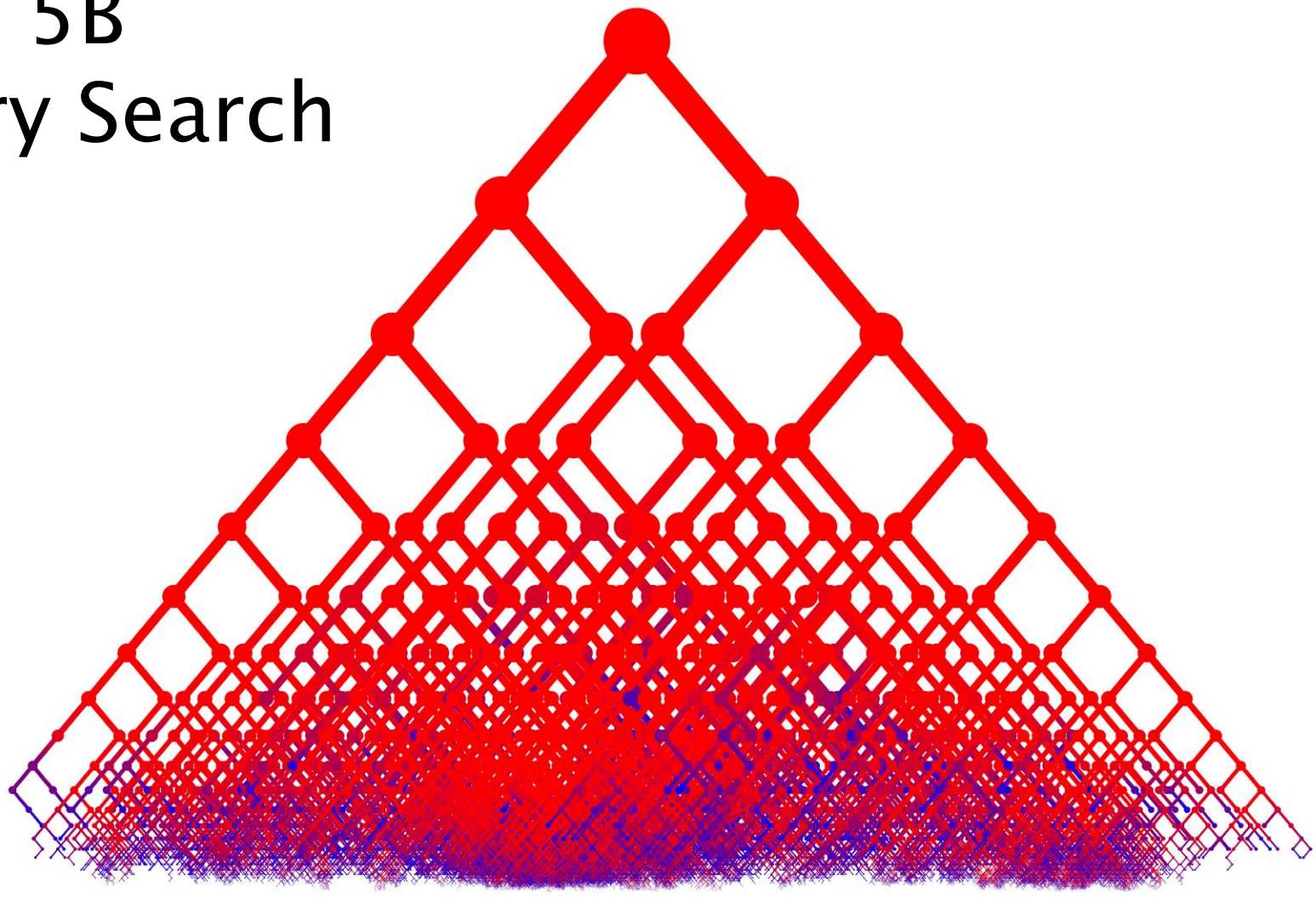
UNIT 5B
Binary Search



Course Announcements

- PA 5 due Monday Night
- PS 5 due Tuesday noon
  (preferably 9am in class)
- Exam Wednesday
  Bring your CMU ID

This Lecture

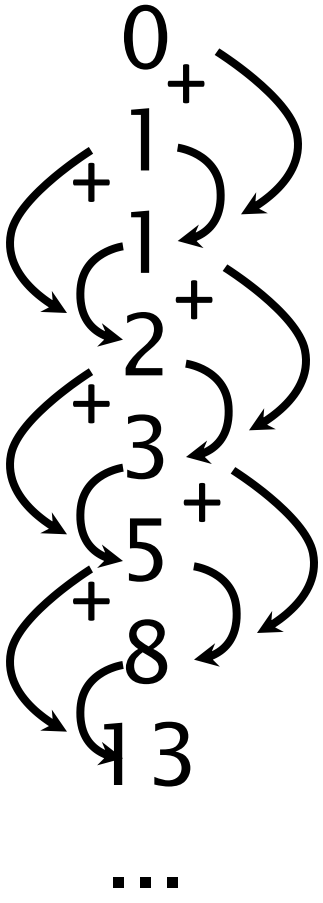
- A new search technique for lists called binary search
- Application of recursion to binary search
- Logarithmic worst-case complexity

but first, return to Previous slides on

FIBONACCI NUMBERS

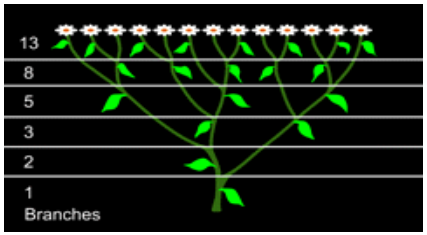
Fibonacci Numbers

A sequence of numbers:

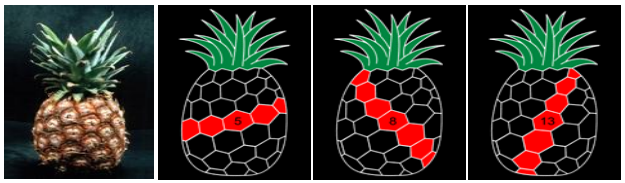


Fibonacci Numbers in Nature

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, etc.
- Number of branches on a tree, petals on a flower, spirals on a pineapple.
- Vi Hart's video on Fibonacci numbers (http://www.youtube.com/watch?v=ahXIMUkSXX0)








