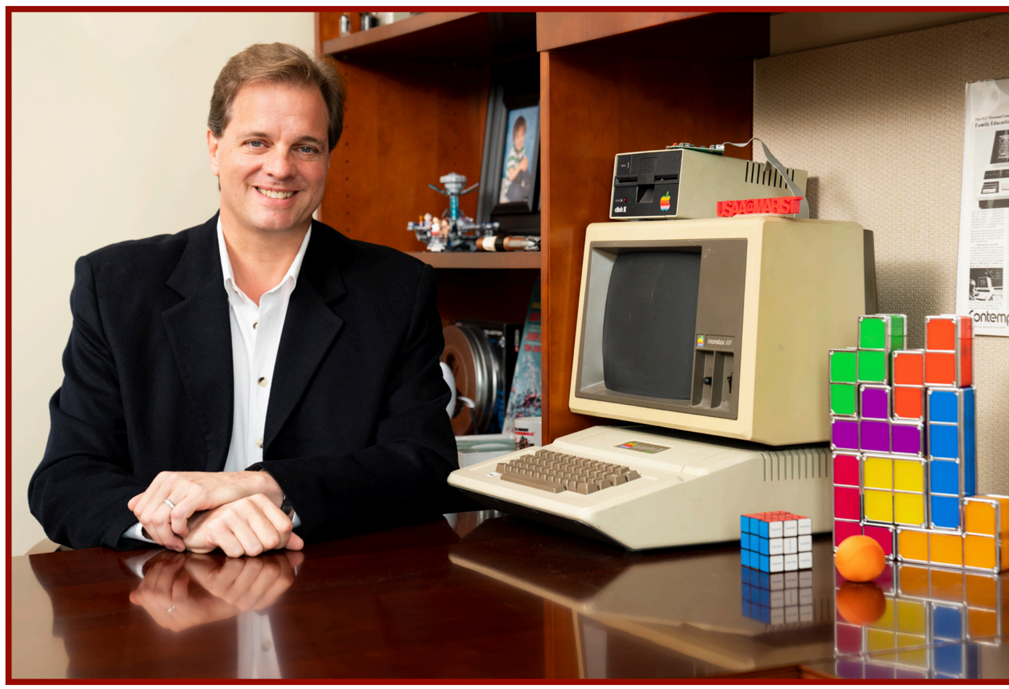
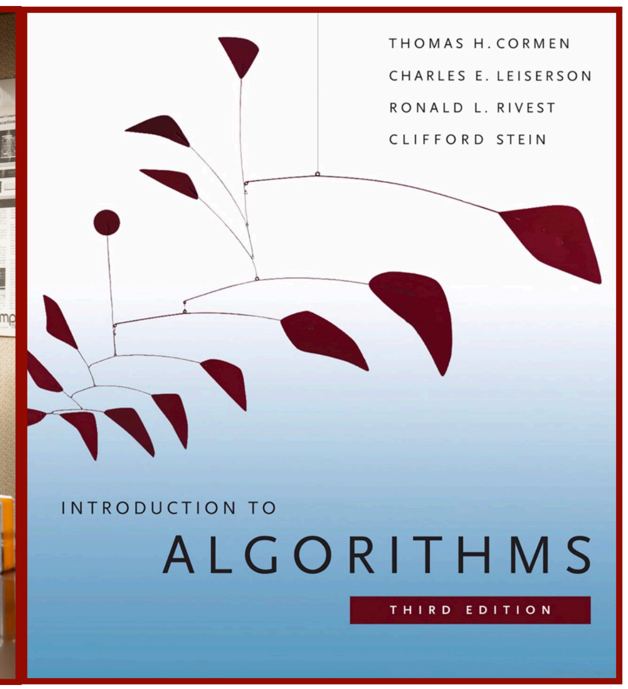
Searching





Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu

Searching

Imagine a data structure containing 9 billion **unordered** names

# 1	Peart
# 2	Schock
# 3	Crump
.	
.	
.	
# 8,999,999,998	White
# 8,999,999,999	Purdie
# 9,000,000,000	Bruford

Searching

Imagine a data structure containing 9 billion **unordered** names
and we want to locate **one** of them.

check # 1	Peart	Is this it? If not, *n*-1 to go.
check # 2	Schock	Is this it? If not, *n*-2 to go.
check # 3	Crump	Is this it? If not, *n*-3 to go.
.		
.		
.		
check # 8,999,999,998	White	Is this it? If not, 2 to go.
check # 8,999,999,999	Purdie	Is this it? If not, 1 to go.
check # 9,000,000,000	Bruford	Is this it? If not, it's not here.

Searching

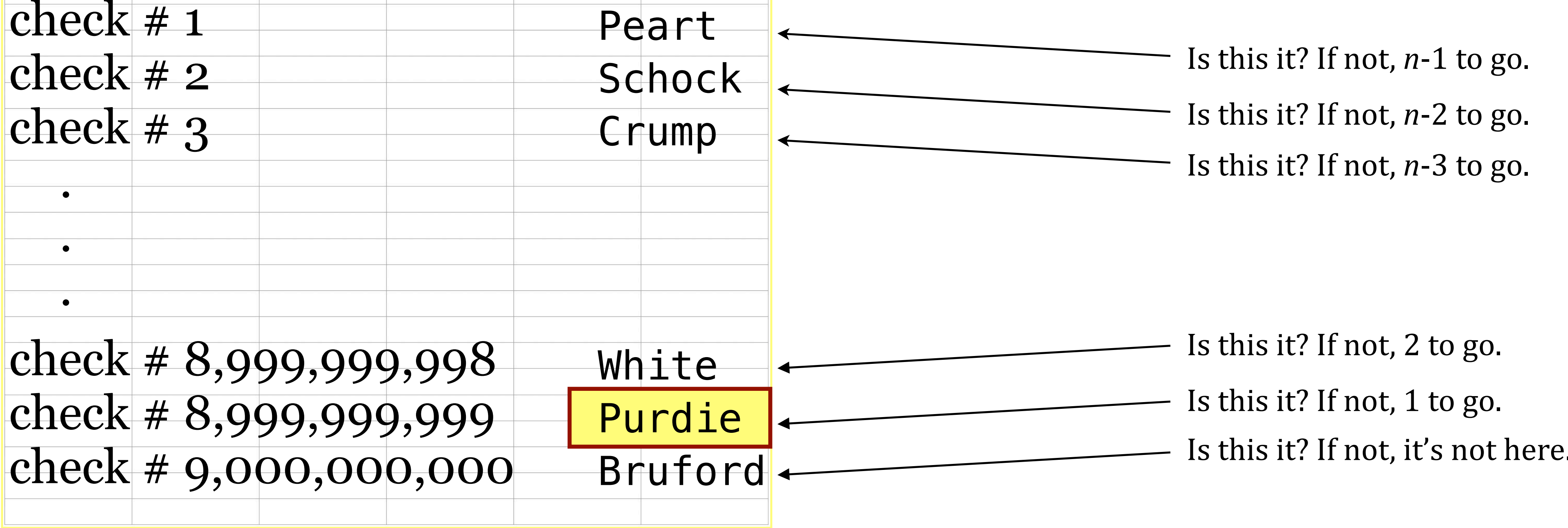
Imagine a data structure containing 9 billion **unordered** names and we want to locate **one** of them.

