ESC101: Introduction to Computing

Sorting

Around Easter 1961, a course on ALGOL 60 was offered … It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining.

It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded. Due credit must be paid to the genius of the designers of ALGOL 60 who **included recursion in their language** and enabled me to describe my invention so elegantly to the world.

I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed.

- **C. A. R. Hoare, ACM Turing Award Lecture, 1980**

QuickSort - Partition Routine

A useful sub-routine (function) for many problems, including quicksort, one of the popular sorting methods.

1. **Partition takes an array a[] of size n and a value called the pivot.**
2. **The pivot is an element in the array, for instance, a[0].**
3. **Partition re-arranges the array elements into two parts:**
   a) **the left part has all elements <= pivot.**
   b) **the right part has all elements >= pivot.**
4. **Partition returns the index of the beginning of the right part.**

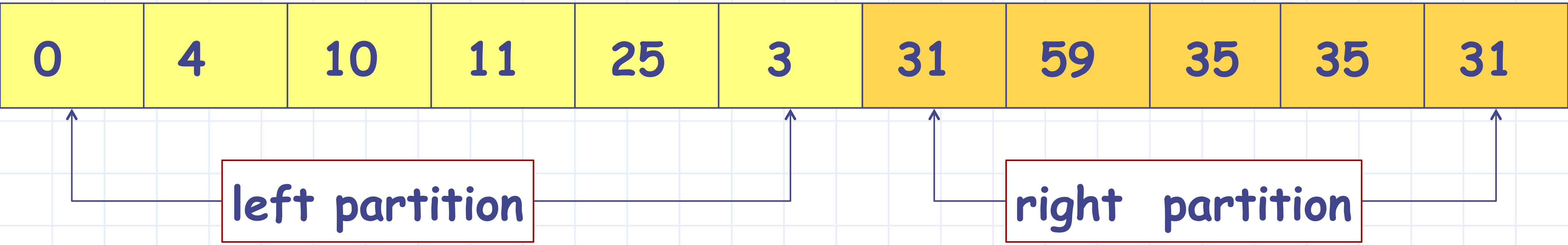
Let us see an example.

1. **Partition** takes an array a[] of size n and a value called the pivot.
2. The pivot is an element in the array, for instance, a[0].
3. Partition re-arranges the array elements into two parts:
   a) all elements in the left part are <= pivot
   b) all elements in the right part are >= pivot

Input Array a[], size is n : 11

31	4	10	35	59	31	3	25	35	11	0

Pivot element is assumed to be a[0]: 31



0	4	10	11	25	3	31	59	35	35	31

Observations

Multiple “partitions” of an array are possible, even for the same pivot. They all would satisfy the above specification.

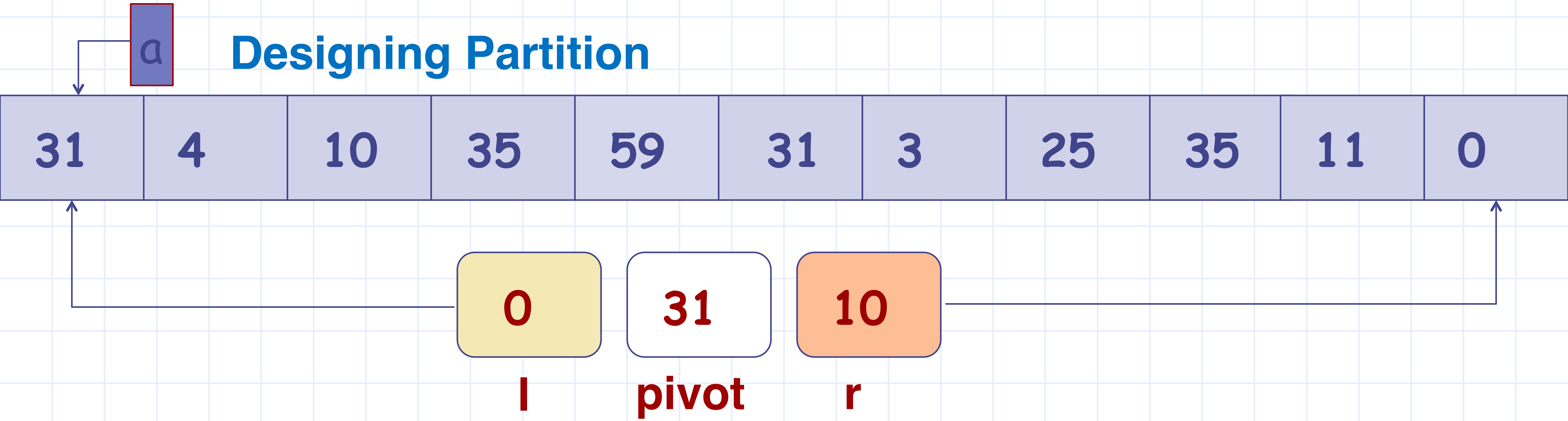
Note: Partition **DOES NOT** sort the array. It is “weaker” than sorting. But it is useful step towards sorting (useful for other problems also).

1. partition(int a[], int start, int end), pivot is a[0].
2. Partition re-arranges the array elements into two parts:
   a) the left part has all elements <= pivot
   b) the right part has all elements >= pivot
3. Partition can return either the first index of the right part or the last index of the left part. (Both answers would be acceptable).

Designing partition: Goal is to have **linear time complexity**, meaning that the number of comparisons and exchanges of items must be linear in the size.

Also, we will do partition **in place** – that is, without using extra arrays.

Designing Partition



31	4	10	35	59	31	3	25	35	11	0

1. Keep two integer variables denoting indices: l starts at the left end and r starts at the right end.
2. pivot is a[0] which is 31.
3. Value of pivot will not change during partition.

