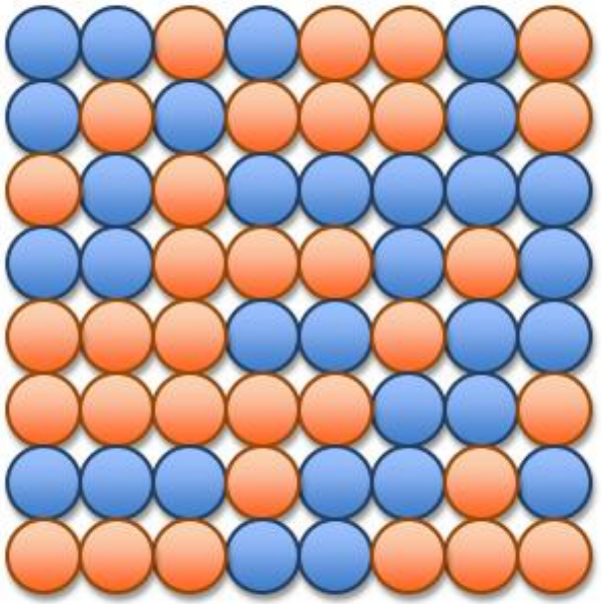


UNIT 5B
Binary Search

Binary Search

- Required: Array A of n unique elements.
  - The elements must be sorted in increasing order.
- Result: The index of a specific element (called the key) or nil if the key is not found.
- Algorithm uses two variables *lower* and *upper* to indicate the range in the array where the search is being performed.
  - *lower* is always one less than the **start** of the range
  - *upper* is always one more than the **end** of the range

Algorithm

1. Set lower = -1.
2. Set upper = the length of the array
3. Return BinarySearch(list, key, lower, upper).

BinSearch(list,key,lower,upper):

1. Return nil if the range is empty.
2. Set mid = the midpoint between lower and upper
3. Return mid if a[mid] is the key you're looking for.
4. If the key is less than a[mid],
   return BinarySearch(list,key,lower,mid)
   Otherwise, return BinarySearch(list,key,mid,upper).

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 nil

Finding `mid`

- How do you find the midpoint of the range?

```
mid = (lower + upper) / 2
```

  Example: **`lower = -1, upper = 9`**

  (range has 9 elements)

  `mid = 4`

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

Range is empty

- How do we determine if the range is empty?

```
lower + 1 == upper
```

Binary Search in Ruby: Recursively

```
def bsearch(list, key)
  return bs_helper(list, key, -1, list.length)
end
def bs_helper(list, key, lower, upper)
      return nil if lower + 1 == upper
      mid = (lower + upper)/2
      return mid if key == list[mid]
      if key < list[mid] then
            return bs_helper(list, key, lower, mid)
      else
            return bs_helper(list, key, mid, upper)
      end
end
```

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

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

```
                  key lower upper
bs_helper(list, 73,  -1,   15)
                                  mid = 7 and 42 < a[7]
bs_helper(list, 73,  -1,    7)
                                  mid = 3 and 42 > a[3]
bs_helper(list, 73,   3,    7)
                                  mid = 5 and 42 < a[5]
bs_helper(list, 73,   3,    5)
                                  mid = 4 and 42 > a[4]
bs_helper(list, 73,   4,    5)
                                  lower+1 == upper
                                  --->    Return nil.
```

Analyzing Efficiency

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

In general...

- In general, we can split search region in half $\lfloor \log_2 n \rfloor + 1$ times before it becomes empty.
- Recall the log function:

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

  Examples:

  $\log_2 128 = 7$

  $\log_2 n = 5$ implies $n = 32$
- In our example: when there were 15 elements, we needed 4 comparisons: $\lfloor \log_2 15 \rfloor + 1 = 3 + 1 = 4$