check # 1	Peart	Is this it? If not, *n*-1 to go.
check # 2	$\boxed{\texttt{Schock}}$	Is this it? If not, *n*-2 to go.
check # 3	Crump	Is this it? If not, *n*-3 to go.
.		
.		
.		
check # 8,999,999,998	White	Is this it? If not, 2 to go.
check # 8,999,999,999	Purdie	Is this it? If not, 1 to go.
check # 9,000,000,000	Bruford	Is this it? If not, it's not here.

Sometimes we will find the target person early.

Searching

Imagine a data structure containing 9 billion **unordered** names
and we want to locate **one** of them.



Sometimes we will find the target person early.
Sometimes we will find the target person late.

Searching

Imagine a data structure containing 9 billion unordered names.

check # 1	Peart
check # 2	Schock
check # 3	Crump
.	
.	
.	
check # 8,999,999,998	White
check # 8,999,999,999	Purdie
check # 9,000,000,000	Bruford

This is called
Linear Search or
Sequential Search

Sometimes we will find the target person early.
Sometimes we will find the target person late.

Q: What's the average — or expected — case for *n* items?

Linear / Sequential Searching

Imagine a data structure containing 9 billion unordered names.

check # 1	`Peart`
check # 2	`Schock`
check # 3	`Crump`
.	
.	
.	
check # 8,999,999,998	`White`
check # 8,999,999,999	`Purdie`
check # 9,000,000,000	`Bruford`

Sometimes we will find the target person early.
Sometimes we will find the target person late.

Q: What's the average — or expected — case for *n* items?
A: The expected case is ½ *n*, which requires
examining 4.5B rows in this example.

Linear / Sequential Searching

Imagine a data structure containing 9 billion unordered names.

check # 1		Peart
check # 2		Schock
check # 3		Crump
.		
.		
.		
check # 8,999,999,998		White
check # 8,999,999,999		Purdie
check # 9,000,000,000		Bruford

That's O(n).

Best case	O(1)
Worst case	O(n)
Average case	O(n)

Recurrence:
$T(n) = T(n\text{-}1) + c$

Sometimes we will find the target person early.
Sometimes we will find the target person late.

Q: What's the average — or expected — case for n items?
A: The expected case is ½ n, which requires
examining 4.5B rows in this example.

Linear / Sequential Searching

Imagine a data structure containing 9 billion unordered names.

check # 1	Peart
check # 2	Schock
check # 3	Crump
.	
.	
.	
check # 8,999,999,998	White
check # 8,999,999,999	Purdie
check # 9,000,000,000	Bruford

That's $O(n)$.

Best case	$O(1)$
Worst case	$O(n)$
Average case	$O(n)$

Recurrence:
$T(n) = T(n\text{-}1) + c$

LIST-SEARCH(L, k)

$x = L.head$
while $x \neq$ NIL and $x.key \neq k$
    $x = x.next$
return x

To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

CLRS 3e p.237

Linear / Sequential Searching

Imagine a data structure containing 9 billion unordered names.

check # 1	Peart
check # 2	Schock
check # 3	Crump
.	
.	
.	
check # 8,999,999,998	White
check # 8,999,999,999	Purdie
check # 9,000,000,000	Bruford

That's O(n).

Best case	O(1)
Worst case	O(n)
Average case	O(n)

Can we do better?