Recursive Definition

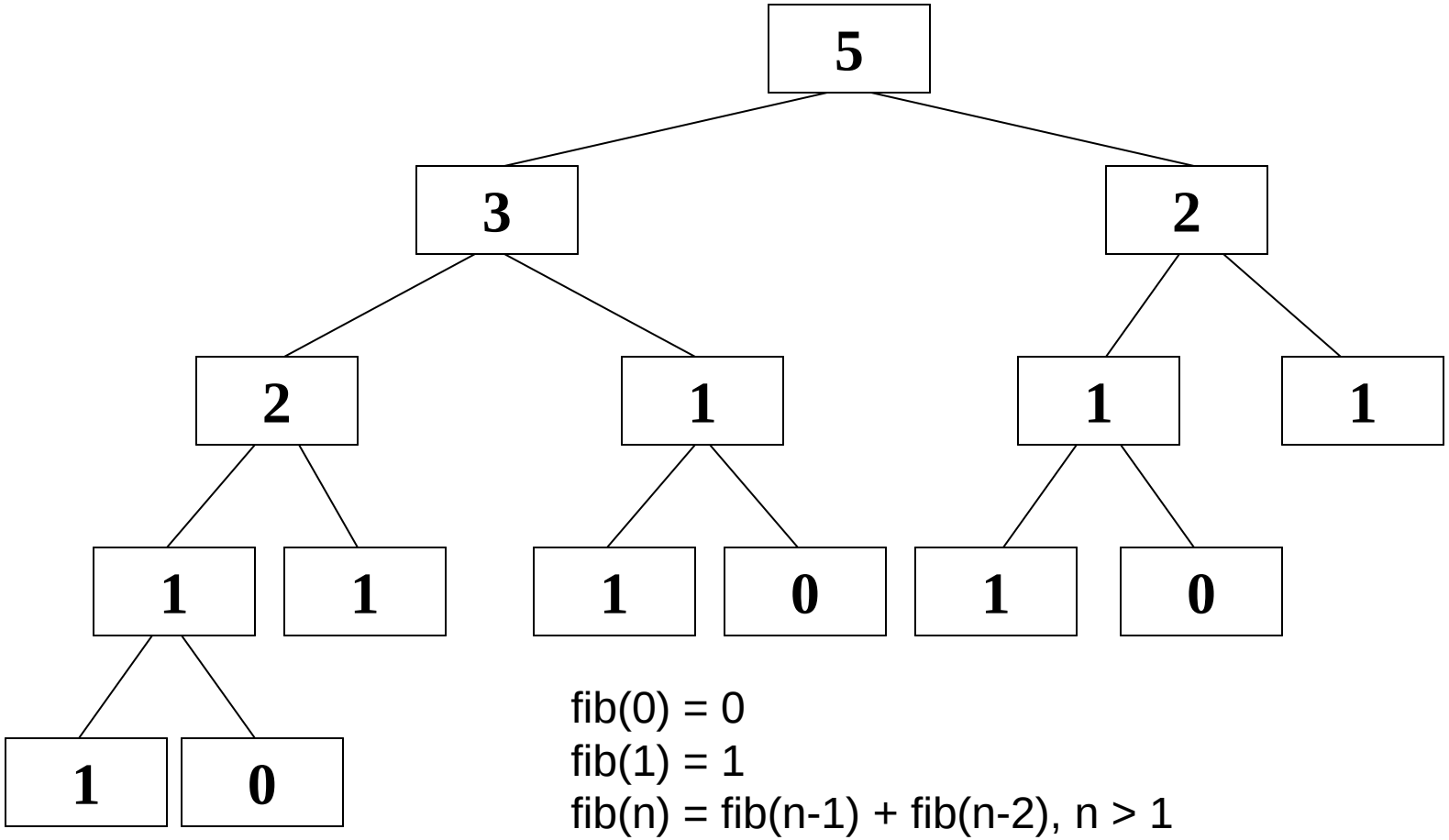
- Let fib(n) = the nth Fibonacci number, n ≥ 0
  - fib(0) = 0 (base case)
  - fib(1) = 1 (base case)
  - fib(n) = fib(n-1) + fib(n-2), n > 1

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, etc

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Two recursive calls!

Recursive Call Tree



fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1

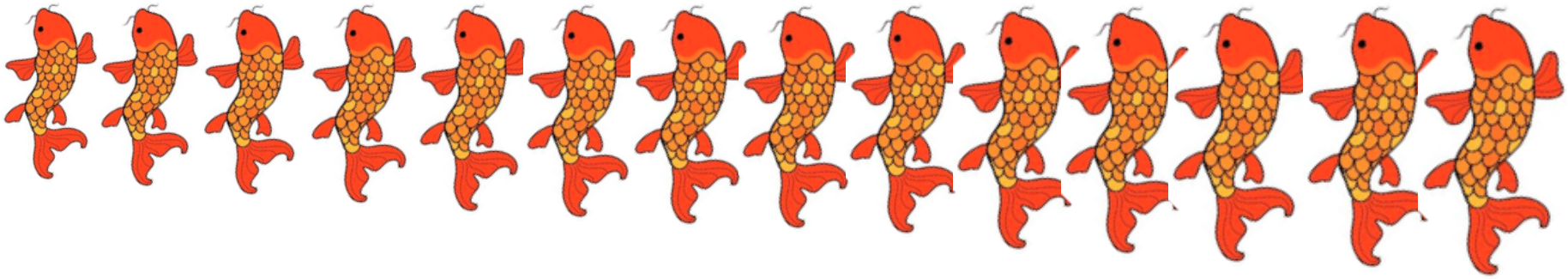
the idea

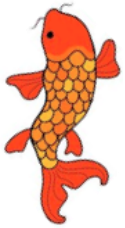
BINARY SEARCH

Number guessing

- I'm thinking of a number between 1 and 16
- You get to ask me, yes or no, is it greater than some number you choose
- How many questions do you need to ask?
- Which questions will you ask to get the answer quickest?

