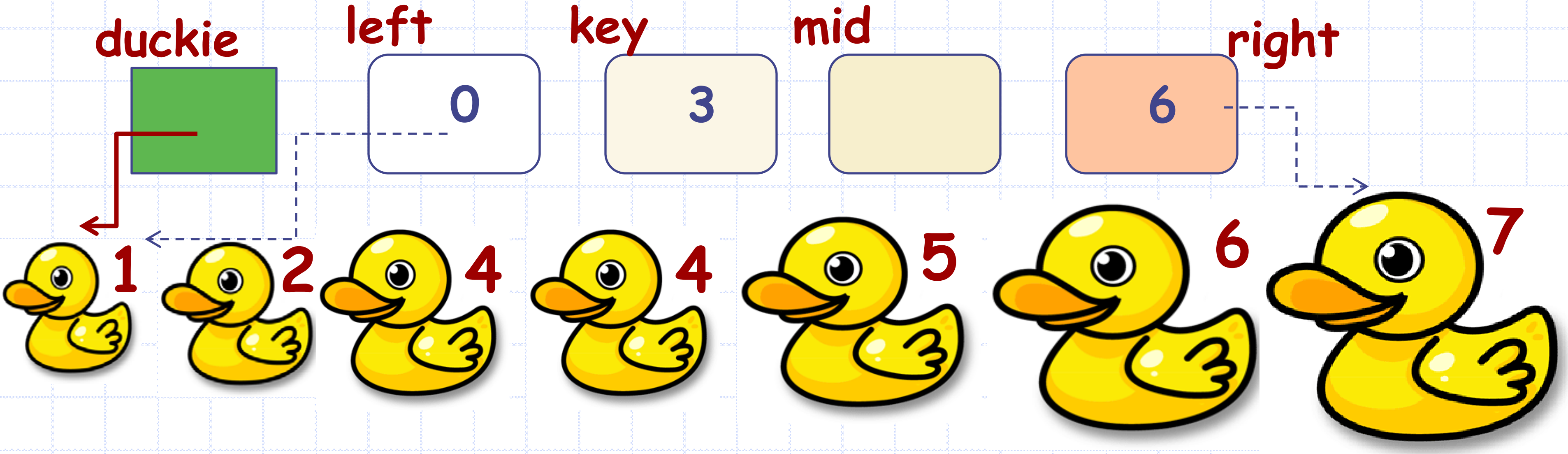
left is 7, right is 6.
The two ends have crossed over.
So the item is not there in the array!

NOT FOUND!

By invariant, item is there between duckie[left] and duckie[right] so long as left <= right.

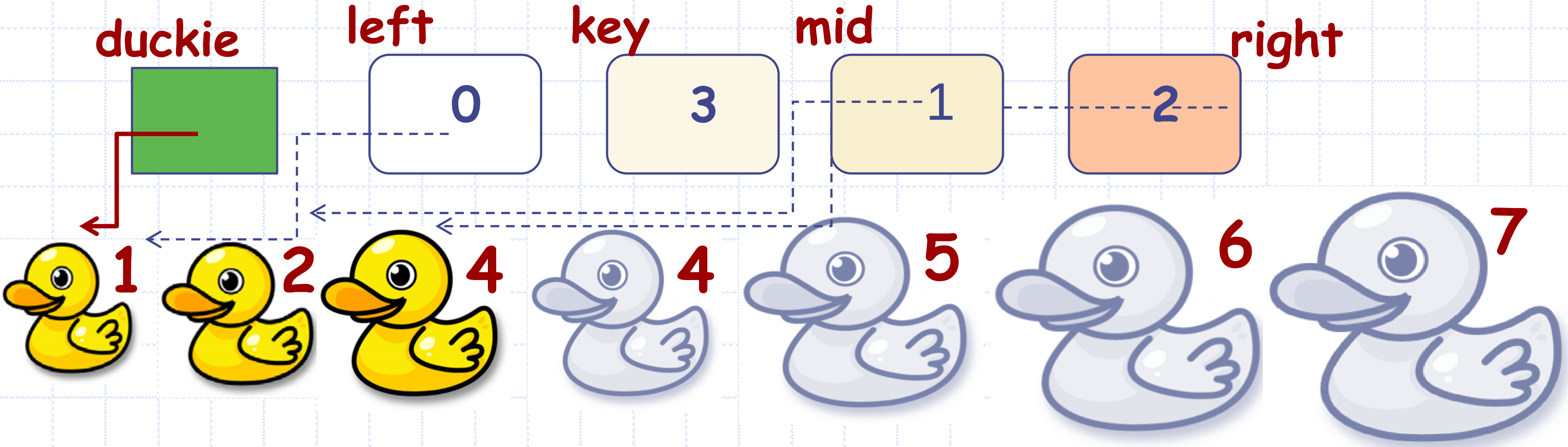
OK, so another condition when the loop terminates is left > right. Is there any other termination condition? Can we search for 3?