LIST-SEARCH(L, k)

```
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

CLRS 3e p.237

Searching 9 billion people

What if we could **search** through **sorted** data?

After all, we are good at sorting…

… in O(n^2) time — Selection and Insertion sort

… in O($n \log_2 n$) time — Merge sort and Quicksort

Searching 9 billion people

What if we could **search** through **sorted** data?

check # 1	Bruford
check # 2	Crump
check # 3	Peart
.	
.	
.	
check # 8,999,999,998	Purdie
check # 8,999,999,999	Schock
check # 9,000,000,000	White

How would you do it?
What's your strategy?

Want to play a number guessing game?

Searching 9 billion people

What if we could **search** through **sorted** data?

check # 1	Bruford
check # 2	Crump
check # 3	Peart
.	
.	
.	
check # 8,999,999,998	Purdie
check # 8,999,999,999	Schock
check # 9,000,000,000	White

We could pick from the middle. If that's not our target, then we **exclude** the *lower* or *upper* **half of the data**, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.

Have we seen this before?

Divide and . . .

Take a big problem and divide it into two smaller problems.

Take a those problems and divide them into two smaller problems.

Take a those problems and divide them into two smaller problems.

Take a those problems and divide them into two smaller problems.

Take a those problems and divide them into two smaller problems.