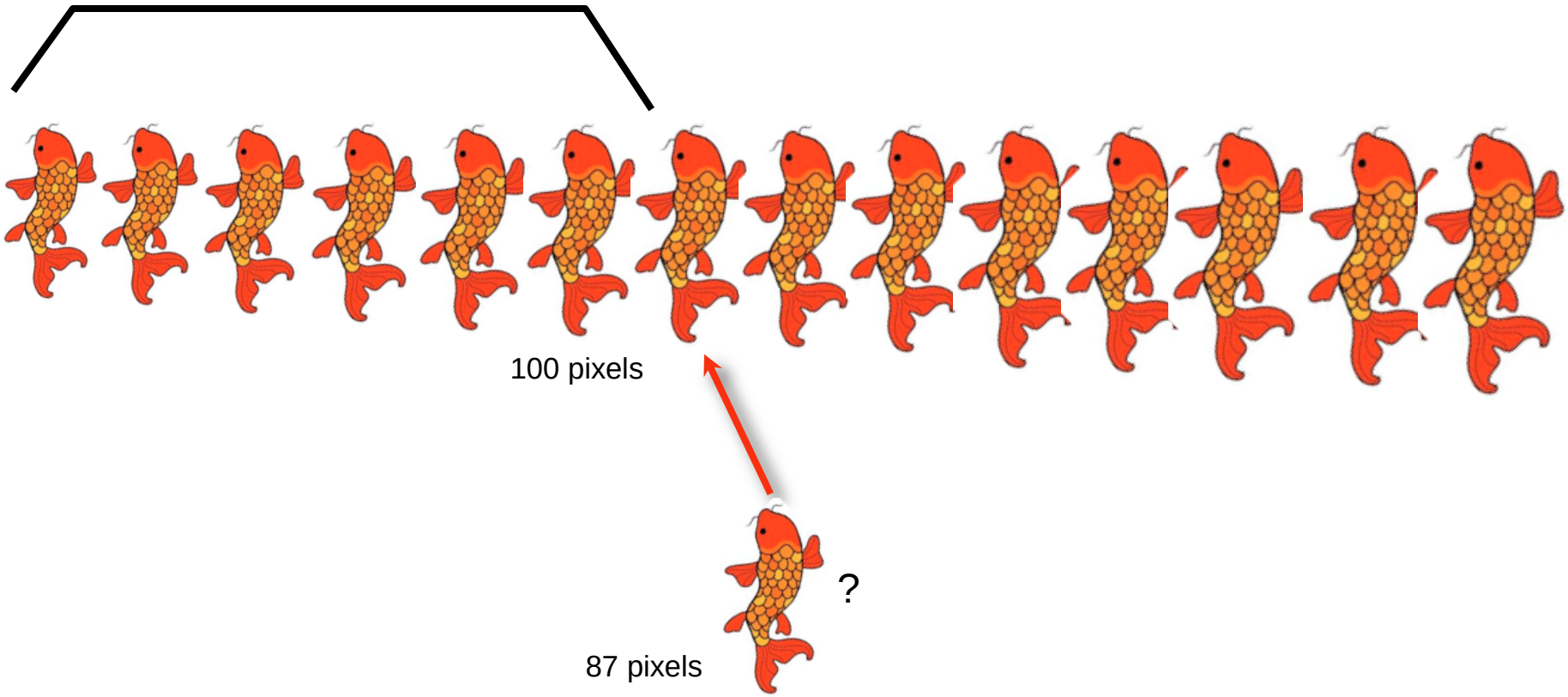
Binary Search in an **Ordered List**



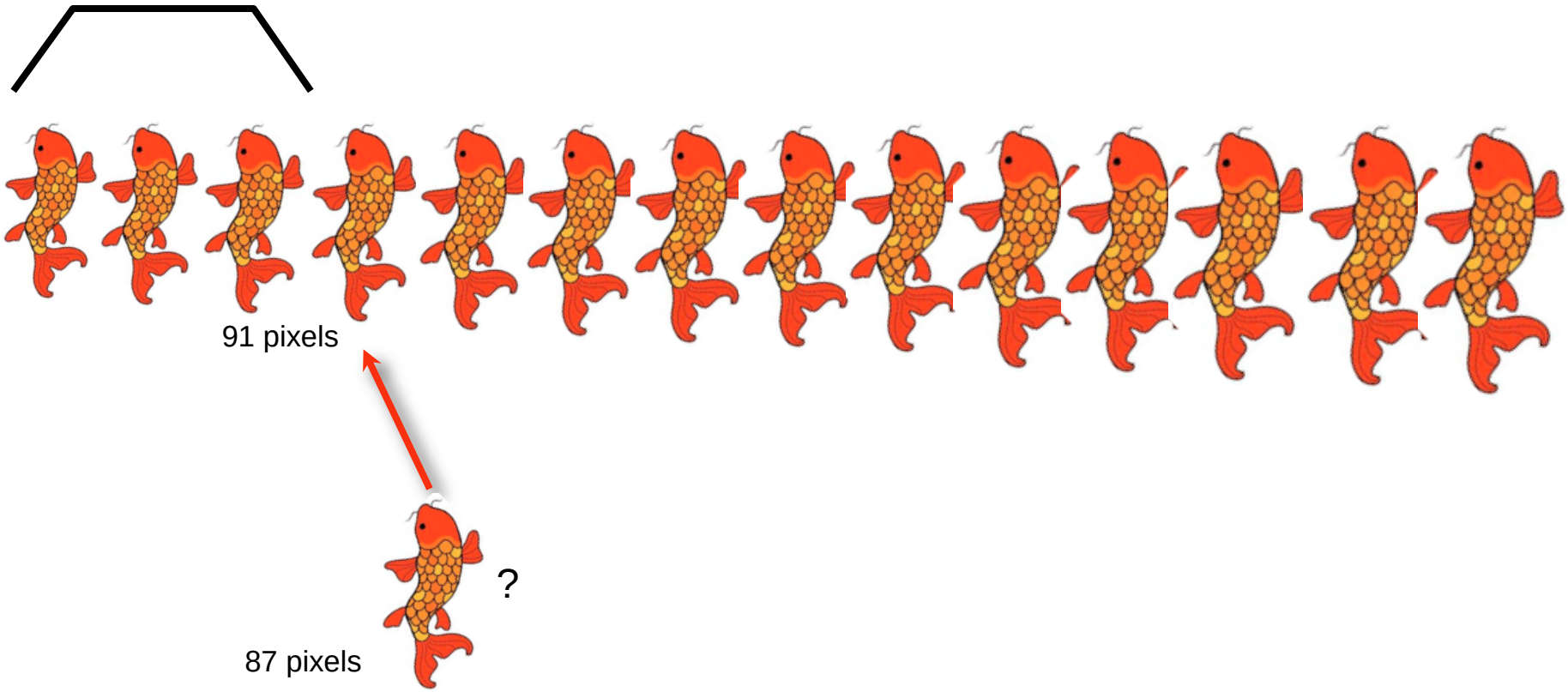


?