Big O

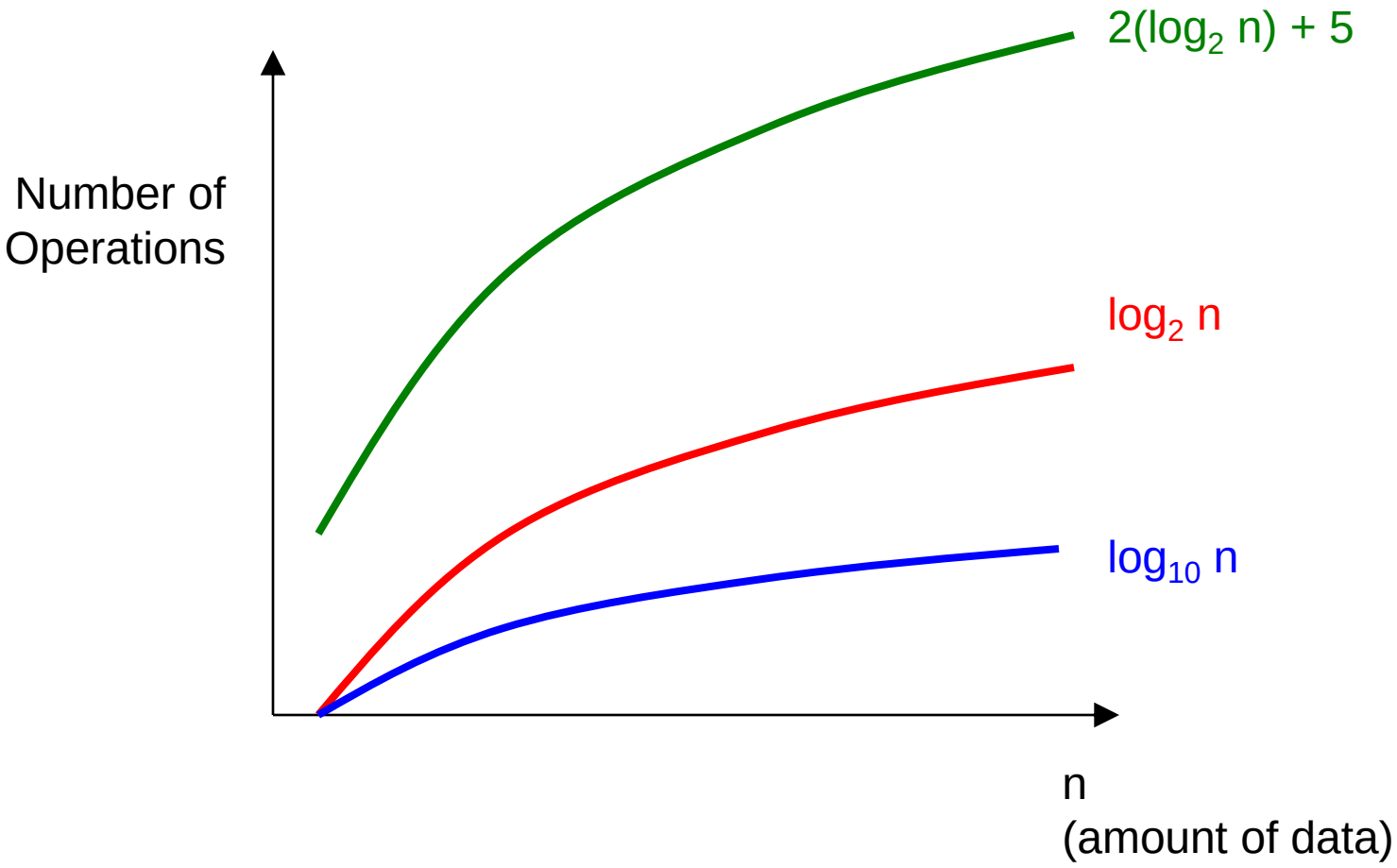
- In the worst case, binary search requires O(log n) time on a sorted array with n elements.
  - Note that in Big O notation, we do not usually specify the base of the logarithm. (It's usually 2.)
- <u>Number of operations</u> <u>Order of Complexity</u>

  $\log_2 n$ O(log n)

  $\log_{10} n$ O(log n)