Take a those problems and divide them into two smaller problems.

Take a those problems and divide them into two smaller problems.

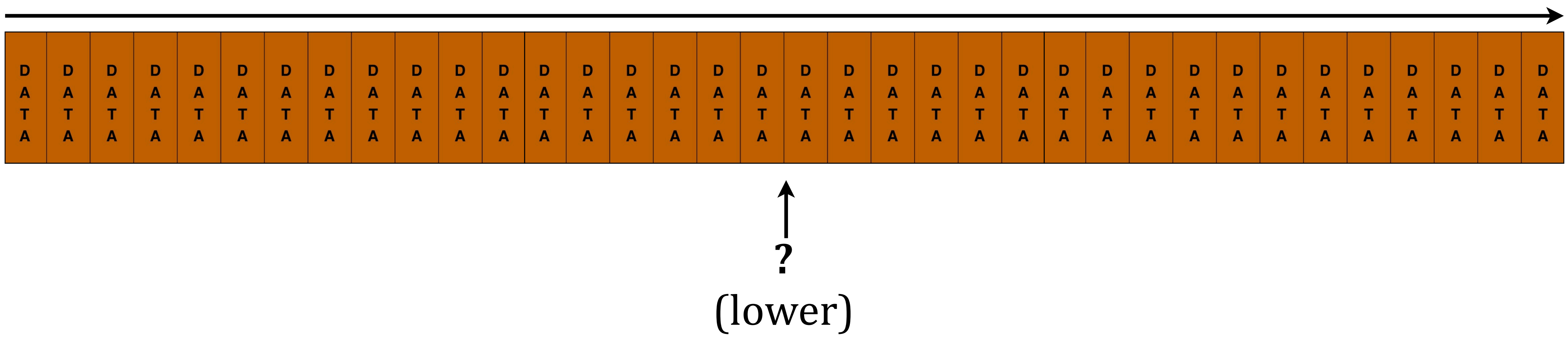
… until the problems get small enough that they are solved.

In this case, it's really just divide.

Let's consider **Binary Search**.

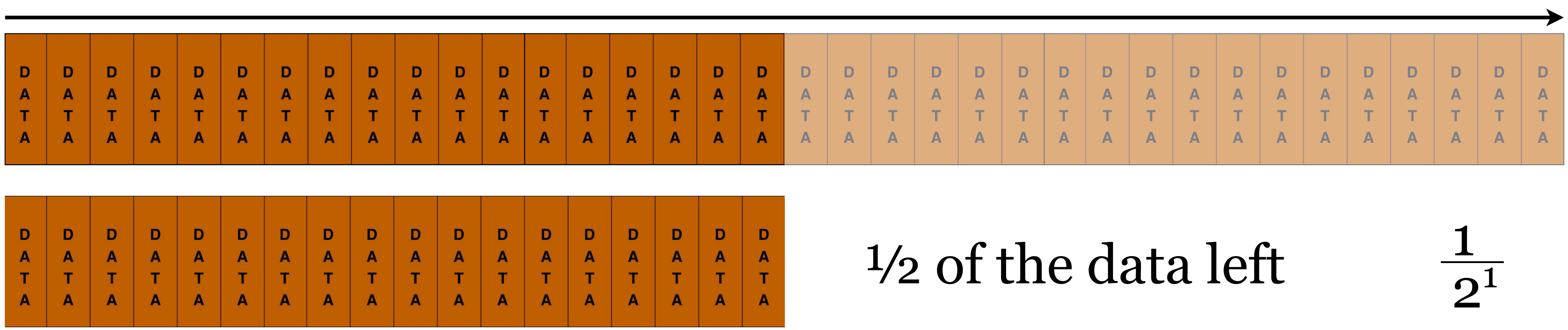
Binary Searching 9 billion people

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



Binary Searching 9 billion people

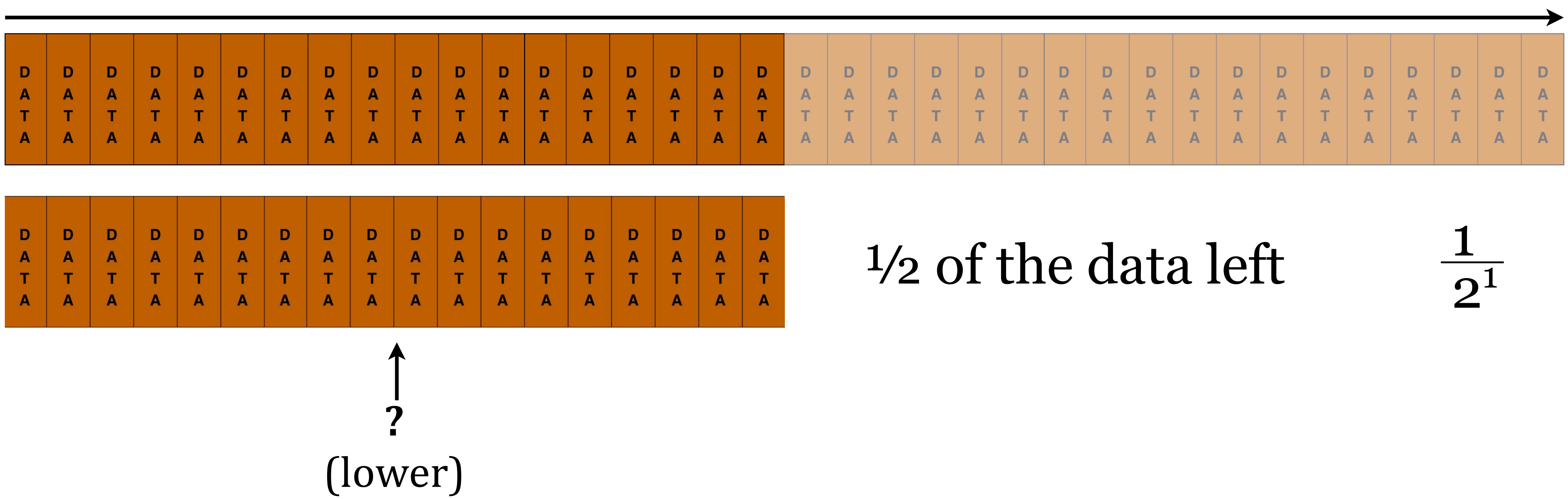
We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



½ of the data left $\frac{1}{2^1}$

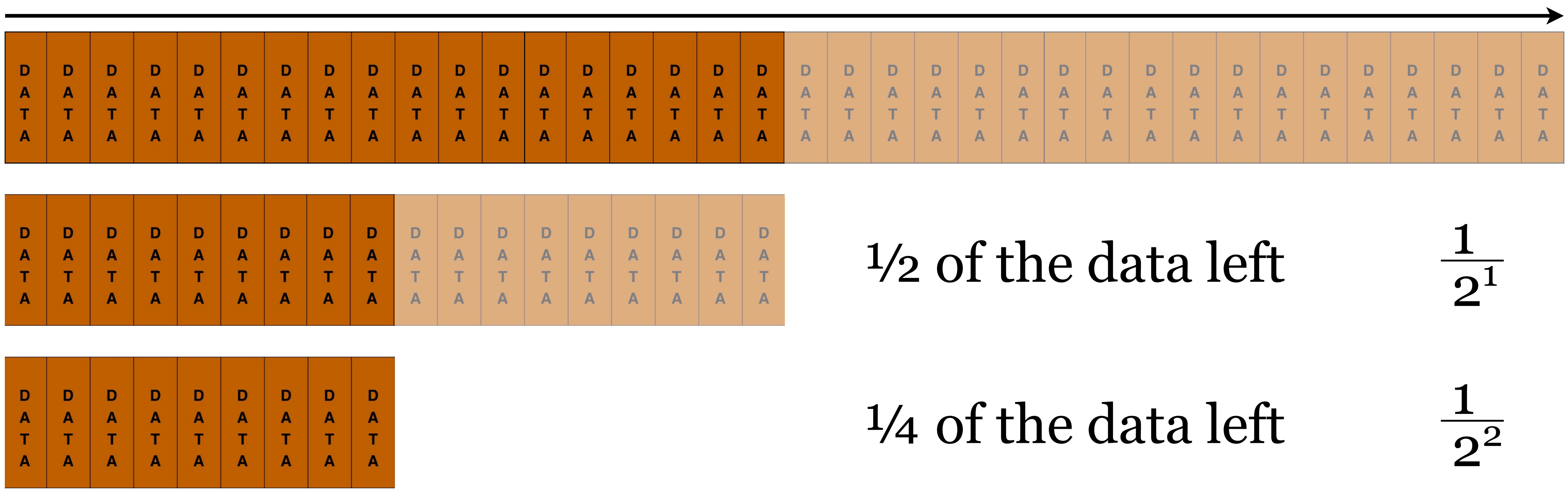
Binary Searching 9 billion people

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



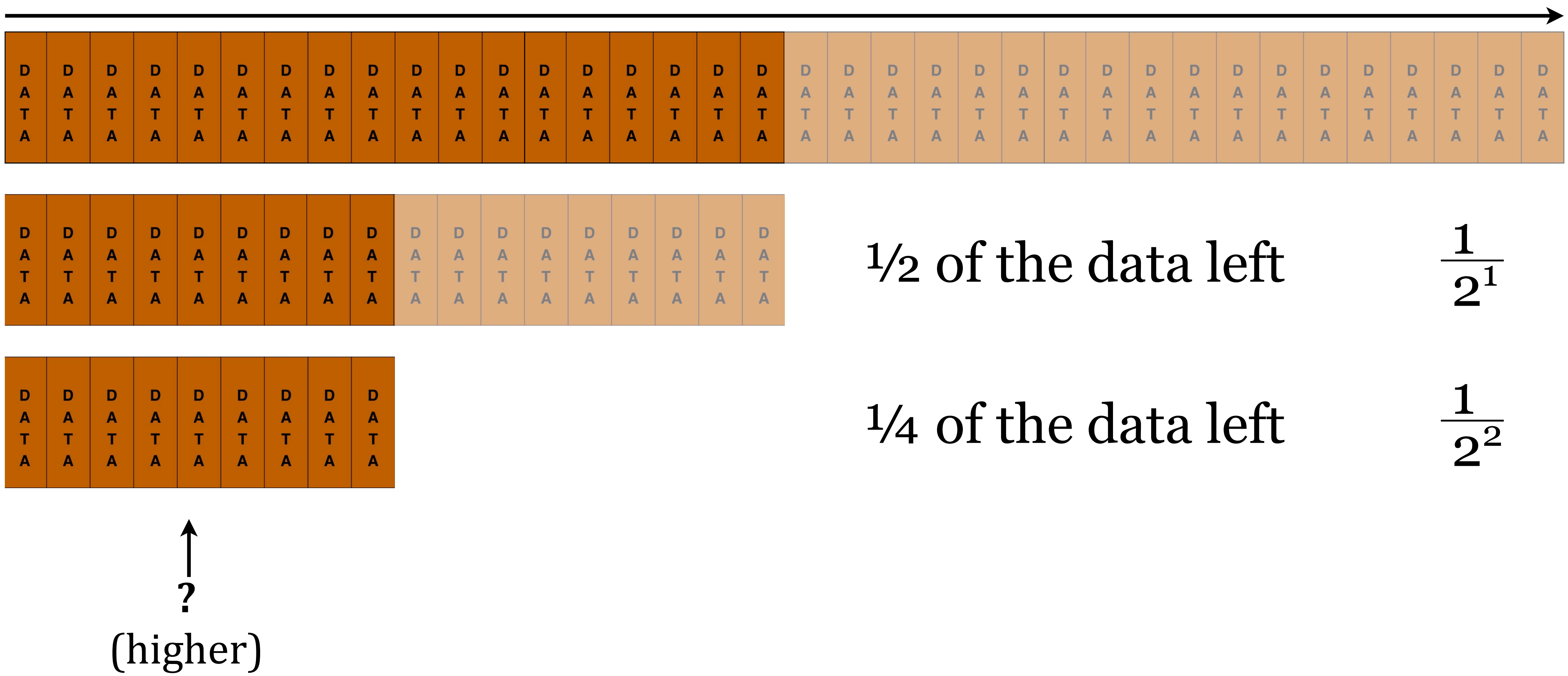
Binary Searching 9 billion people

We could pick from the middle. If that’s not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



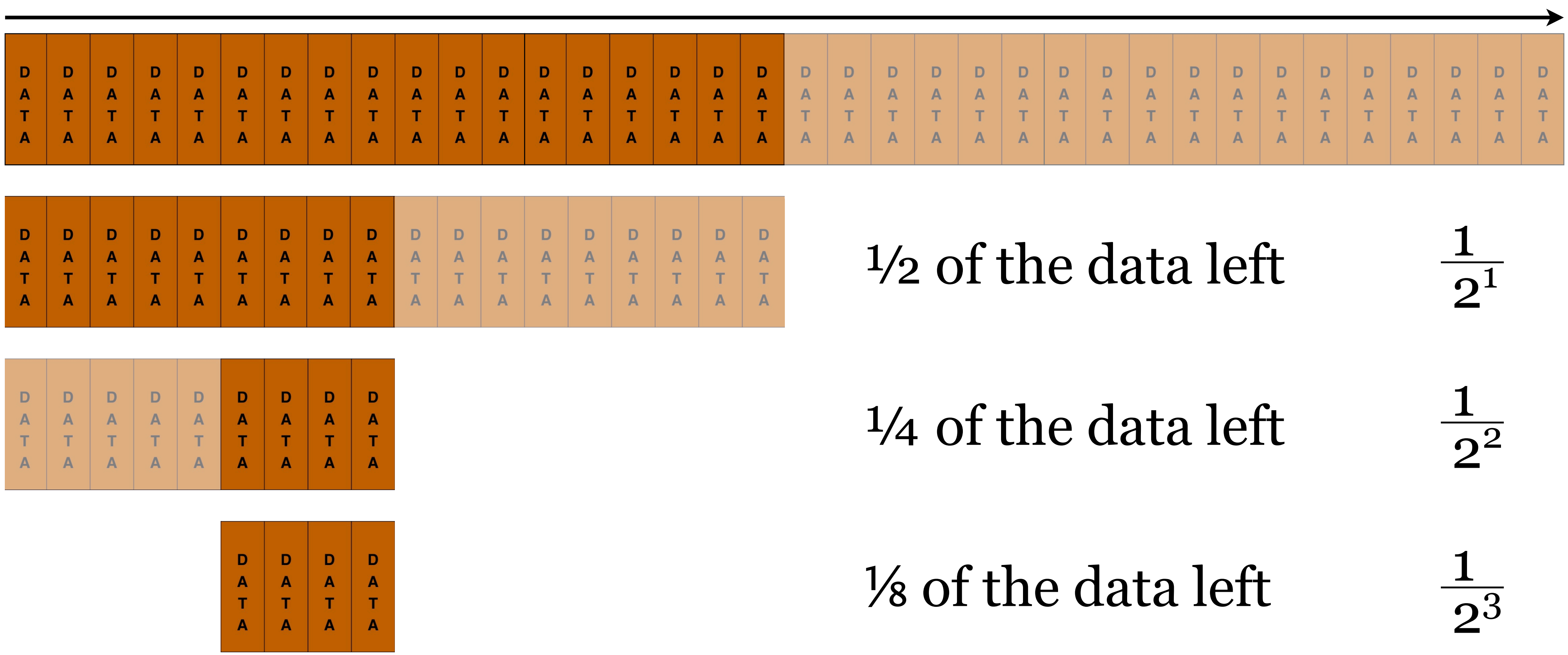
Binary Searching 9 billion people

We could pick from the middle. If that’s not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



Binary Searching 9 billion people

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



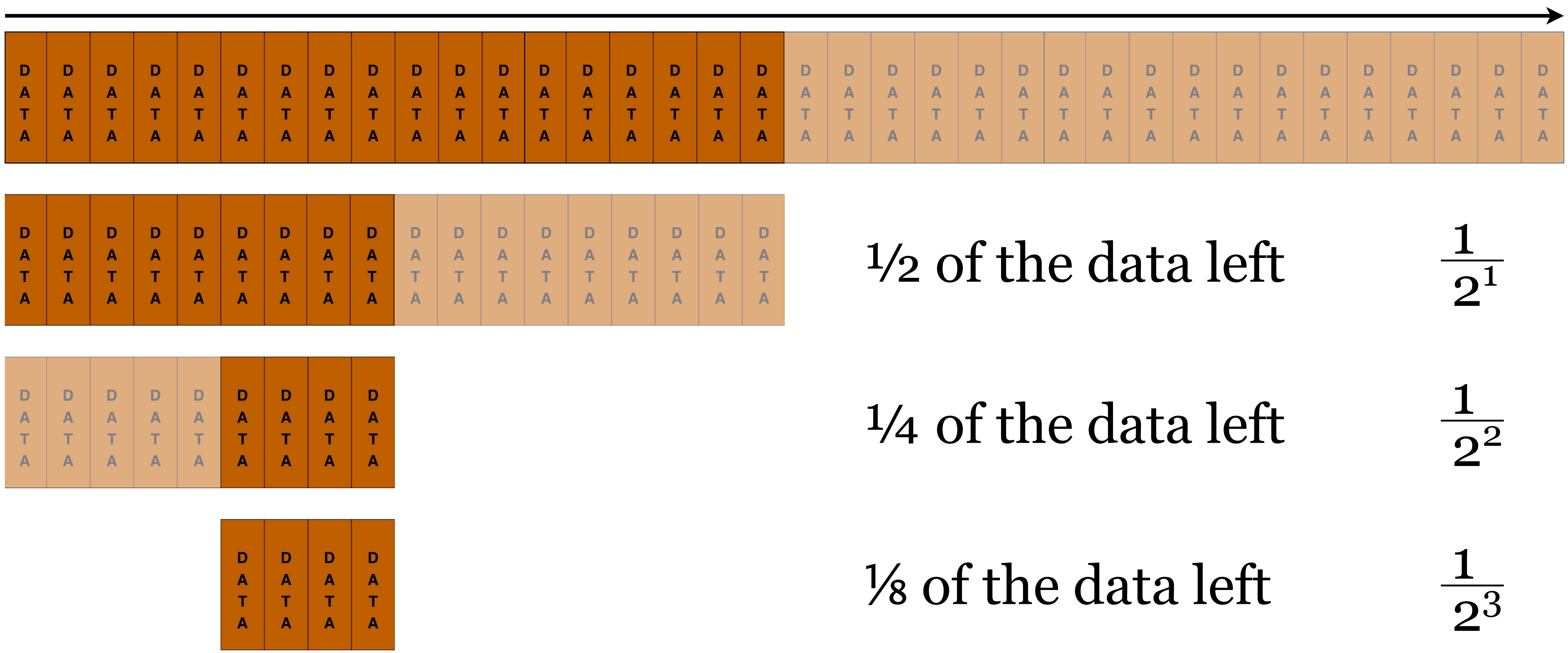
½ of the data left $\frac{1}{2^1}$

¼ of the data left $\frac{1}{2^2}$

⅛ of the data left $\frac{1}{2^3}$

Binary Searching 9 billion people