Searching for 3 in this array.



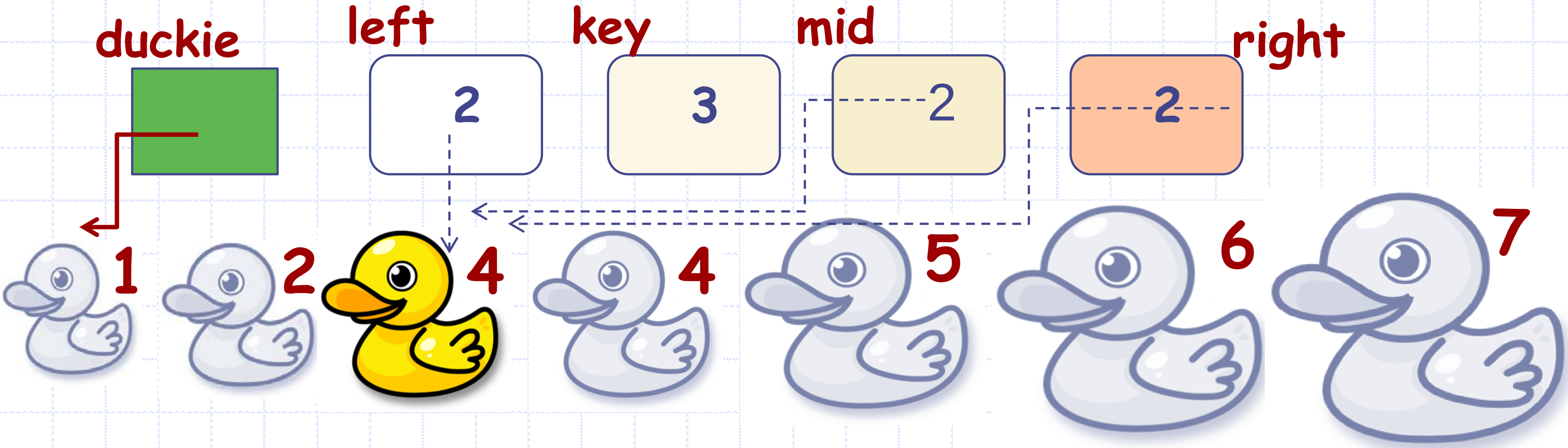
1. left is 0, right is 6.
2. mid is (0+6)/2 which is 3.
3. duckie[mid] is 4, key is 3, so we have to move left.
4. right will be set to mid-1.

Searching for 3 in this array.