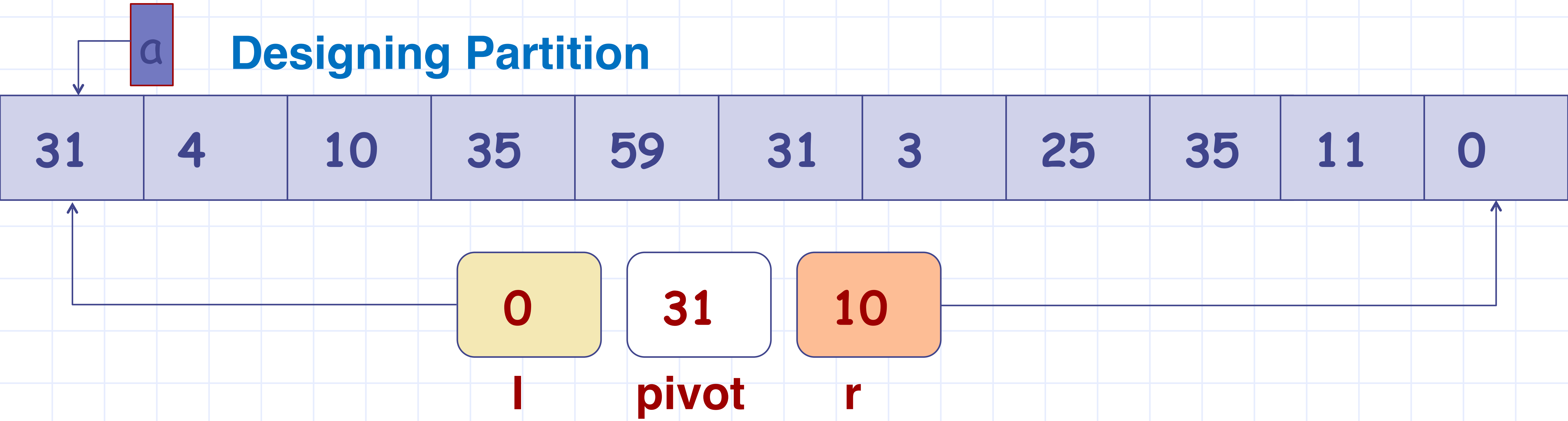
Basic Step in Partition Loop:

As long as a[l] < pivot, increment l by 1.

As long as a[r] > pivot, decrease r by 1.

If l < r, Exchange a[l] with a[r].
advance l by 1; decrement r by 1

Designing Partition



a										
31	4	10	35	59	31	3	25	35	11	0

l	pivot	r
0	31	10

Basic Step in Partition Loop:

As long as a[l] < pivot, increment l by 1.

As long as a[r] > pivot, decrease r by 1.

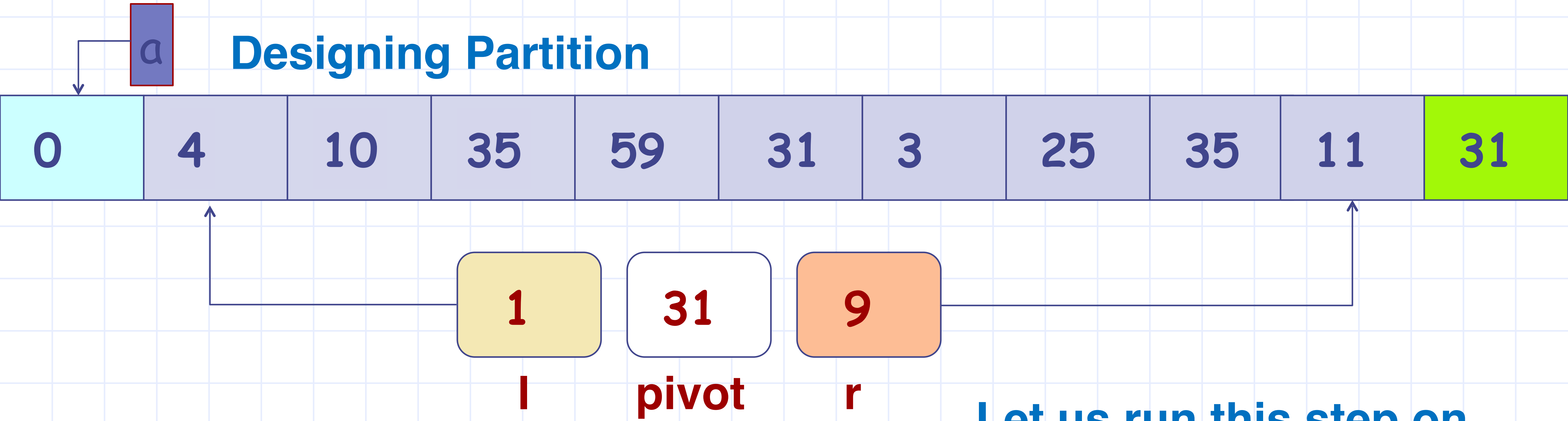
If l < r, Swap a[l] with a[r]. advance l by 1; decrement r by 1

Let us run this step on the above array

1. First loop terminates, with l as 0.
2. Second loop terminates immediately, with r as 10.

Now we swap a[0] with a[10]

Designing Partition



a

0	4	10	35	59	31	3	25	35	11	31

1	31	9
l	pivot	r

Let us run this step on the above array

Basic Step in Partition Loop:

As long as a[l] < pivot, increment l by 1.

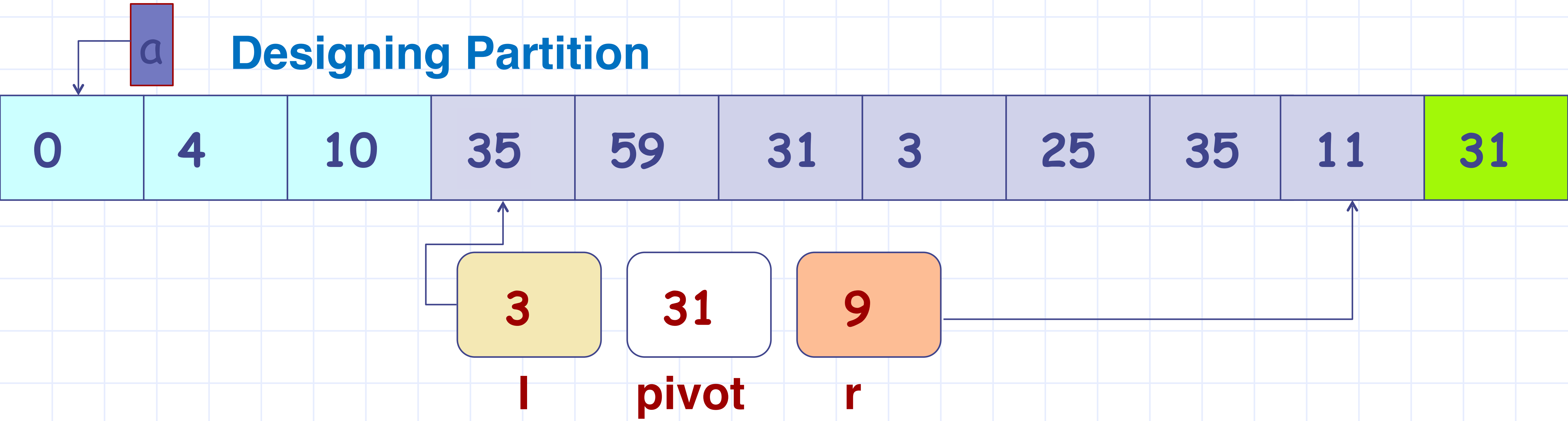
As long as a[r] > pivot, decrease r by 1.