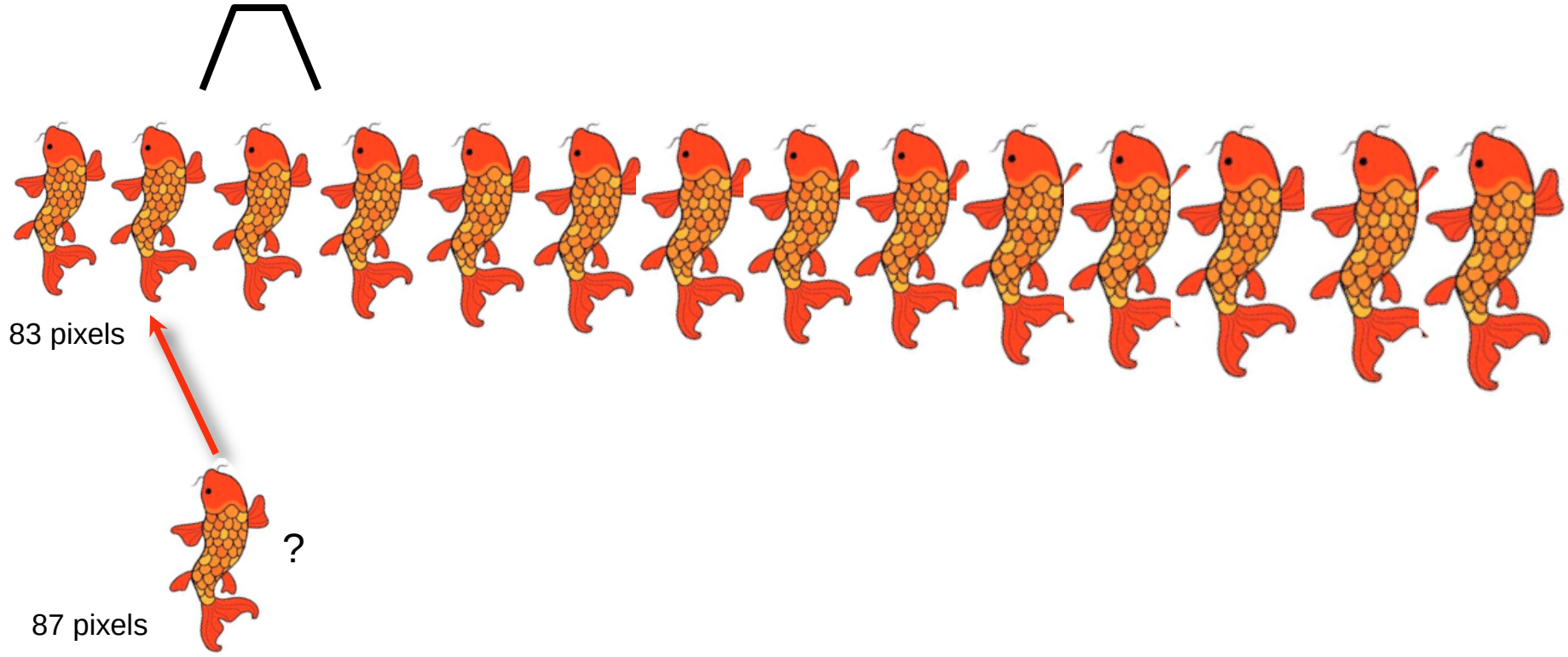
Binary Search in an Ordered List



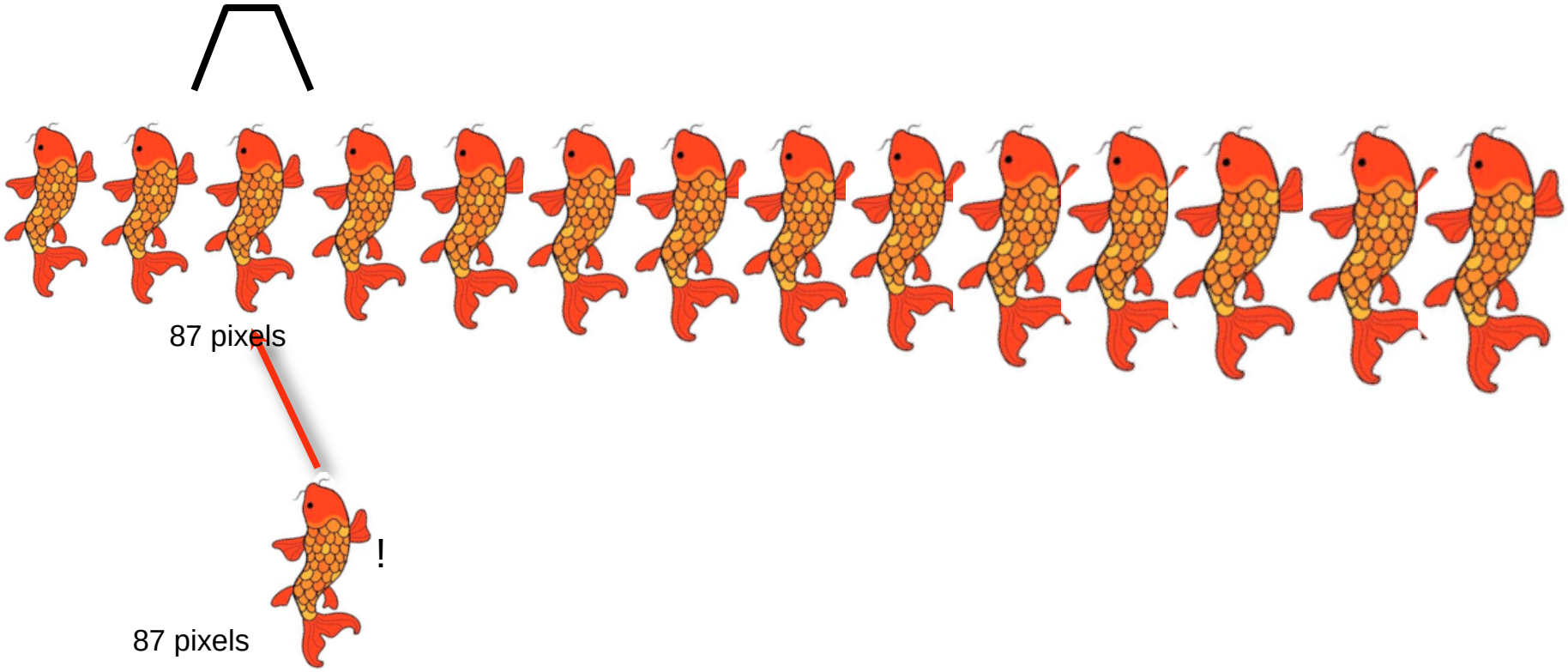
Binary Search in an Ordered List



Binary Search in an Ordered List



Binary Search in an Ordered List



from idea to

ALGORITHM

Specification: the Search Problem

- Input:
  A list of n unique elements and a key to search for
  - **The elements are <u>sorted</u> in increasing order.**

- Result:
  The index of an element matching the key, or **None** if the key is not found.

Recursive Algorithm

BinarySearch (list, key):

1. Return **None** if the list is **empty**.
2. Compare the **key** to the **middle** element of the list
3. Return the **index** of the middle element if they match
4. If the key is less than the middle element then
   return BinarySearch(first half of list, key)
   Otherwise,
   return BinarySearch(second half of list,key).

Example 1: Search for 73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	**25**	**32**	**37**	**41**	**48**	**58**	**60**	**66**	**73**	**74**	**79**	**83**	**91**	**95**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	**66**	**73**	**74**	**79**	**83**	**91**	**95**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	**66**	**73**	**74**	79	83	91	95

Found: return 9

Example 2: Search for 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	**25**	**32**	**37**	**41**	**48**	**58**	**60**	**66**	**73**	**74**	**79**	**83**	**91**	**95**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	**25**	**32**	**37**	**41**	**48**	**58**	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	**41**	**48**	**58**	60	66	73	74	79	83	91	95

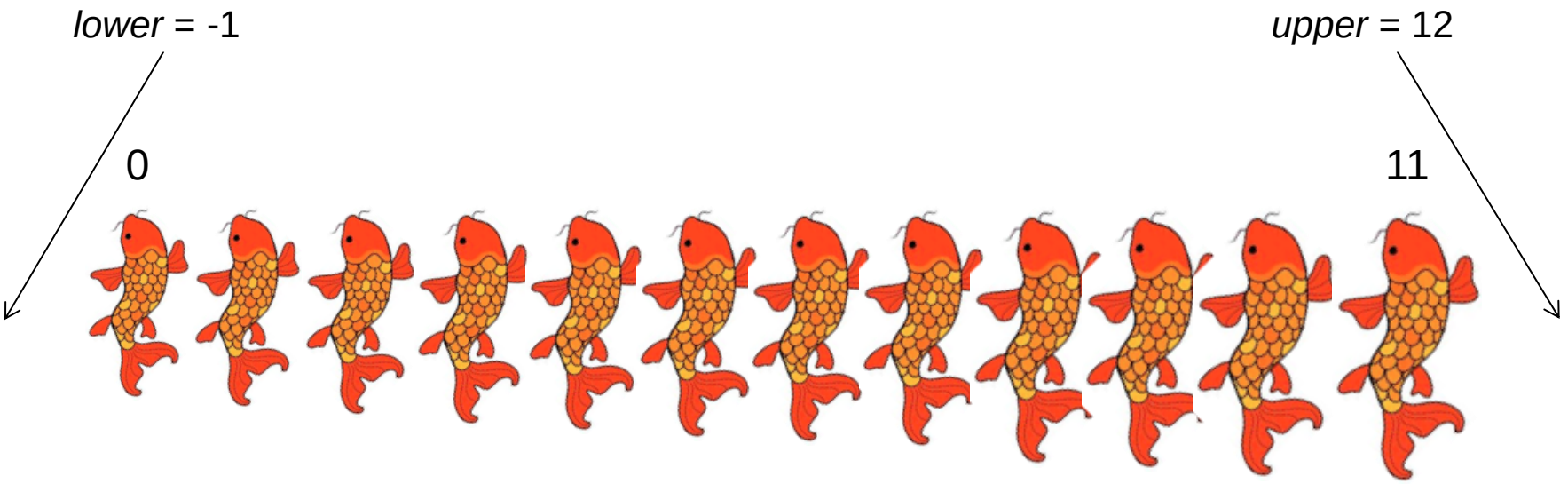
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	**41**	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

Not found: return None

Controlling the range of search

- Maintain three numbers: *lower*, *upper*, *mid*
- Initially *lower* is -1, *upper* is length of the list