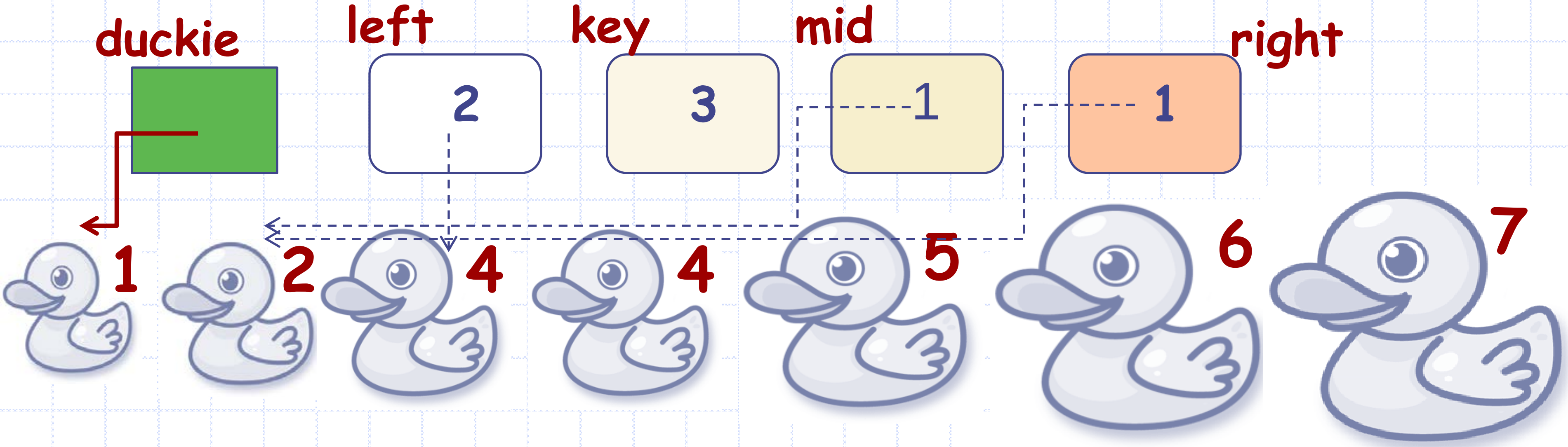
1. left is 0, right is 2.
2. mid is (0+2)/2 which is 1.
3. duckie[mid] is 2 , key is 3, so we have to move right.
4. left will be set to mid+1.

Searching for 3 in this array.



1. left is mid+1 which is 2, right is 2.
2. Now mid is (2+2)/2 which is 2.
3. duckie[mid] is 4 , key is 3, so we have to move left.
4. right will be set to mid-1,So right will be 1.

Searching for 3 in this array.



1. left is 2, right is 1.
2. Left and right have crossed over, NOT FOUND!

Binary Search for Sorted Arrays

- binsearch(a, start, end, key)
  - Search key between a[start]…a[end], where a is a sorted (non-decreasing) array

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
if (a[mid] > key)
        return binsearch(a, start, mid-1, key);
else  return binsearch(a, mid+1, end, key);
```

Wait, isn't this same as search2?

- Lets look closely

```
int search2(a, start, end, key) {
  if start > end, return 0;
  mid = (start + end)/2 ;
  if a[mid]==key, return 1;
  return search2(a, start, mid-1, key)
           || search2(a, mid+1, end, key);
}
```

In worst case,
Both search2 may fire.
But, **only ONE** of the two
binsearch will fire.

```
int binsearch(a, start, end, key) {
  if start > end, return 0;
  mid = (start + end)/2 ;
  if a[mid]==key, return 1;
  if (a[mid] > key)
        return binsearch(a, start, mid-1,
key);
  else  return binsearch(a, mid+1, end, key);
}
```

It matters for the time taken!

How does it matter?

Estimating the Time taken

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
if (a[mid] > key)
        return binsearch(a, start, mid-1, key);
else  return binsearch(a, mid+1, end, key);
```

- binsearch
  - Recurrence?

$$T(n) = T(n/2) + C$$

  - Solution?

$$T(n) \propto \log n$$

  - The worst case run time of binsearch is proportional to the log of the size of array
    - Much faster than Search/Search1/Search2 for large arrays
    - Remember: It works for SORTED arrays only

Some problems related to binary search

Given a sorted array, find the left-most (right-most) occurrence of a key.

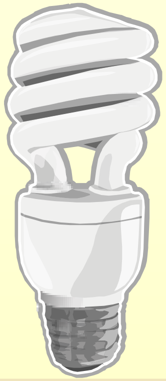
Given a key, find its **successor (predecessor)** in the array. That is, find the smallest (largest) value larger (smaller) than the given key that occurs in the array.

You are not allowed to:

1. Find an occurrence of the key and then sequentially go left (for predecessor) or go right (for successor).
2. **Why?**
3. because this may have linear complexity. Solve the problem as efficiently as binary search, that is, number of comparisons is bounded by constant times log(n).
4. Also the given key may not exist in the array.

Recursion vs Iteration

```
int fib(int n)
{
   int first = 0, second = 1;
   int next, c;
   if (n <= 1)
      return n;
   for ( c = 1; c < n ; c++ ) {
      next = first + second;
      first = second;
      second = next;
   }
   return next;
}
```



The recursive program is closer to the definition and easier to read.

But very very inefficient

```
int fib(int n)
{
   if ( n <= 1 )
      return n;
   else
      return fib(n-1) + fib(n-2);
}
```