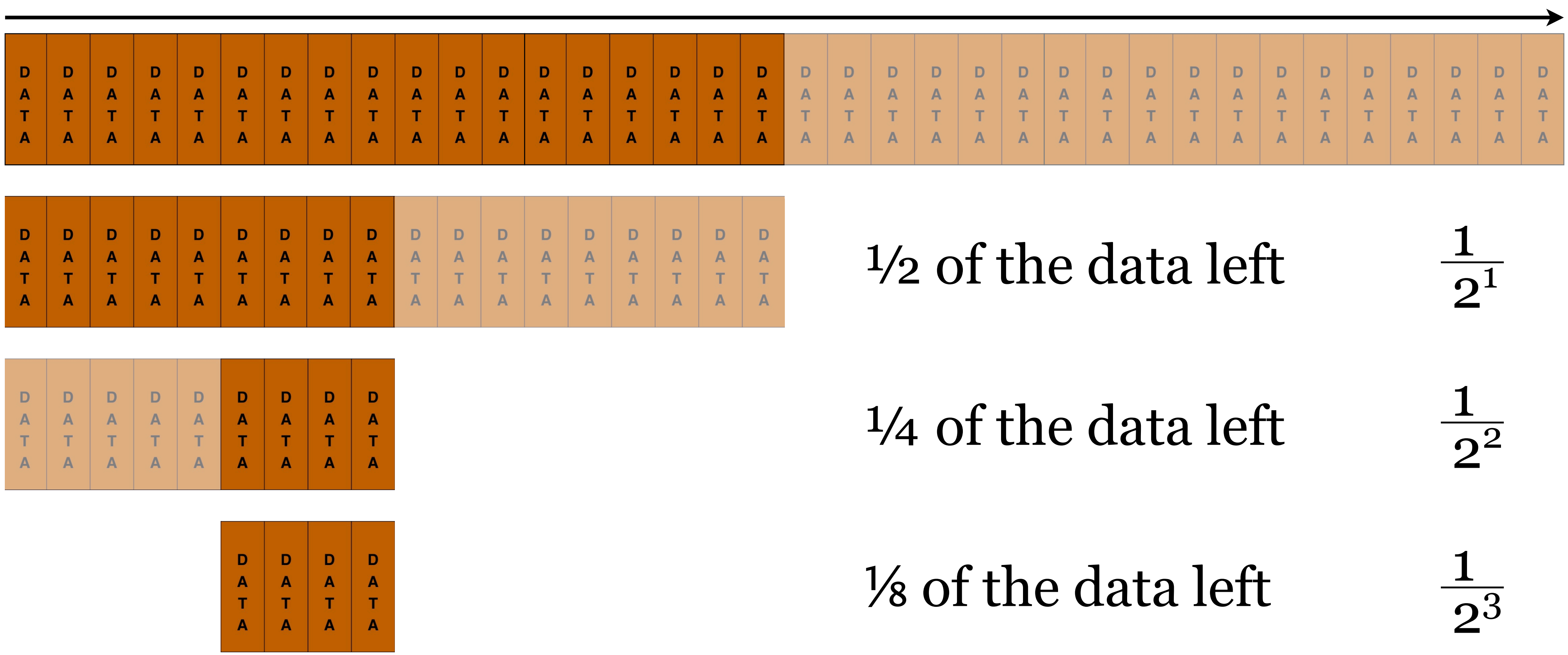
We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



Q: What's the average or — expected — case for *n* rows?

Binary Searching 9 billion people

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



Q: What's the average or — expected — case for *n* rows?
A: The expected case is $\mathbf{\log_2 n}$, because we cut it in half each time.

Binary Searching 9 billion people

What if we could search through **sorted** data?

check # 1	Bruford
check # 2	Crump
check # 3	Peart
.	
.	
.	
check # 8,999,999,998	Purdie
check # 8,999,999,999	Schock
check # 9,000,000,000	White

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.

Q: What's the average or — expected — case for n rows?
A: The expected case is $\log_2 n$. By the way, $\log_2$ 9B is . . . ?

Binary Searching 9 billion people

What if we could search through **sorted** data?

check # 1	Bruford
check # 2	Crump
check # 3	Peart
.	
.	
.	
check # 8,999,999,998	Purdie
check # 8,999,999,999	Schock
check # 9,000,000,000	White

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.

Q: What's the average or — expected — case for n rows?
A: The expected case is $\log_2 n$. By the way, $\log_2$ 9B is . . . 33

Binary Searching 9 billion people

What if we could search through **sorted** data?

check # 1	`Bruford`
check # 2	`Crump`
check # 3	`Peart`
.	
.	
.	