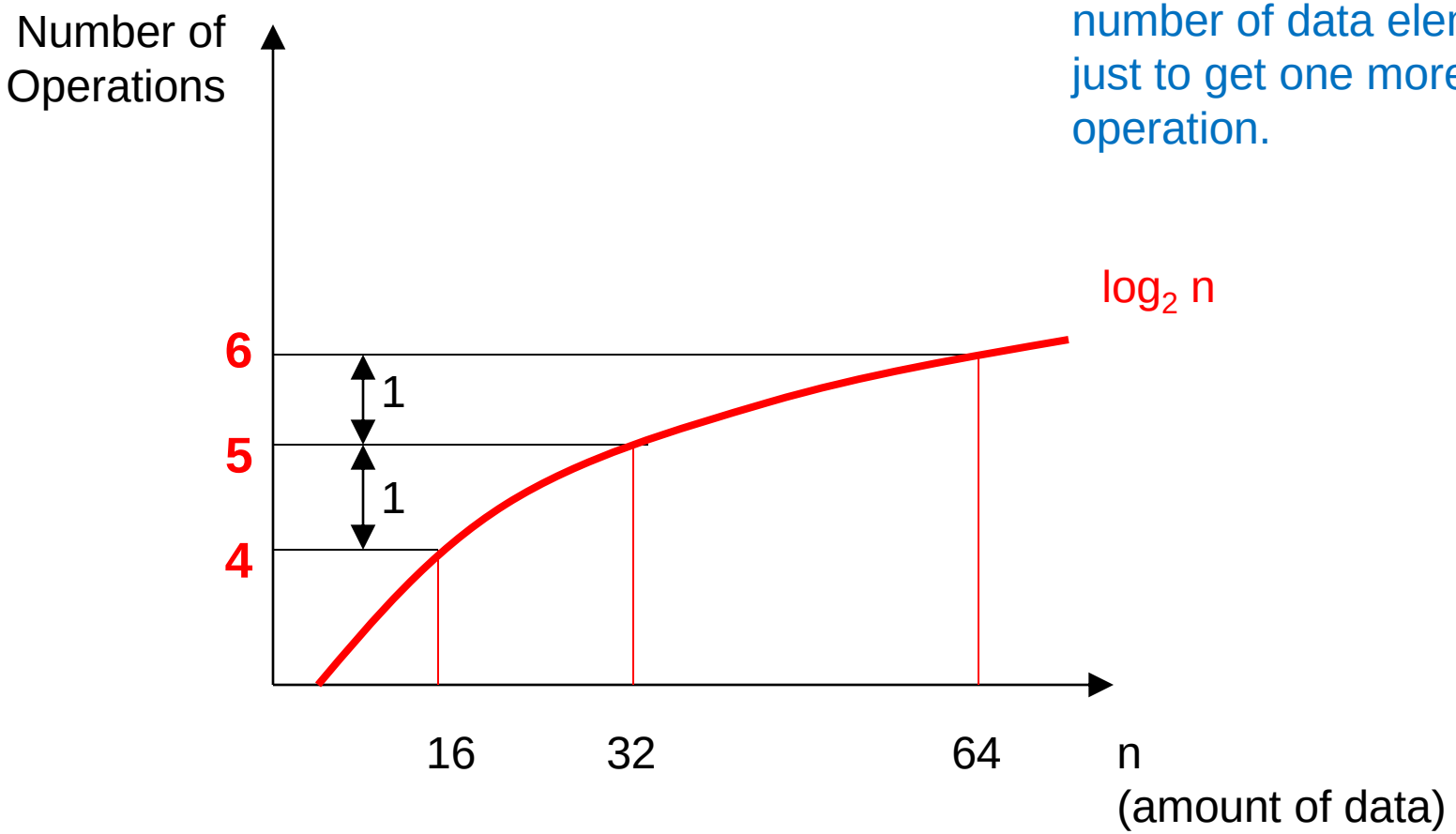
  $2(\log_2 n) + 5$ O(log n)

O(log n) ("logarithmic")



O(log n)

For a $\log_2$ algorithm, you have to double the number of data elements just to get one more operation.



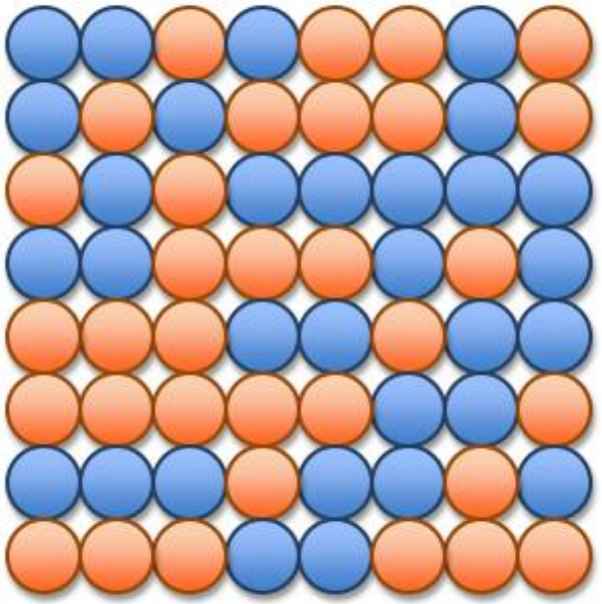
Binary Search (Worst Case)

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

Binary Search Pays Off

- Finding an element in an array with a million elements requires only 20 comparisons!
- BUT....
  - The array must be sorted.
  - What if we sort the array 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...