If l < r, Swap a[l] with a[r]. advance l by 1; decrement r by 1

Swap and Advance

1. swap a[0] with a[10]
2. Advance l to 1
3. Decrement r to 9

Designing Partition



a

0	4	10	35	59	31	3	25	35	11	31

3	31	9
l	pivot	r

Basic Step in Partition Loop:

As long as a[l] < pivot, increment l by 1.

As long as a[r] > pivot, decrease r by 1.

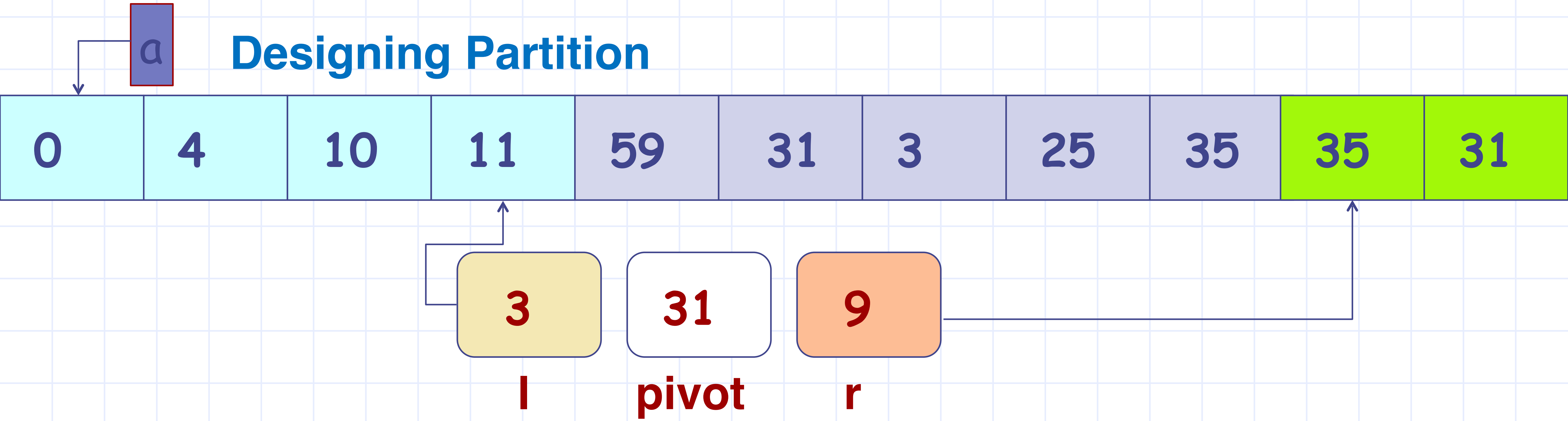
If l < r, Swap a[l] with a[r]. advance l by 1; decrement r by 1

Let us run this step on the above array

Invariant

1. a[0]…a[l-1] are all <= pivot.
2. a[r+1]…a[n-1] are all >= pivot.

Designing Partition



a

0	4	10	11	59	31	3	25	35	35	31

3

l

31

pivot

9

r

Basic Step in Partition Loop:

As long as a[l] < pivot, increment l by 1.

As long as a[r] > pivot, decrease r by 1.

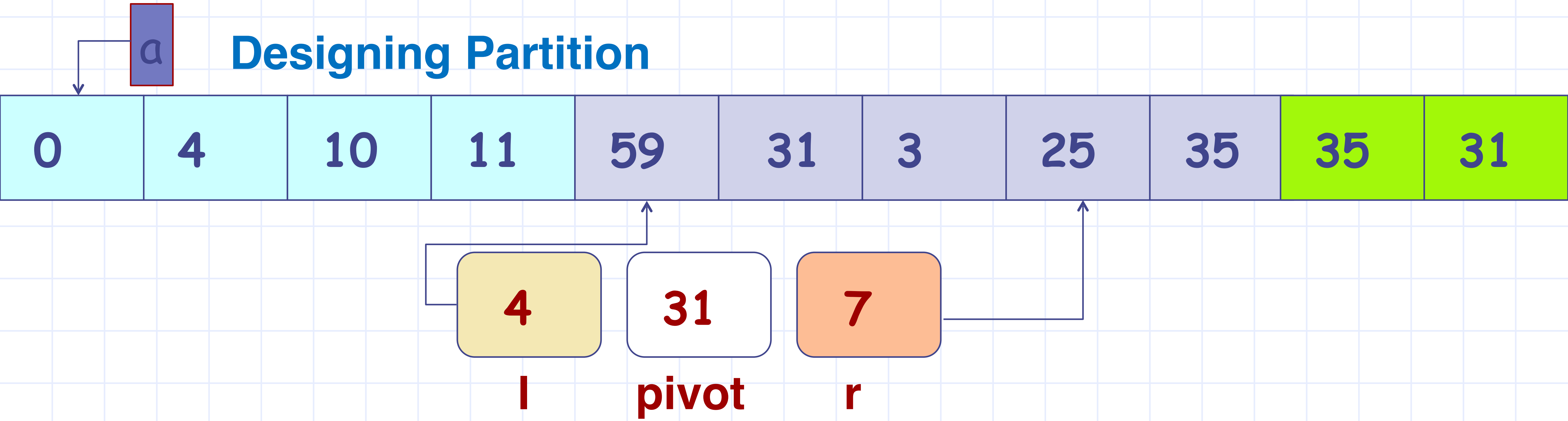
If l < r, Swap a[l] with a[r]. advance l by 1; decrement r by 1

Let us run this step on the above array

Invariant

1. a[0]…a[l-1] are all <= pivot.
2. a[r+1]…a[n-1] are all >= pivot.