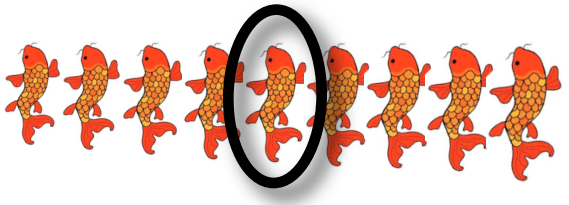
Controlling the range

- mid is the midpoint of the range:

  mid = (lower + upper) // 2 (integer division)

lower = -1, *upper* = 9 (range has 9 elements)

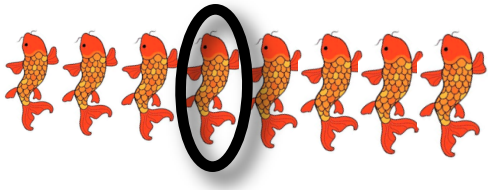
mid = 4



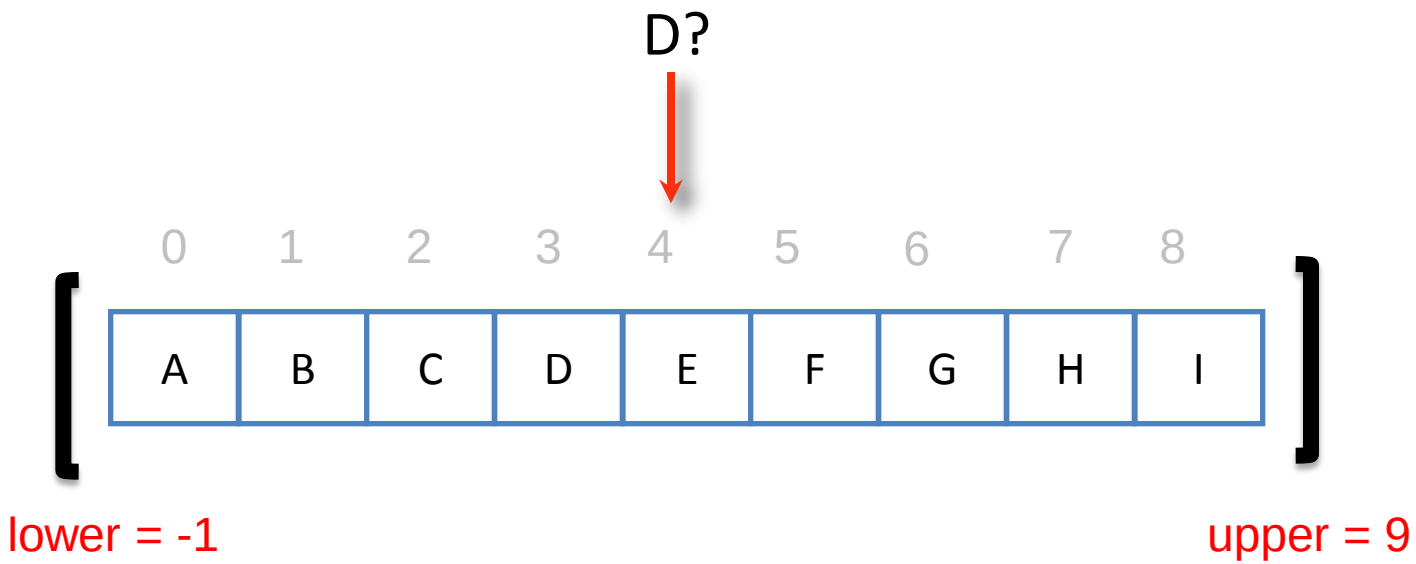
What happens if the range has an even number of elements?

lower = -1, *upper* = 8

mid = 3

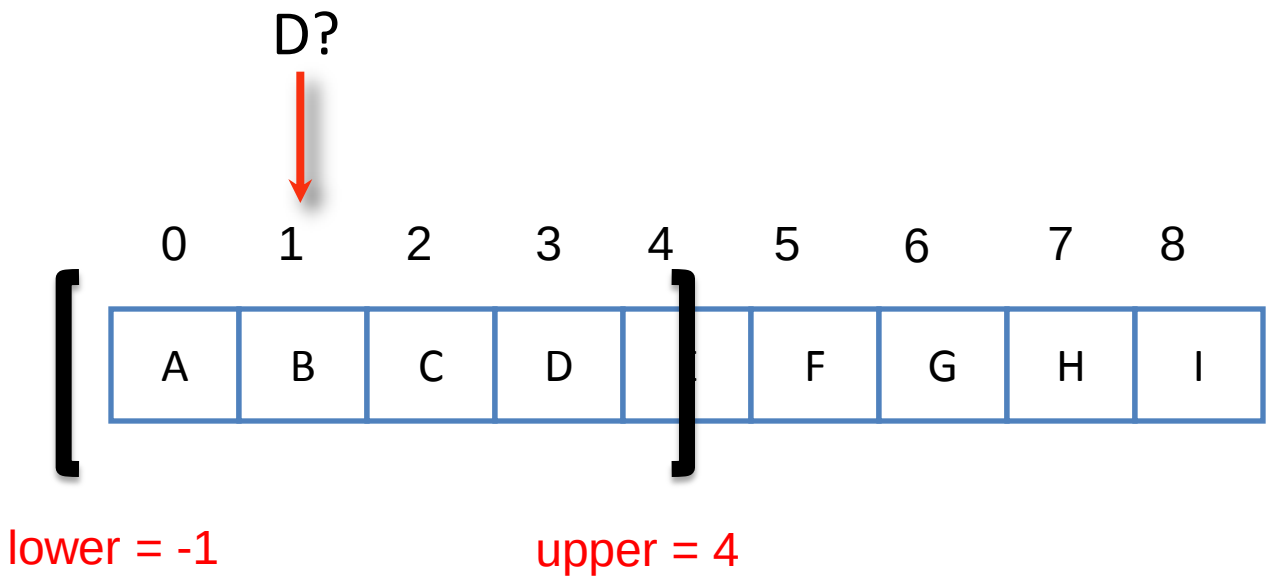


Example



List is sorted in ascending order.
Suppose we are searching for D.

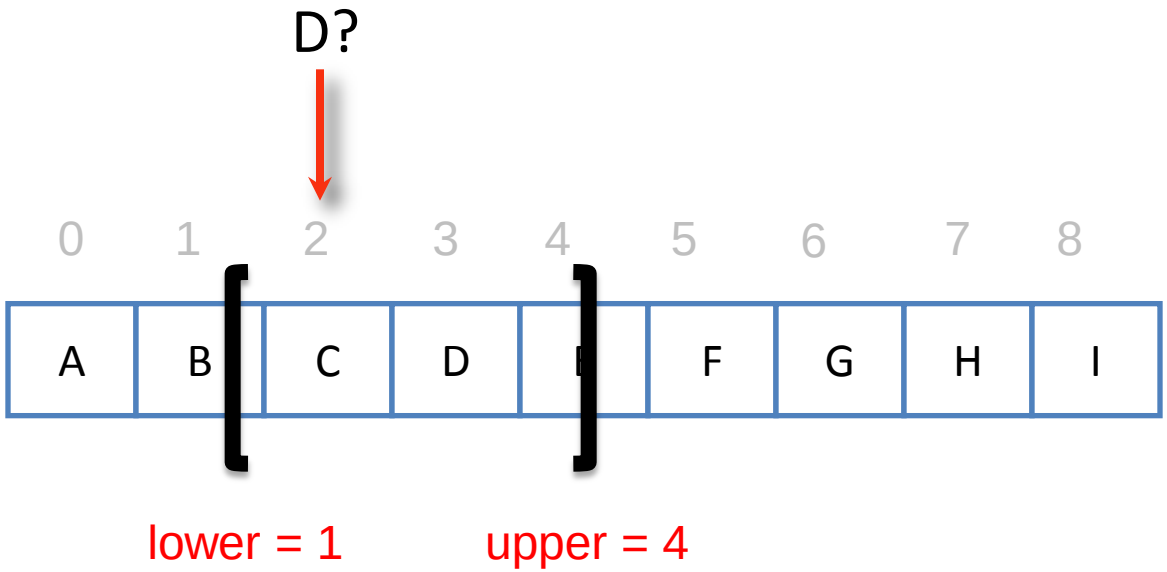
Example



Each time we

look at a smaller portion of the list within the window and ignore all the elements outside of the window