UNIT 5C
Merge Sort

Divide and Conquer

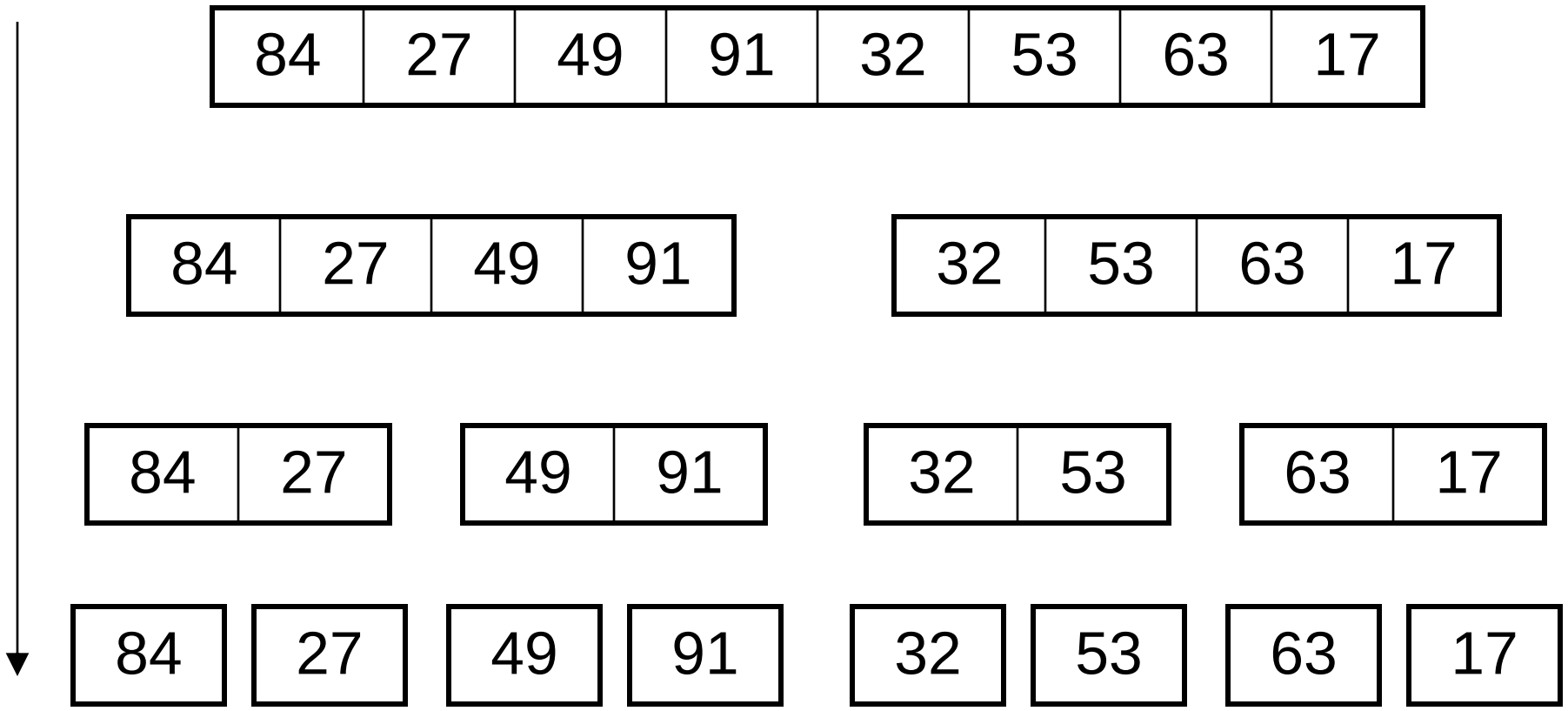
- In the military: strategy to gain or maintain power
- In computation:
  - Divide the problem into "simpler" versions of itself.
  - Conquer each problem using the same process (usually recursively).
  - Combine the results of the "simpler" versions to form your final solution.
- Examples: Towers of Hanoi, fractals, Binary Search, Merge Sort