Designing Partition



Basic Step in Partition Loop:

As long as a[l] < pivot, increment l by 1.

As long as a[r] > pivot, decrease r by 1.

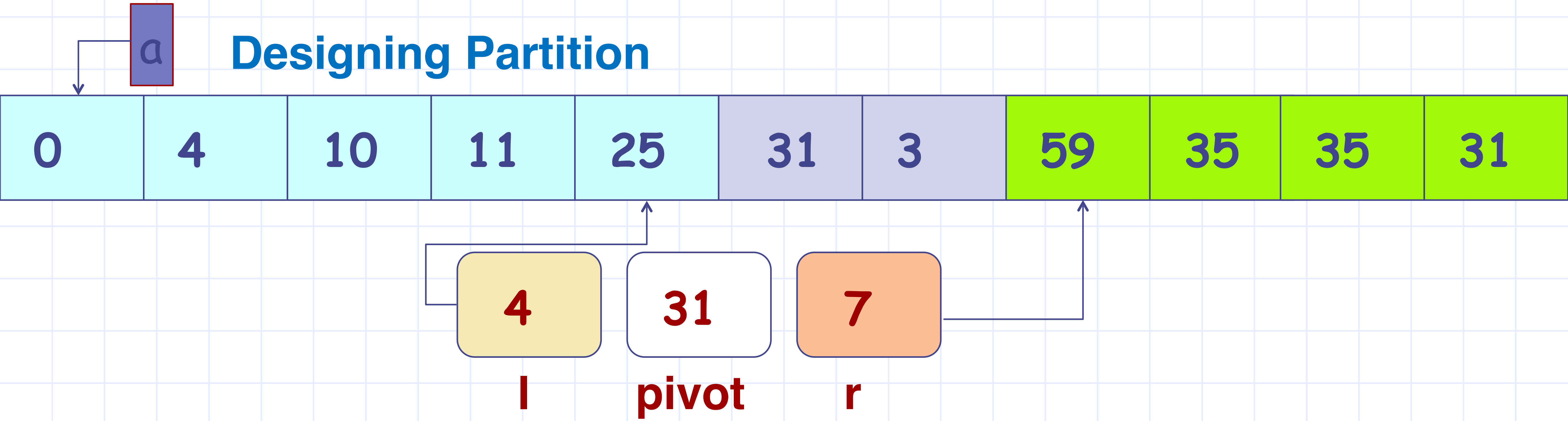
If l < r, Swap a[l] with a[r]. advance l by 1; decrement r by 1

Let us run this step on the above array

Invariant

1. a[0]…a[l-1] are all <= pivot.
2. a[r+1]…a[n-1] are all >= pivot.

Designing Partition



a

0	4	10	11	25	31	3	59	35	35	31

l	pivot	r
4	31	7

Basic Step in Partition Loop:

As long as a[l] < pivot, increment l by 1.

As long as a[r] > pivot, decrease r by 1.

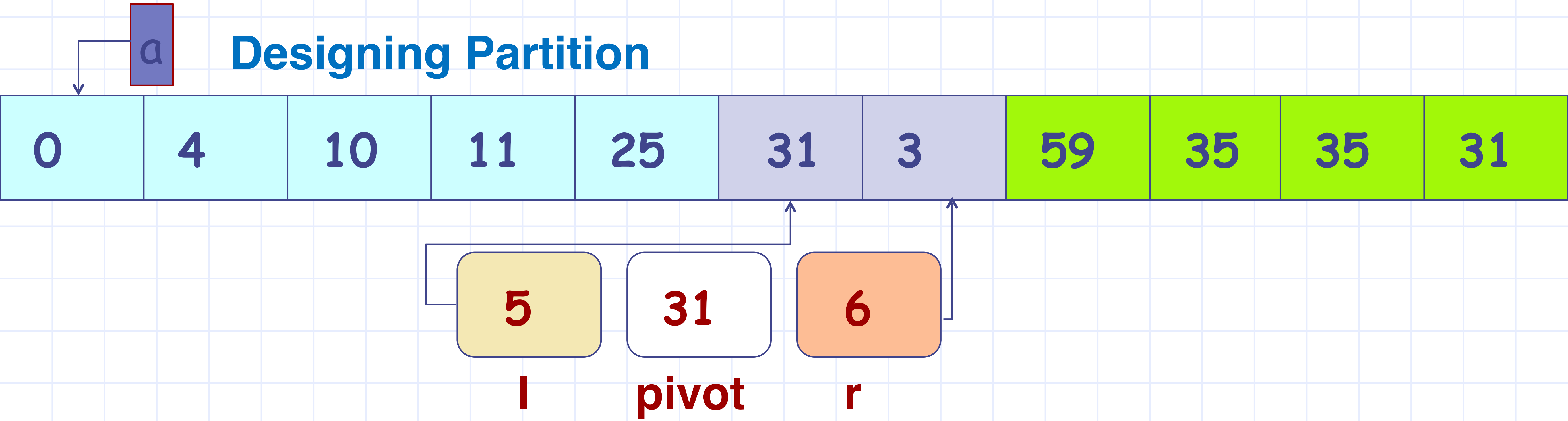
If l < r, Swap a[l] with a[r]. advance l by 1; decrement r by 1

Let us run this step on the above array

Invariant

1. a[0]…a[l-1] are all <= pivot.
2. a[r+1]…a[n-1] are all >= pivot.

Designing Partition



0	4	10	11	25	31	3	59	35	35	31

Basic Step in Partition Loop:

As long as a[l] < pivot, increment l by 1.

As long as a[r] > pivot, decrease r by 1.