check # 8,999,999,998	`Purdie`
check # 8,999,999,999	`Schock`
check # 9,000,000,000	`White`

Now **that** is a better way!
33 < 4.5B

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.

Q: What's the average or — expected — case for n rows?
A: The expected case is $\log_2 n$. By the way, $\log_2$ 9B is . . . 33

Binary Searching 9 billion people

What if we could search through **sorted** data?

check # 1	Bruford
check # 2	Crump
check # 3	Peart
.	
.	
.	
check # 8,999,999,998	Purdie
check # 8,999,999,999	Schock
check # 9,000,000,000	White

Binary Search is $O(\log_2 n)$.

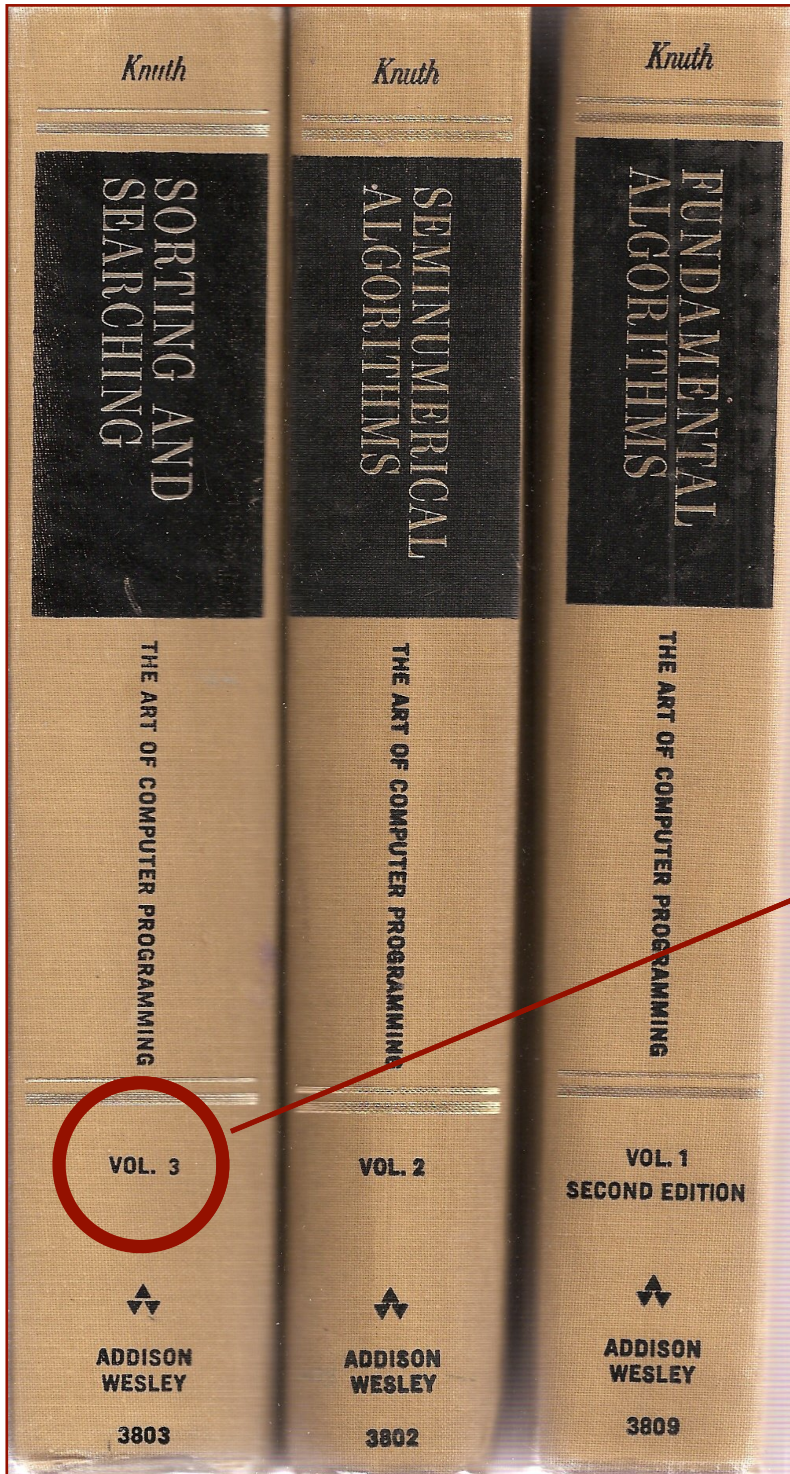
Best case	$O(1)$
Worst case	$O(\log_2 n)$
Average case	$O(\log_2 n)$

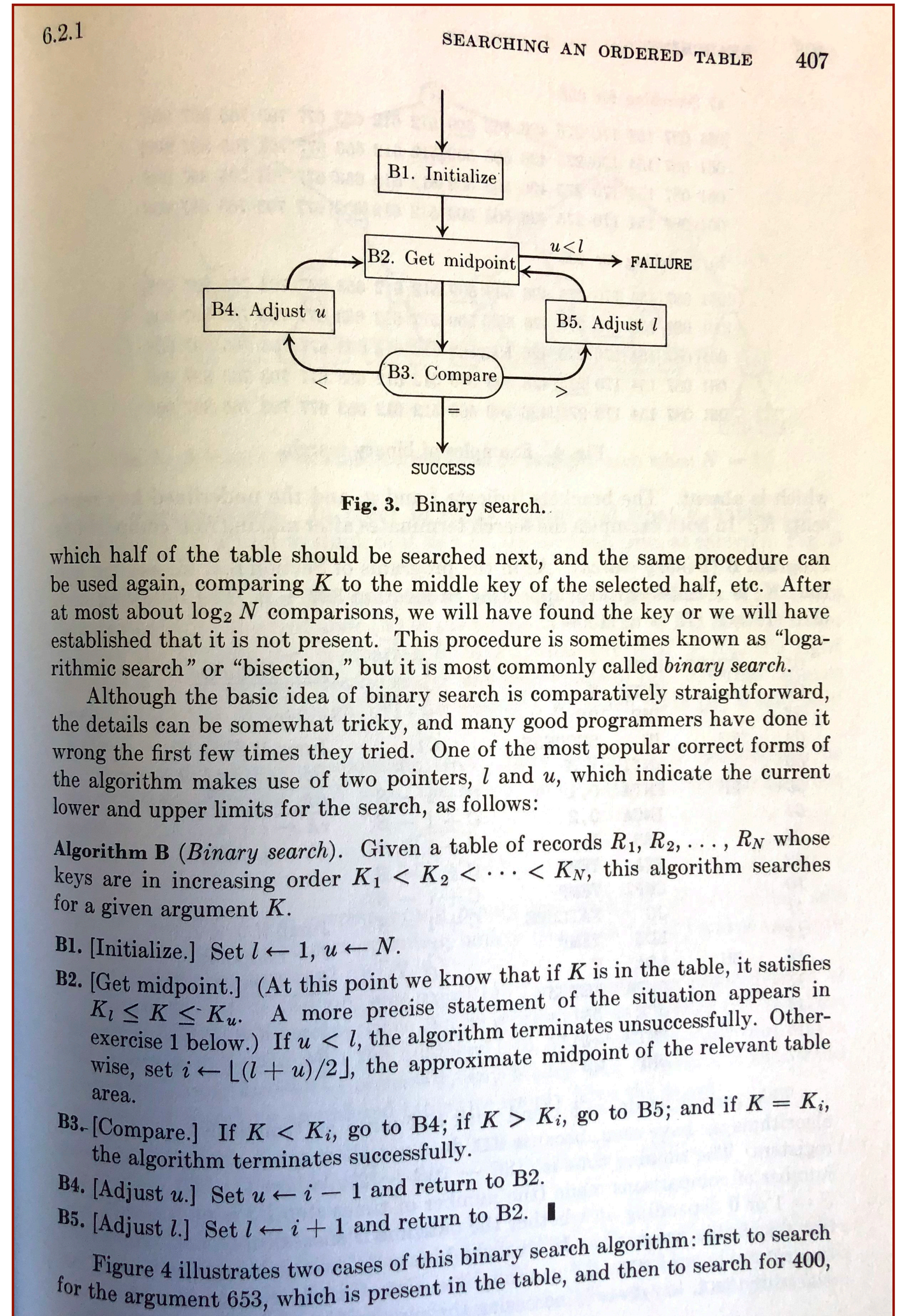
Recurrence:
$T(n) = T(\frac{n}{2}) + c$

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.

Q: What's the average or — expected — case for n rows?
A: The expected case is $\log_2 n$.

Binary Search Algorithm





6.2.1 SEARCHING AN ORDERED TABLE 407

Fig. 3. Binary search.

which half of the table should be searched next, and the same procedure can be used again, comparing K to the middle key of the selected half, etc. After at most about $\log_2 N$ comparisons, we will have found the key or we will have established that it is not present. This procedure is sometimes known as "logarithmic search" or "bisection," but it is most commonly called *binary search*.