Merge Sort

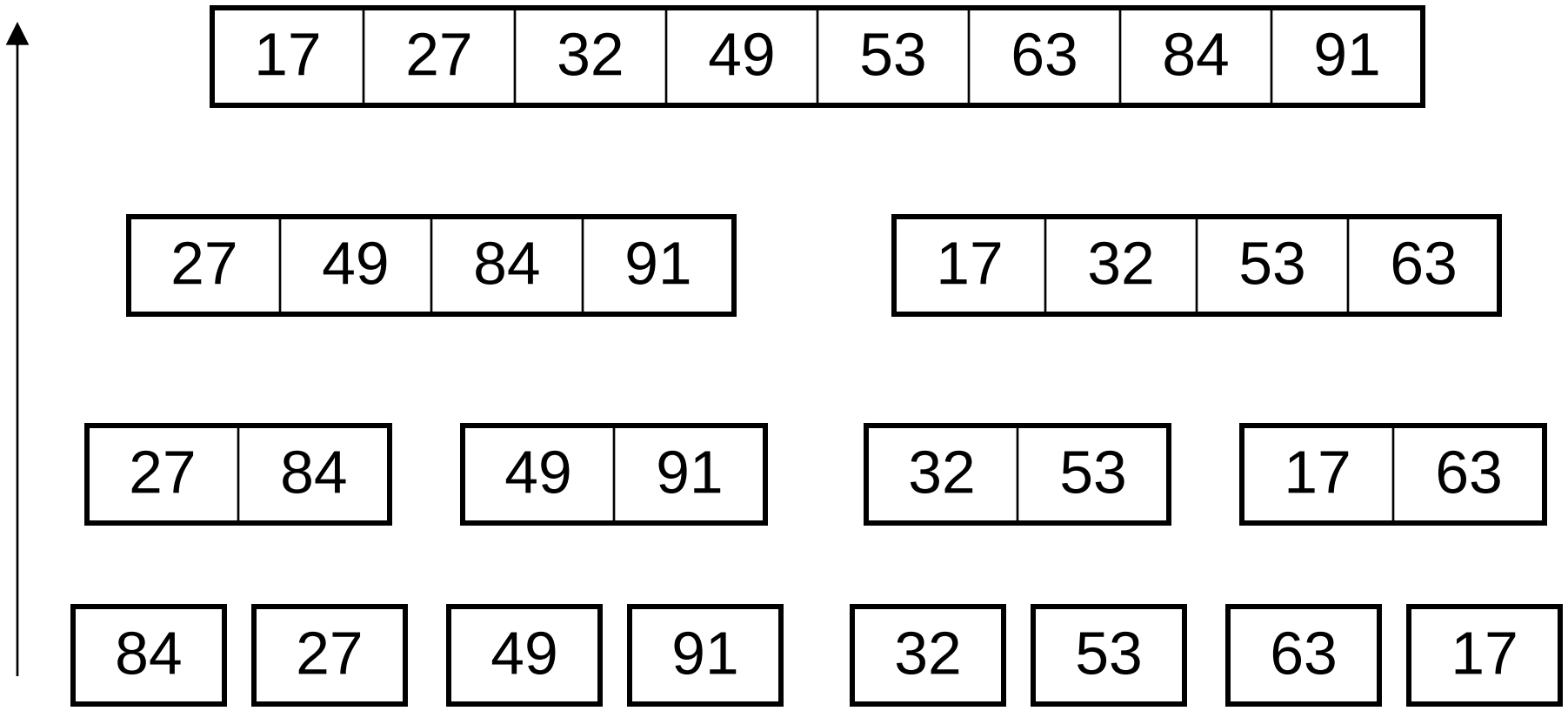
- Required: Array A of n elements.
- Result: Returns a new array containing the same elements in non-decreasing order.
- General algorithm for merge sort:

  1. Sort the first half using merge sort. (recursive!)
  2. Sort the second half using merge sort. (recursive!)
  3. Merge the two sorted halves to obtain the final sorted array.