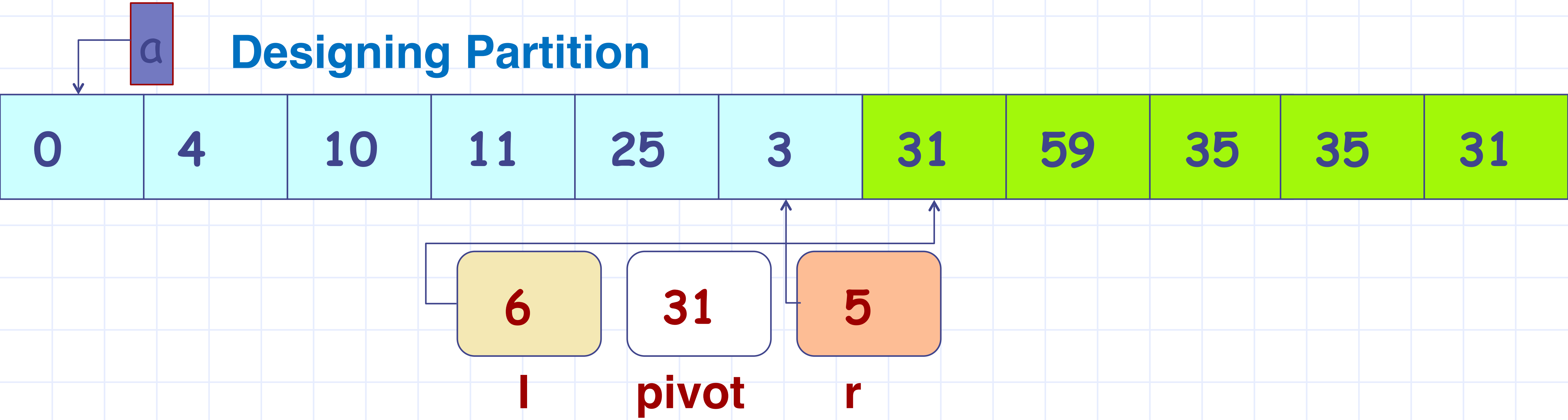
If l < r, Swap a[l] with a[r]. advance l by 1; decrement r by 1

Let us run this step on the above array

Invariant

1. a[0]…a[l-1] are all <= pivot.
2. a[r+1]…a[n-1] are all >= pivot.

Designing Partition



0	4	10	11	25	3	31	59	35	35	31

Basic Step in Partition Loop:

As long as a[l] < pivot, increment l by 1.

As long as a[r] > pivot, decrease r by 1.

If l < r, Swap a[l] with a[r]. advance l by 1; decrement r by 1

Let us run this step on the above array

Invariant

1. a[0]…a[l-1] are all <= pivot.
2. a[r+1]…a[n-1] are all >= pivot.

```
int partition(int a[], int start, int end) {
        int l = start, r = end, pivot = a[l];
        while (l <=end && r>=start) {
            while (a[l] < pivot && l <end) { l=l+1; }
                   while (a[r] > pivot && r>start) { r=r-1; }
                if(l>=r)  return r;
                  else {
                     swap(a, l, r);
                     l = l+1; r = r-1;
                }
         }
}
```

The Partition function

We designed a function int partition(int a[], int start, int end) that returns an index pindex of the array a[] such that the following are true.

1. all items in a[start, …, pindex] are <= pivot,
2. all items in a[pindex+1,…,end] are >= pivot,
3. Number of operations required by partition is O(n), that is bounded by c n for some constant c. Required only a single pass over the array: each element is touched once.

Pivoting choices

Pivot may be chosen to be any value of a[]. Some choices are

1. Pivot is a[0]: simple choice.
2. Pivot is some random member of a[]: randomized pivot choice.
3. Pivot is the median element of a[]. This gives the most equal sized partitions, but is much more complicated.

Sorting

After the call **pindex = partition(a,start,end)**

1. each element of a[start,…,pindex] <= pivot.
2. each element of a[pindex+1,…,end] >= pivot.

Suppose we wish to sort the array a[].

To sort a[], we can sort the left partition and the right partition independently.

1. sort the array a[0,…,pindex], and,
2. sort the array a[pindex+1,…,n-1].

How should we do the sorting?
Any way we wish, but… how about choosing the same algorithm, that is, run partition on each half again (and then again on smaller parts—this is recursion)

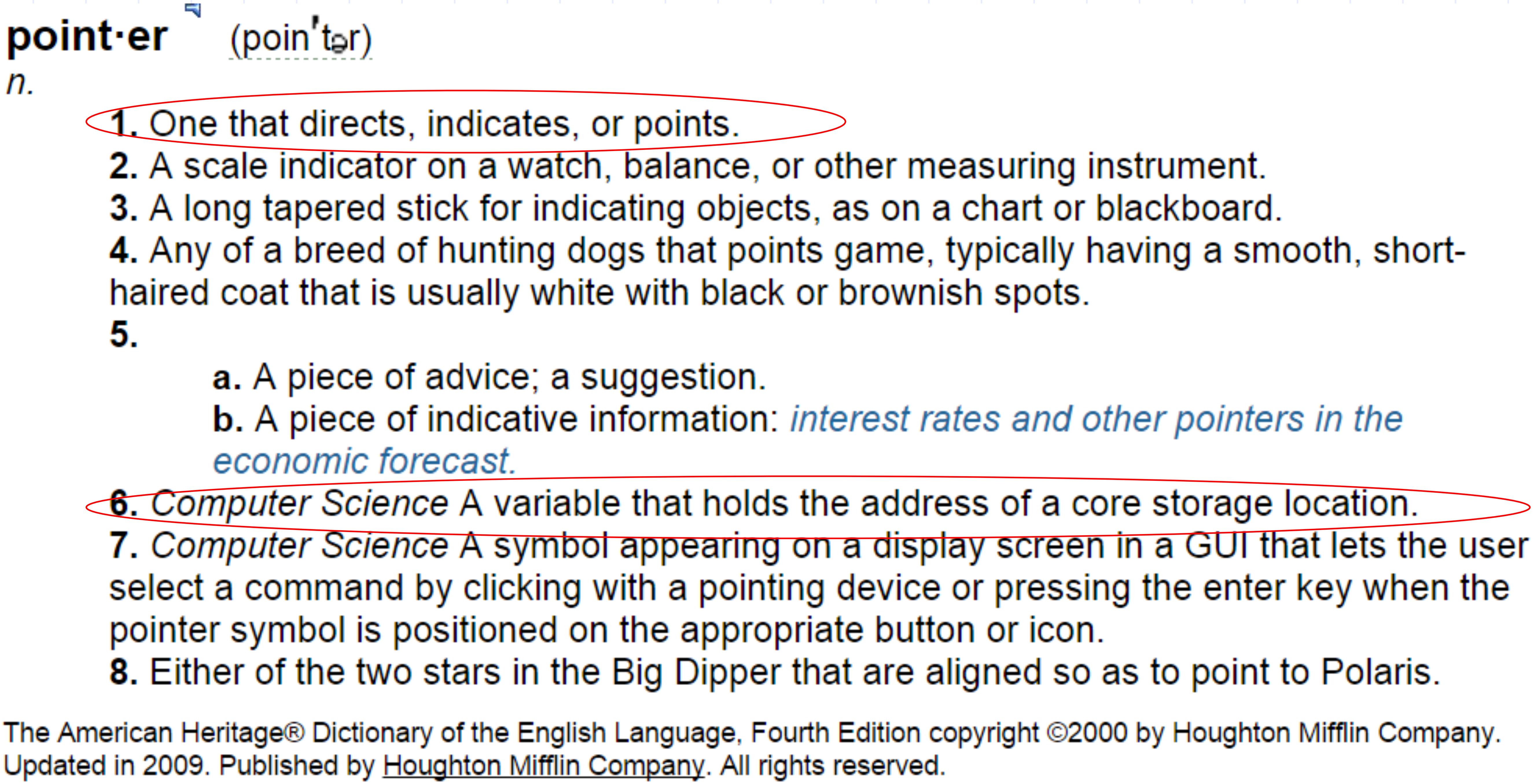
QuickSort

```
void qsort(int a[], int start, int end) {
     int pindex;
     if (start >=end) return;  /* nothing to sort */
     else {
        pindex = partition(a, start, end);
        qsort(a, start, pindex);
        qsort(a, pindex+1, end);
   }
}
```