Although the basic idea of binary search is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried. One of the most popular correct forms of the algorithm makes use of two pointers, l and u, which indicate the current lower and upper limits for the search, as follows:

Algorithm B (*Binary search*). Given a table of records $R_1, R_2, \ldots, R_N$ whose keys are in increasing order $K_1 < K_2 < \cdots < K_N$, this algorithm searches for a given argument K.

B1. [Initialize.] Set $l \leftarrow 1$, $u \leftarrow N$.

B2. [Get midpoint.] (At this point we know that if K is in the table, it satisfies $K_l \le K \le K_u$. A more precise statement of the situation appears in exercise 1 below.) If $u < l$, the algorithm terminates unsuccessfully. Otherwise, set $i \leftarrow \lfloor (l+u)/2 \rfloor$, the approximate midpoint of the relevant table area.

B3. [Compare.] If $K < K_i$, go to B4; if $K > K_i$, go to B5; and if $K = K_i$, the algorithm terminates successfully.

B4. [Adjust u.] Set $u \leftarrow i - 1$ and return to B2.

B5. [Adjust l.] Set $l \leftarrow i + 1$ and return to B2. ▮

Figure 4 illustrates two cases of this binary search algorithm: first to search for the argument 653, which is present in the table, and then to search for 400,

Binary Search Algorithm

Here is an iterative version of Binary Search from the CLRS text.