Example

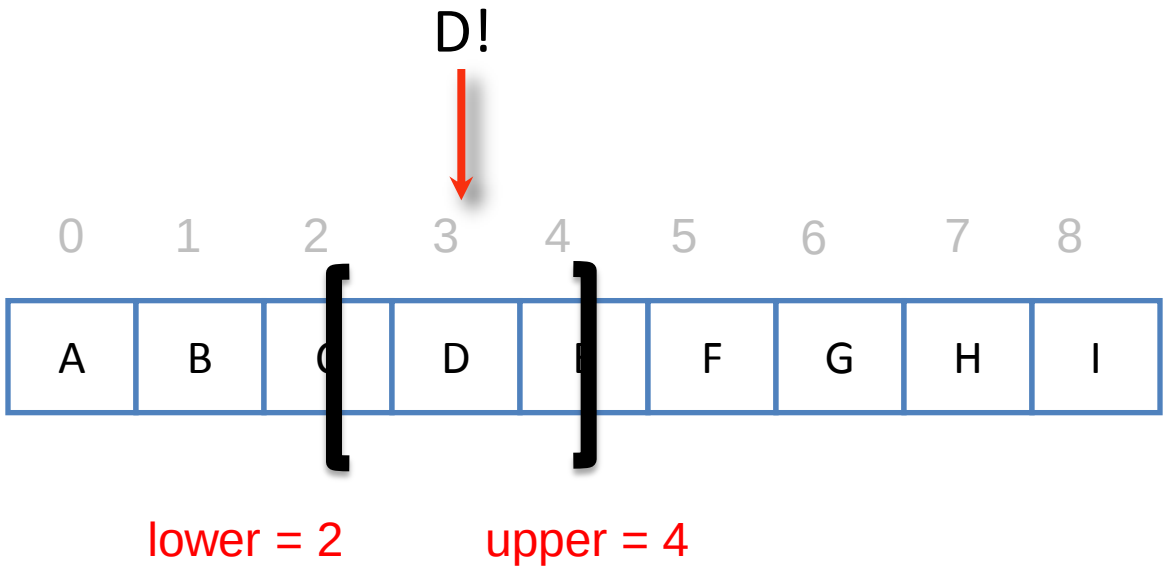


Each time we

look at a smaller portion of the list within the window

and ignore all the elements outside of the window

Example



Each time we

look at a smaller portion of the list within the window

and ignore all the elements outside of the window

towards a Python program:

DESIGNING THE RECURSION

Base case: range empty

- How do we determine if the range is empty?
  - *lower* + 1 == *upper*
- What should we return then?
  - None

Base case: key found

- The key is compared to the element at *mid:*
  - *list[mid] == key*
- What should we return then?
  - *mid*

Recursive Case

- Non-empty range:
  What subproblem(s) should we solve?
  - search left half or search right half

- What should we return then?
  - result of searching left or right half

- New value for *lower?* value for *upper?*
  - left half : *lower*, *mid*
  - right half : *mid*, *upper*

Parameters for recursion

- **Inputs:** *key* and *list of items*
- But we also need *lower* and *upper* bounds
  - since they change throughout the search, they have to be parameters of the search function
- **Design:** *main function* and *recursive helper function*

Recursive Binary Search in Python

```
# main function
def bsearch(items, key):
    return bs_helper(items, key, -1, len(items))

# recursive helper function
def bs_helper(items, key, lower, upper):
    if lower + 1 == upper: # Base case: empty
        return None
    mid = (lower + upper) // 2 # Recursive case
    if key == items[mid]:
        return mid
    if key < items[mid]: # Go left
        return bs_helper(items, key, lower, mid)
    else:                # Go right
        return bs_helper(items, key, mid, upper)
```

first value for *upper*

first value for *lower*

new value for *upper*

same value for *lower*

new value for *lower*

same value for *upper*

Caveat: specification

- The algorithm and the code was developed on the assumption that the input list is sorted.

- If the function is called with an unsorted list it has no obligation to behave correctly.

reflections

MEASUREMENT AND ANALYSIS

Trace: Search for 73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

```
                        key lower upper
bs_helper(items, 73,   -1,    15)
                             mid = 7 and 73 > items[7]
bs_helper(items, 73,    7,    15)
                             mid = 11 and 73 < items[11]
bs_helper(items, 73,    7,    11)
                             mid = 9 and 73 == items[9]
                             --->  return 9
```

Trace: Search for 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

	key	lower	upper	
bs_helper(items,	42,	-1,	15)	
				mid = 7 and 42 < items[7]
bs_helper(items,	42,	-1,	7)	
				mid = 3 and 42 > items[3]
bs_helper(items,	42,	3,	7)	
				mid = 5 and 42 < items[5]
bs_helper(items,	42,	3,	5)	
				mid = 4 and 42 > items[4]
bs_helper(items,	73,	4,	5)	
				lower + 1 == upper
				---> Return None.

Instrumenting Binary Search Code

```
count = 0 # count of comparisons

def bsearch(list, key):
    global count
    count = 0
    print("Searching list of length ", len(list))
    result = bs_helper(list, key, -1, len(list))
    print("Number of comparisons:", count)
    return result

def bs_helper(list, key, lower, upper):
    global count
    if lower + 1 == upper:
        print("Not found")
        return None
    mid = (lower + upper) // 2
    print("mid:", mid, "lower:", lower, "upper", upper)
    count = count + 1
    if key == list[mid]:
        return mid
    if key < list[mid]:
        return bs_helper(list, key, lower, mid)
    else:
        return bs_helper(list, key, mid, upper)
```

Instrumented Output

```
>>> bsearch(list(range(1,500,2)), 21)
Searching list of length  250
mid: 124 lower: -1 upper 250
mid: 61 lower: -1 upper 124
mid: 30 lower: -1 upper 61
mid: 14 lower: -1 upper 30
mid: 6 lower: -1 upper 14
mid: 10 lower: 6 upper 14
Number of comparisons: 6
10
>>> bsearch(list(range(1,500,2)), 256)
Searching list of length  250
mid: 124 lower: -1 upper 250
mid: 187 lower: 124 upper 250
mid: 155 lower: 124 upper 187
mid: 139 lower: 124 upper 155
mid: 131 lower: 124 upper 139
mid: 127 lower: 124 upper 131
mid: 129 lower: 127 upper 131
mid: 128 lower: 127 upper 129
Not found
Number of comparisons: 8
```

```
>>> bsearch(list(range(1,500000,2)), 256)
Searching list of length  250000
mid: 124999 lower: -1 upper 250000
…
mid: 127    lower: 126 upper 128
Not found
Number of comparisons: 18
```

```
>>> bsearch(list(range(1,5000000,2)), 256)
Searching list of length  2500000
mid: 1249999 lower: -1 upper 2500000
…
mid: 128     lower: 127 upper 129
Not found
Number of comparisons: 21
```