ESC101: Introduction to Computing

Pointers

Pointer: Dictionary Definition

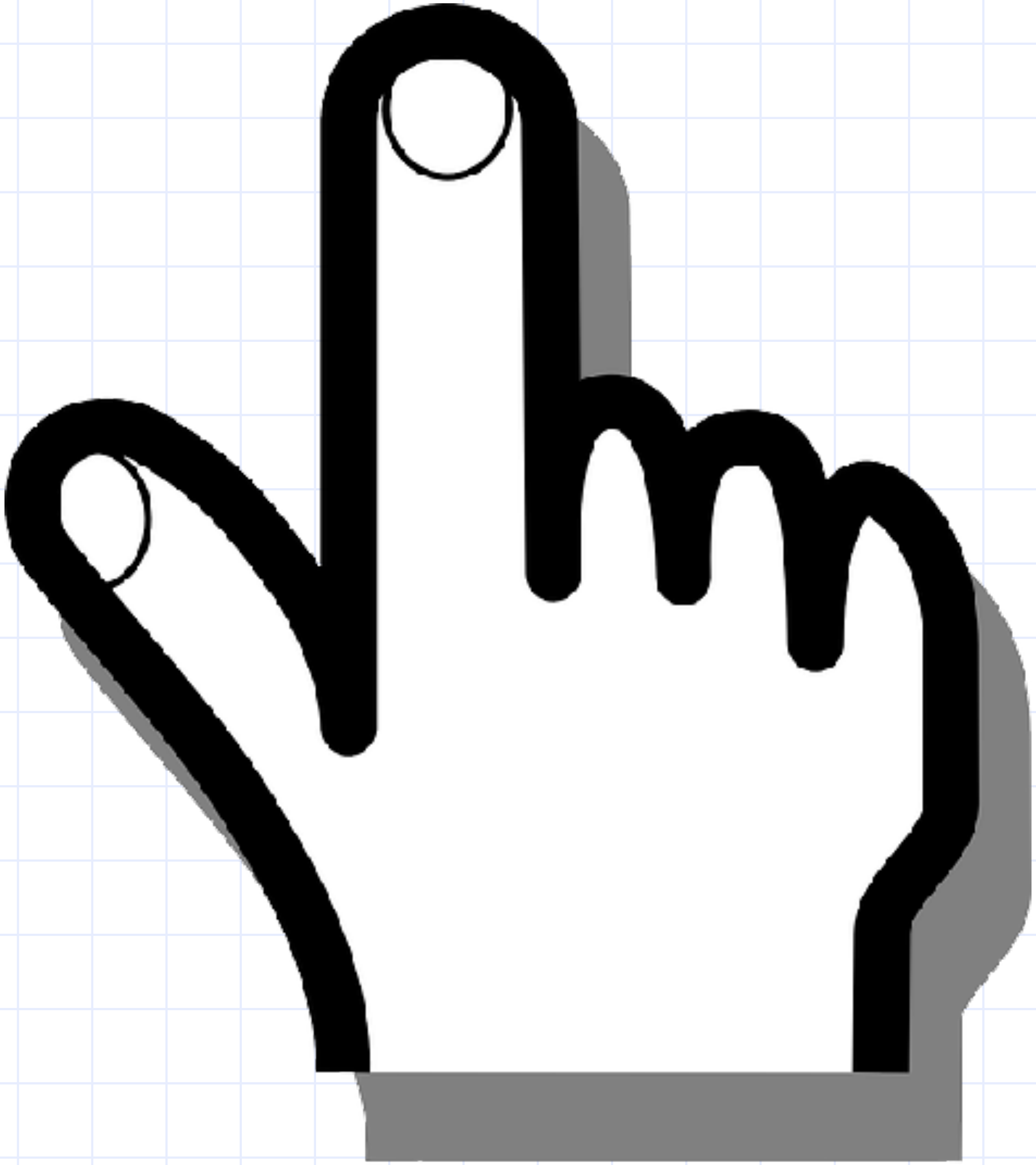


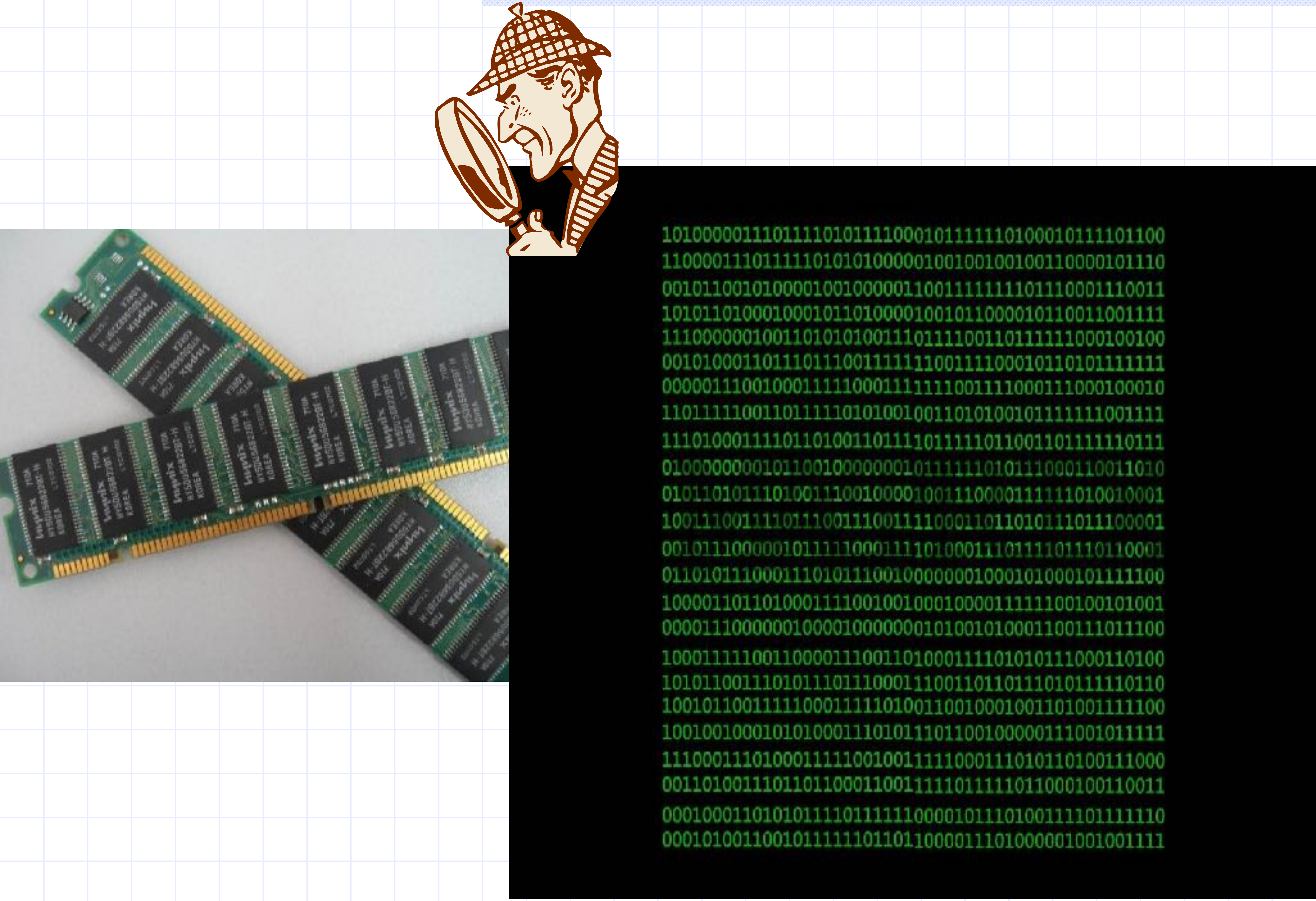
point·er (poin′tər)
n.

1. One that directs, indicates, or points.
2. A scale indicator on a watch, balance, or other measuring instrument.
3. A long tapered stick for indicating objects, as on a chart or blackboard.
4. Any of a breed of hunting dogs that points game, typically having a smooth, short-haired coat that is usually white with black or brownish spots.
5.
  a. A piece of advice; a suggestion.
  b. A piece of indicative information: *interest rates and other pointers in the economic forecast.*
6. *Computer Science* A variable that holds the address of a core storage location.
7. *Computer Science* A symbol appearing on a display screen in a GUI that lets the user select a command by clicking with a pointing device or pressing the enter key when the pointer symbol is positioned on the appropriate button or icon.
8. Either of the two stars in the Big Dipper that are aligned so as to point to Polaris.

The American Heritage® Dictionary of the English Language, Fourth Edition copyright ©2000 by Houghton Mifflin Company. Updated in 2009. Published by Houghton Mifflin Company. All rights reserved.