```
BINARY-SEARCH(x, T, p, r)
low = p
high = max(p, r + 1)
while low < high
    mid = ⌊(low + high)/2⌋
    if x ≤ T[mid]
        high = mid
    else low = mid + 1
return high
```

CLRS 3e p.799

x - target
T - collection of data
p - start index
r - stop index

Binary Search Algorithm

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

Here is a recursive version of Binary Search, with some issues:

(1) There are two exits. That's bad software craftsmanship.
(2) It does not return **where** the target is found, just that it is.

Fix these issues when you program your own version.

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

```
BinarySearch(A, 0, 7, 2)
```

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

```
BinarySearch(A, 0, 7, 2)
```

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

`BinarySearch(A, 0, 7, 2)`

$\ngtr$

1	2	3	4	5	6	7	8
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

`BinarySearch(A, 0, 7, 2)` ≠

1	2	3	4	5	6	7	8
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

BinarySearch(A, 0, 7, 2) <

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int( (start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

```
BinarySearch(A, 0, 7, 2)
BinarySearch(A, 0, 2, 2)
```

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

```
BinarySearch(A, 0, 7, 2)
BinarySearch(A, 0, 2, 2)
```

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

```
BinarySearch(A, 0, 7, 2)
BinarySearch(A, 0, 2, 2)
```

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

```
BinarySearch(A, 0, 7, 2)
BinarySearch(A, 0, 2, 2)
```

=

1	2	3	4	5	6	7	8
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

```
BinarySearch(A, 0, 7, 2)
BinarySearch(A, 0, 2, 2) → true
```

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)  true
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

```
BinarySearch(A, 0, 7, 2) → true
BinarySearch(A, 0, 2, 2) → true
```

1	2	3	4	5	6	7	8
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

BinarySearch(A, 0, 7, 2) → true
BinarySearch(A, 0, 2, 2) → true

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

```
BinarySearch(A, 0, 7, 2) → true
BinarySearch(A, 0, 2, 2) → true
```

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Binary Search Example

```
proc BinarySearch(A, start, stop, target)
   midPoint = int((start+stop)/2) // round, ceil, floor?
   if (start > stop)
      return false
   else if (target == A[midPoint])
      return true
   else if (target < A[midPoint])
      BinarySearch(A, start, midPoint-1, target)
   else // target > A[midPoint] or not there at all
      BinarySearch(A, midPoint+1, stop, target)
   end if
end proc
```

Remember the issues:

(1) There are two exits. That's bad software craftsmanship.
(2) It does not return **where** the target is found, just that it is.

Fix these issues when you program your own version.