Divide (Split)



Conquer (Merge)



Example 1: Merge

array a				array b				array c							
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	44	58	62	**29**	31	74	80	12							

0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	**44**	58	62	**29**	31	74	80	12	29						

0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	**44**	58	62	29	**31**	74	80	12	29	31					

0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	**44**	58	62	29	31	**74**	80	12	29	31	44				

Example 1: Merge (cont'd)

array a				array b				array c							
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	44	**58**	62	29	31	**74**	80	12	29	31	44	58			

0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	44	58	**62**	29	31	**74**	80	12	29	31	44	58	62		

0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	44	58	62	29	31	**74**	80	12	29	31	44	58	62	74	80

Example 2: Merge

array a			
0	1	2	3
58	67	74	90

array b			
0	1	2	3
19	26	31	44

array c							
0	1	2	3	4	5	6	7
19							

0	1	2	3
58	67	74	90

0	1	2	3
19	**26**	31	44

0	1	2	3	4	5	6	7
19	26						

0	1	2	3
58	67	74	90

0	1	2	3
19	26	**31**	44

0	1	2	3	4	5	6	7
19	26	31					

0	1	2	3
58	67	74	90

0	1	2	3
19	26	31	**44**

0	1	2	3	4	5	6	7
19	26	31	44				

0	1	2	3
58	67	74	90

0	1	2	3
19	26	31	44

0	1	2	3	4	5	6	7
19	26	31	44	58	67	74	90

Merge

- Required: Two arrays a and b.
  - Each array must be sorted already in non-decreasing order.
- Result: Returns a new array containing the same elements merged together into a new array in non-decreasing order.
- We'll need two variables to keep track of where we are in arrays a and b: index_a and index_b.

1. Set index_a equal to 0.
2. Set index_b equal to 0.
3. Create an empty array c.

Merge (cont'd)

4. While index_a < the length of array a and index_b < the length of array b, do the following:

   a. If a[index_a] ≤ b[index_b], then do the following:

      i. append a[index_a] on to the end of array c

      ii. add 1 to index_a

   Otherwise, do the following:

      i. append b[index_b] on to the end of array c

      ii. add 1 to index_b

Merge (cont'd)

(Once we finish step 4, we've added all of the elements of either array a or array b to array c. The other array still has some elements left that need to be added to array c.)

5. If index_a < the length of array a, then:

   append all remaining elements of array a on to the end of array c

   Otherwise:

   append all remaining elements of array b on to the end of array c

6. Return array c as the result.

Merge in Ruby

```
def merge(a, b)
  index_a = 0
  index_b = 0
  c = []
  while index_a < a.length and index_b < b.length do
      if a[index_a] <= b[index_b] then
            c << a[index_a]
            index_a = index_a + 1
      else
            c << b[index_b]
            index_b = index_b + 1
      end
  end
```

Merge in Ruby (cont'd)

```
  if (index_a < a.length) then
      for i in (index_a..a.length-1) do
            c << a[i]
      end
  else
      for i in (index_b..b.length-1) do
            c << b[i]
      end
  end
  return c
end
```

Merge Sort: Base Case

- General algorithm for merge sort:

  1. Sort the first half using merge sort. (recursive!)
  2. Sort the second half using merge sort. (recursive!)
  3. Merge the two sorted halves to obtain the final sorted array.

- What is the base case?
  If the list has only 1 element, it is already sorted so just return the list as the result.

Merge Sort: Halfway Point

- General algorithm for merge sort:

  1. Sort the first half using merge sort. (recursive!)
  2. Sort the second half using merge sort. (recursive!)
  3. Merge the two sorted halves to obtain the final sorted array.

- How do we determine the halfway point where we want to split the array *list*?

  First half: `0..list.length/2-1`

  Second half: `list.length/2..list.length-1`

Merge Sort in Ruby

```
def msort(list)
  return list if list.length == 1  # base case
  halfway = list.length/2
  list1 = list[0..halfway-1]
  list2 = list[halfway..list.length-1]
  newlist1 = msort(list1)    # recursive!
  newlist2 = msort(list2)    # recursive!
  newlist = merge(newlist1, newlist2)
  return newlist
end
```

Analyzing Efficiency

- If you merge two lists of size i/2 into one new list of size i, what is the maximum number of appends that you must do?
- Clearly, each element must be appended to the new list at some point, so the total number of appends is i.
- If you have a set of pairs of lists that need to be merged (two pairs at a time), and the total number of elements in all of the lists combined is n, the total number of appends will be n.