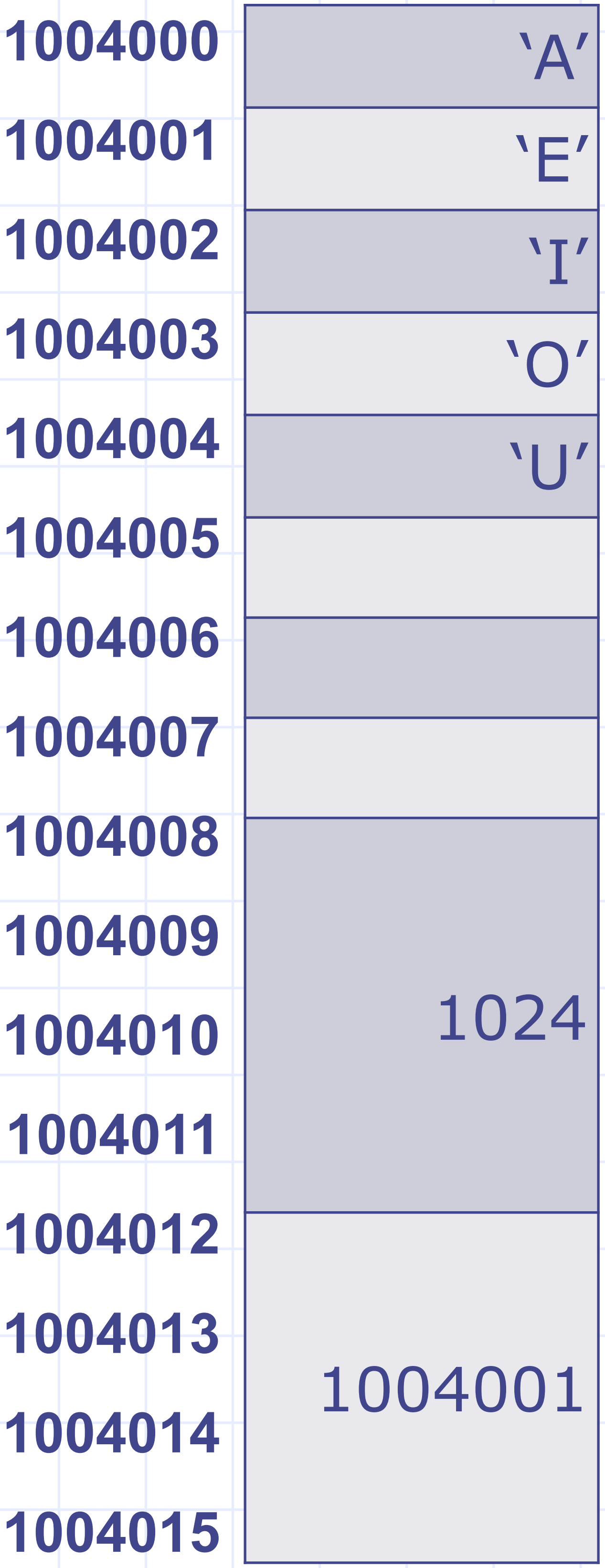
Pointer we are all born with





Simplified View of Memory

- "Array" of blocks
- Each block can hold a byte (8-bits)
- "char" stored in 1 block
- "int" (32-bit) stored in 4 consecutive blocks
- Finite number of blocks
  - Limited by the capacity of (Virtual) Memory
  - Blocks are addressable – [0… 2^N-1]



1004000	'A'
1004001	'E'
1004002	'I'
1004003	'O'
1004004	'U'
1004005	
1004006	
1004007	
1004008	1024
1004009	
1004010	
1004011	
1004012	1004001
1004013	
1004014	
1004015	

Simplified View of Memory

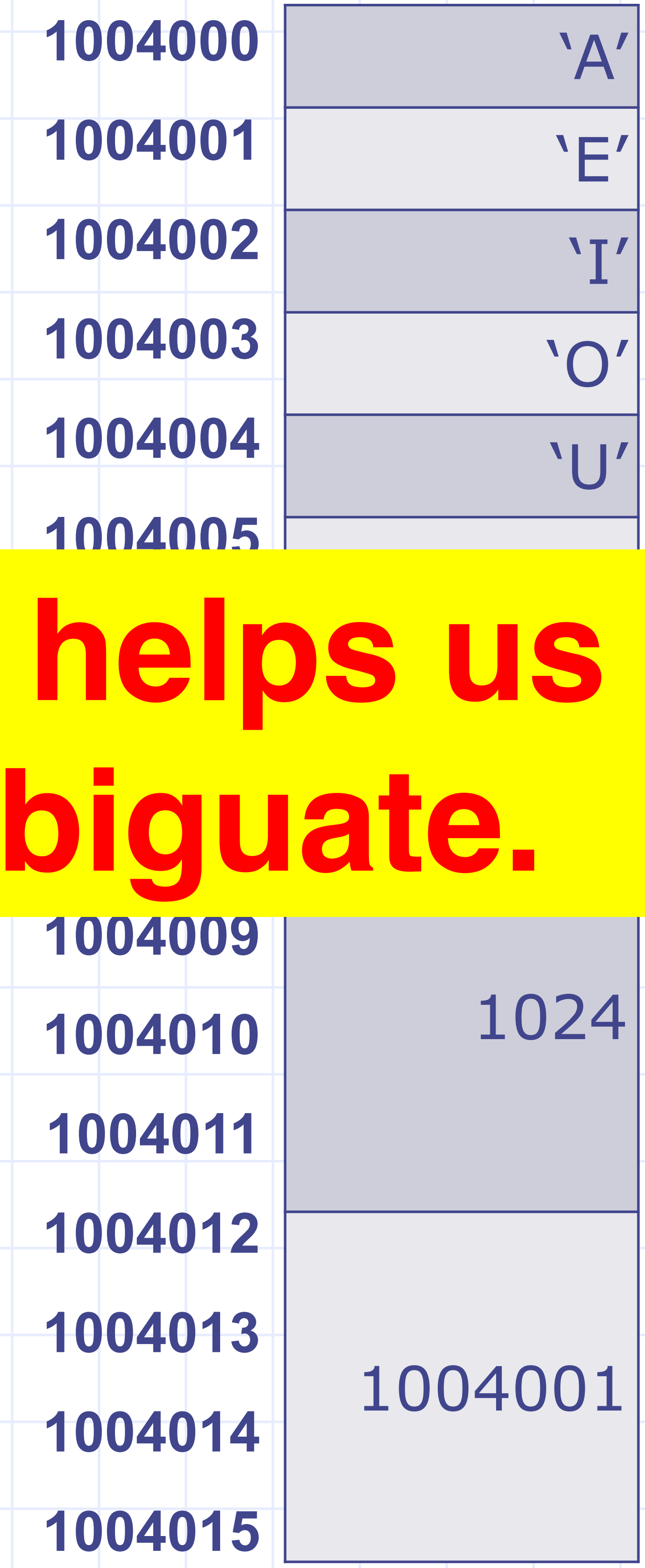
- Blocks are addressable.
- Address range: [0…2^N-1]
- N is the number of bits in address (number of digits in binary world)
- Any integer in the above range
  - Can be used as an index in the MEMORY ARRAY
- Since memory array is unique, we can use this index alone
  - If context is clear

1004000	‘A’
1004001	‘E’
1004002	‘I’
1004003	‘O’
1004004	‘U’
1004005	
1004006	
1004007	
1004008	1024
1004009	
1004010	
1004011	
1004012	1004001
1004013	
1004014	
1004015	

Simplified View of Memory

- Content of the 4-blocks starting at address 1004012
  ✓1004001
- Without knowing the context it is not possible to determine t significance of number
  ✓It could be an integer value 1004001
  ✓It could be the “location” of the block that stores ‘E’

How do we decide what it is?



1004000	‘A’
1004001	‘E’
1004002	‘I’
1004003	‘O’
1004004	‘U’
1004005	
1004009	1024
1004010	
1004011	
1004012	1004001
1004013	
1004014	
1004015	

“Type” helps us disambiguate.