Analyzing Efficiency

- For binary search, consider the worst-case scenario (target is not in list)
- How many times can we split the search area in half before the list becomes empty?
- For the previous examples:
  15 --> 7 --> 3 --> 1 --> 0 ... 4 times

Analyzing Binary Search

How many times can we split the search area in half before the list becomes empty? (worst-case scenario: target is not in list)

15 → 7 → 3 → 1 → 0 ... 4 times

Example sequences of range sizes:

8 → 4 → 2 → 1	(4 key comparisons)
16 → 8 → 4 → 2 → 1	(5 key comparisons)
17 → 8 → 4 → 2 → 1	(5 key comparisons)
...	
31 → 15 → 7 → 3 → 1	(still 5 key comparisons)
32 → 16 → 8 → 4 → 2 → 1	(at last, 6 key comparisons)

Notice: $8 = 2^3$, $16 = 2^4$, $32 = 2^5$

Recall : $\log_a b = c$ is equivalent to $a^c = b$

Therefore: $\log_2 8 = 3$, $\log_2 16 = 4$, $\log_2 32 = 5$

Generalizing the Analysis

"floor"

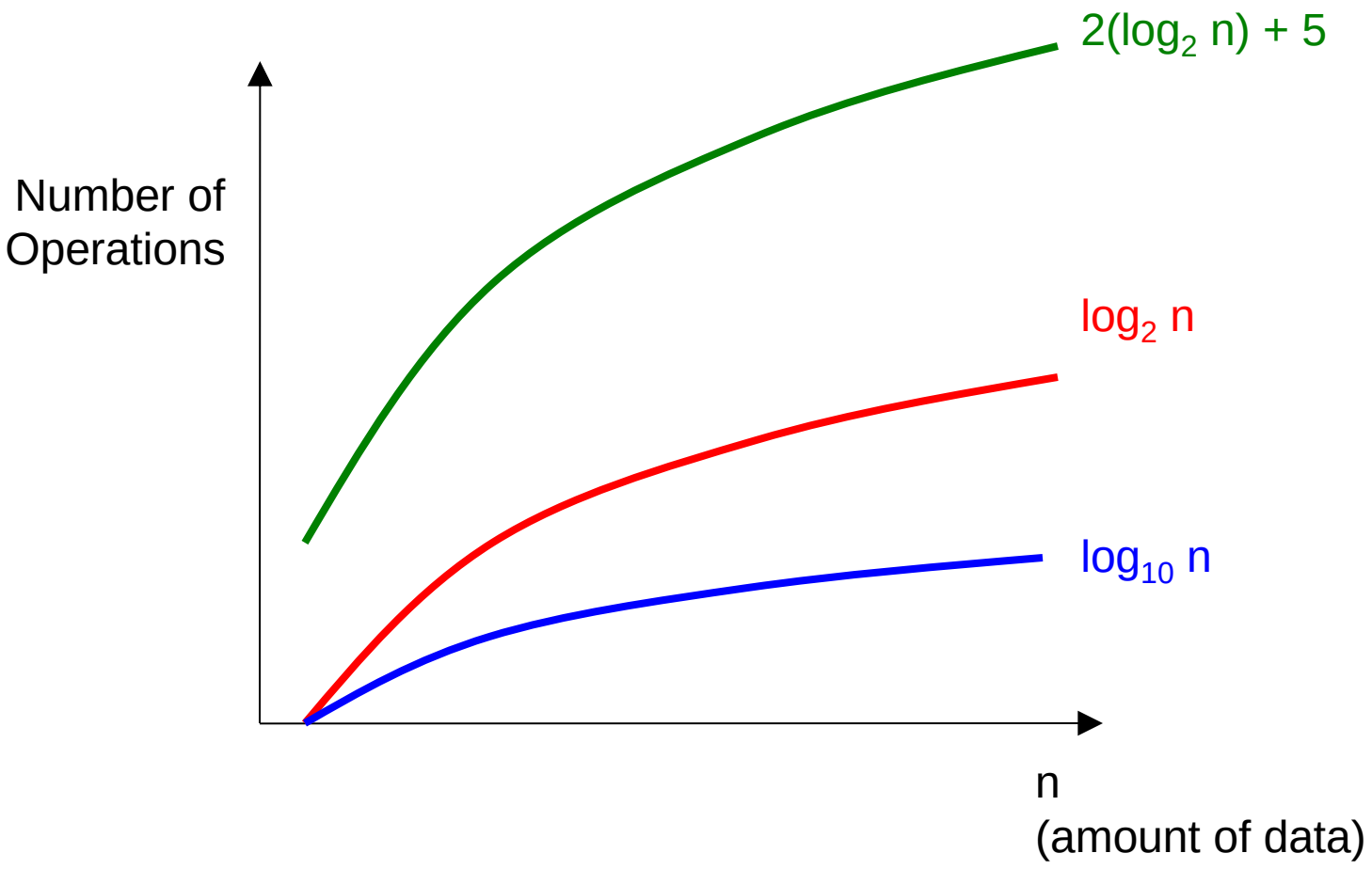
- *Some notation:* $\leq xf$ means round *x* down, so $\leq 2.5f = 2$ ($\lfloor 2.5 \rfloor = 2$)

- Binary search of *n* elements will do at most
  $1 + \leq \log_2 nf$ comparisons
  $1 + \leq \log_2 8f = 1 + \leq \log_2 9f = \ldots 1 + \leq \log_2 15f = 4$
  $1 + \leq \log_2 16f = 1 + \leq \log_2 17f = \ldots 1 + \leq \log_2 31f = 5$

- Why? We can split search region in half
  $1 + \leq \log_2 nf$ times before it becomes empty. ($\lfloor \log_2 n \rfloor + 1$)

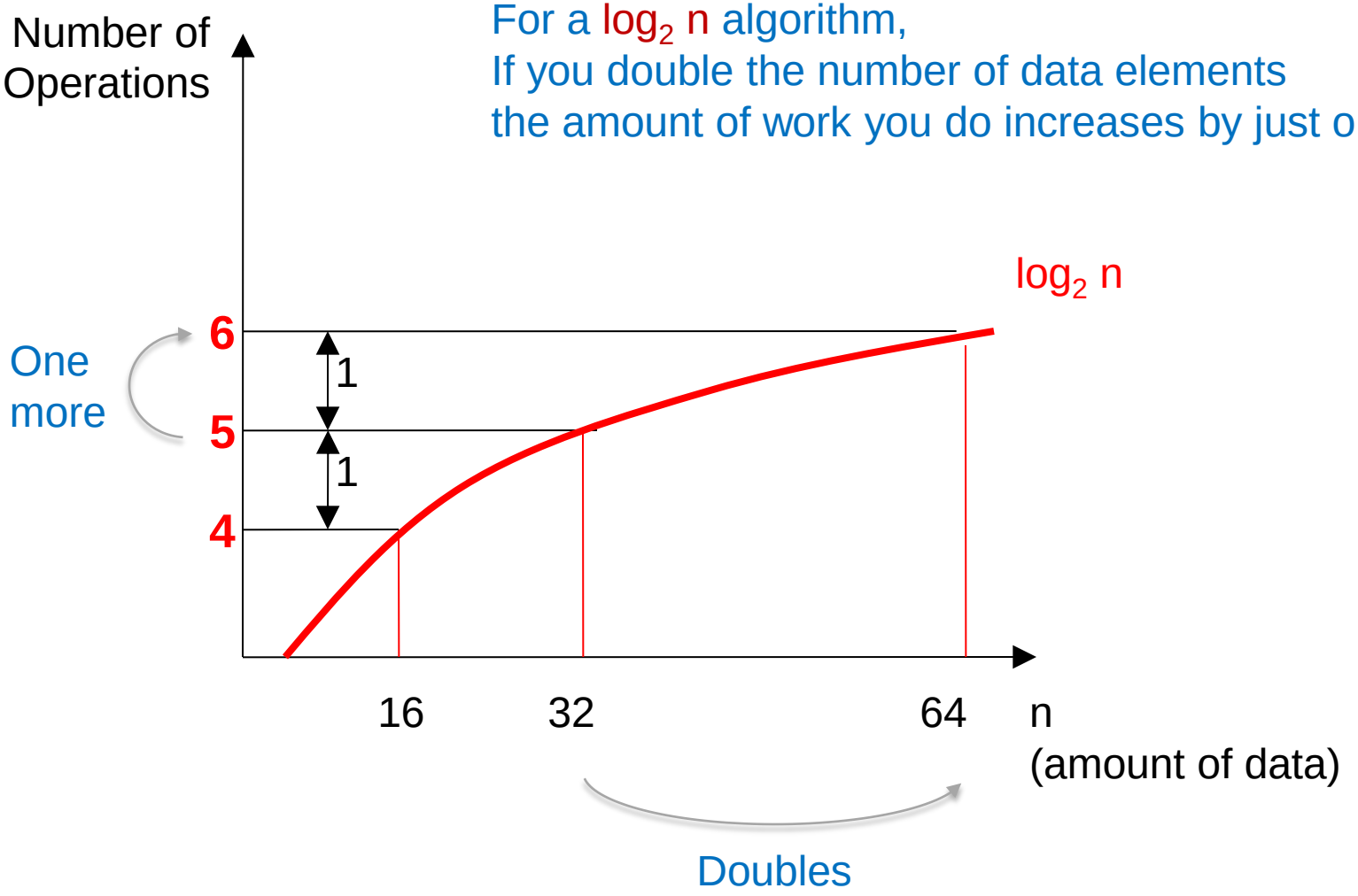
- "Big O" notation: we ignore the "1 +" and the floor function.
  We say Binary Search has complexity O(log *n*).

O(log n) (“logarithmic time”)



O(log n)

For a $\log_2 n$ algorithm,
If you double the number of data elements
the amount of work you do increases by just one unit



Binary Search (Worst Case)

Number of elements	Number of Comparisons
15	4
31	5
63	6
127	7
255	8
511	9
1023	10
1 million	20

Searching (Worst Case)

Number of elements	Number of Comparisons	
	Linear Search	Binary Search
$15 \approx 2^4$	15	4
$31 \approx 2^5$	31	5
$63 \approx 2^6$	63	6
$127 \approx 2^7$	127	7
$255 \approx 2^8$	255	8
$511 \approx 2^9$	511	9
$1023 \approx 2^{10}$	1023	10
1 million$\approx 2^{20}$	1000000	20
1 billion $\approx 2^{30}$	1000000000	30

Binary Search Pays Off

- Finding an element in an list with a million elements requires only 20 comparisons!

- BUT….
  - The list **must be sorted.**

  - **What if we sort the list first** using insertion sort?
    - Insertion sort $O(n^2)$ (worst case)
    - Binary search $O(\log n)$ (worst case)
    - **Total time:** **$O(n^2) + O(\log n)$** **$= O(n^2)$**

    Luckily there are faster ways to sort in the worst case…

Next Time

Merge Sort: a faster sorting algorithm

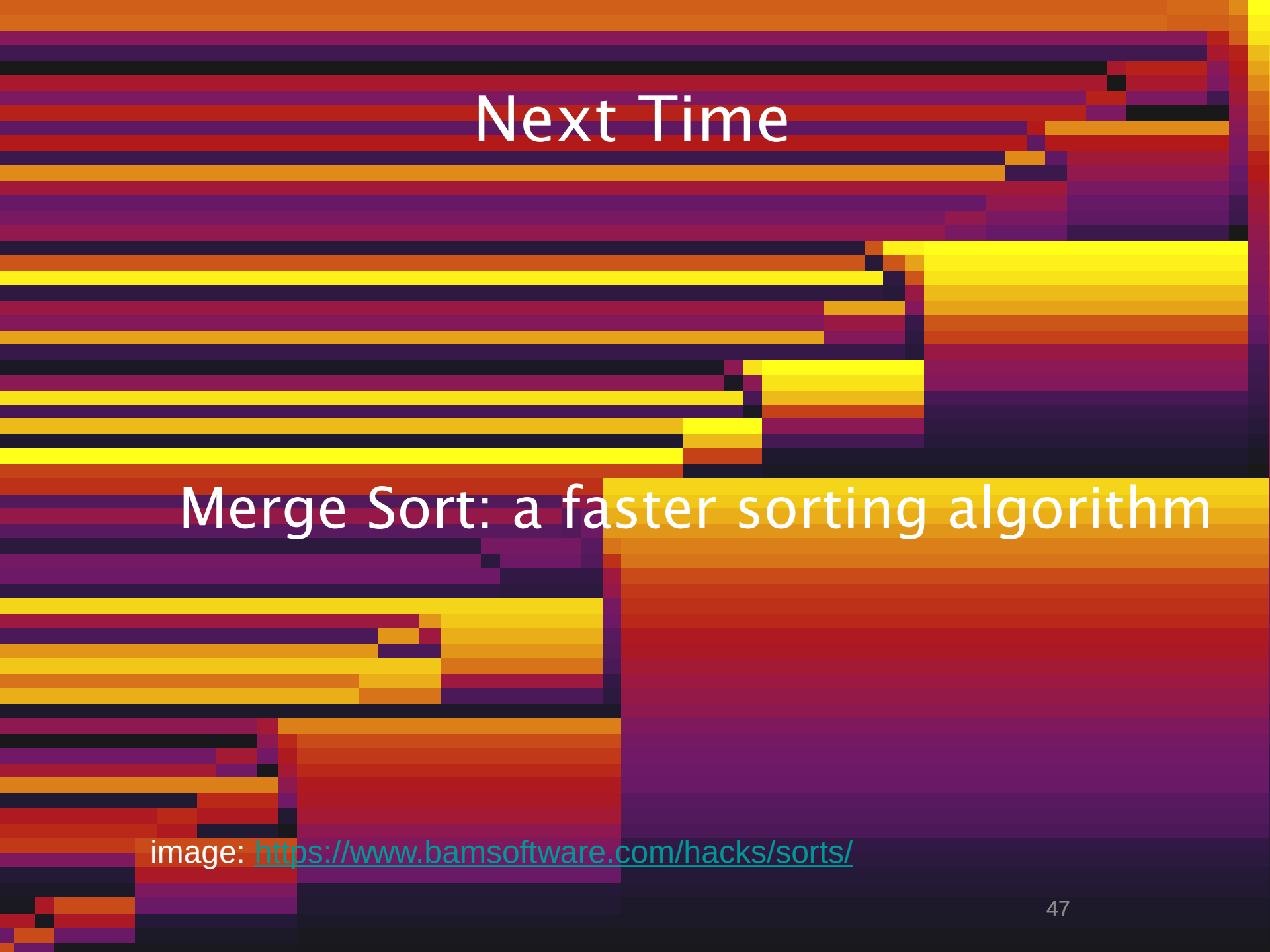


image: https://www.bamsoftware.com/hacks/sorts/