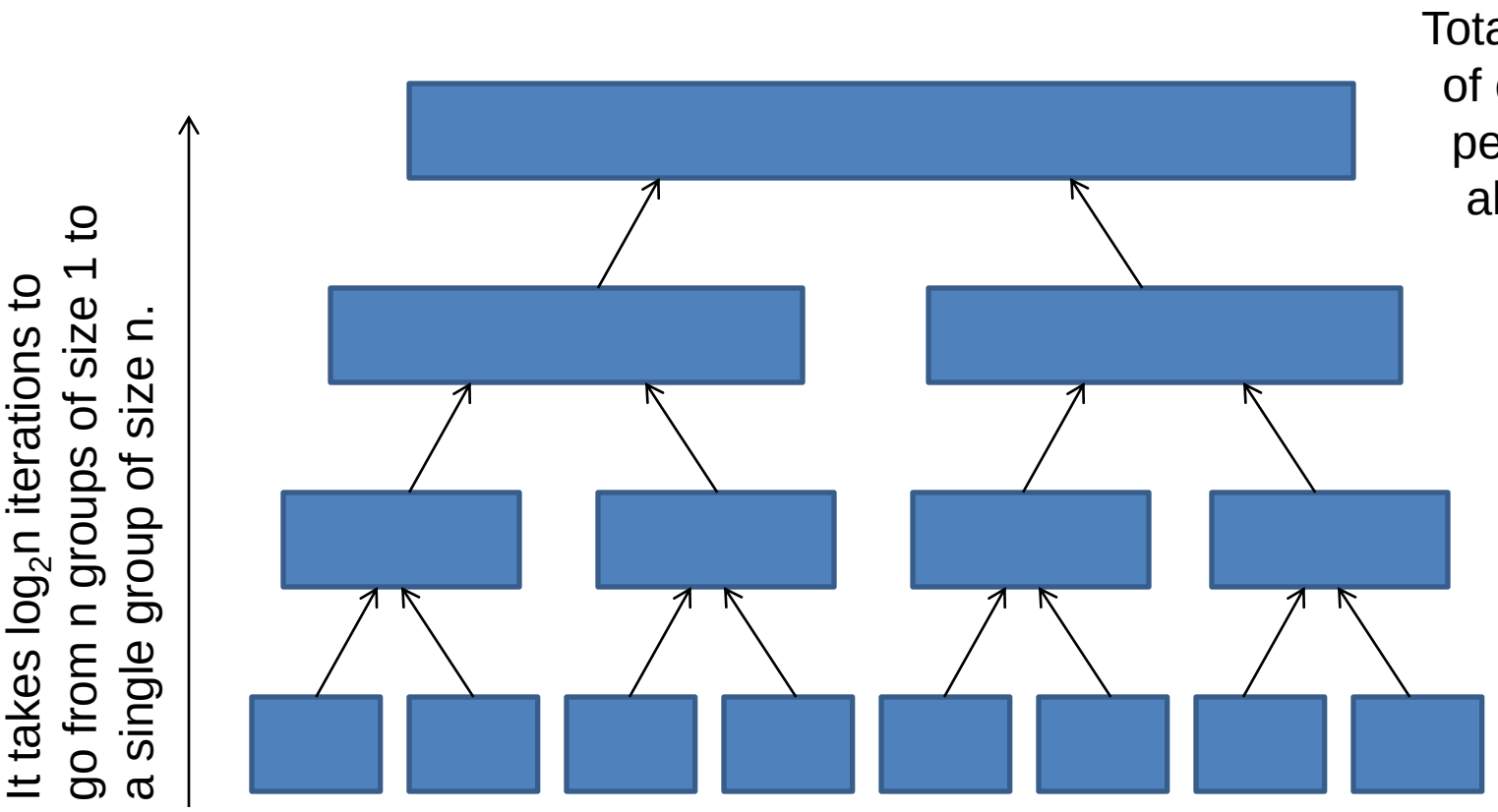
How many group merges?

- How many group merges does it take to go from n groups of size 1 to 1 group of size n?
- Example: Merge sort on 32 elements.
  - Break down to groups of size 1 (base case).
  - Merge 32 lists of size 1 into 16 lists of size 2.
  - Merge 16 lists of size 2 into 8 lists of size 4.
  - Merge 8 lists of size 4 into 4 lists of size 8.
  - Merge 4 lists of size 8 into 2 lists of size 16.
  - Merge 2 lists of size 16 into 1 list of size 32.

$5 = \log_2 32$

- In general: $\log_2 n$ group merges must occur.

Putting it all together



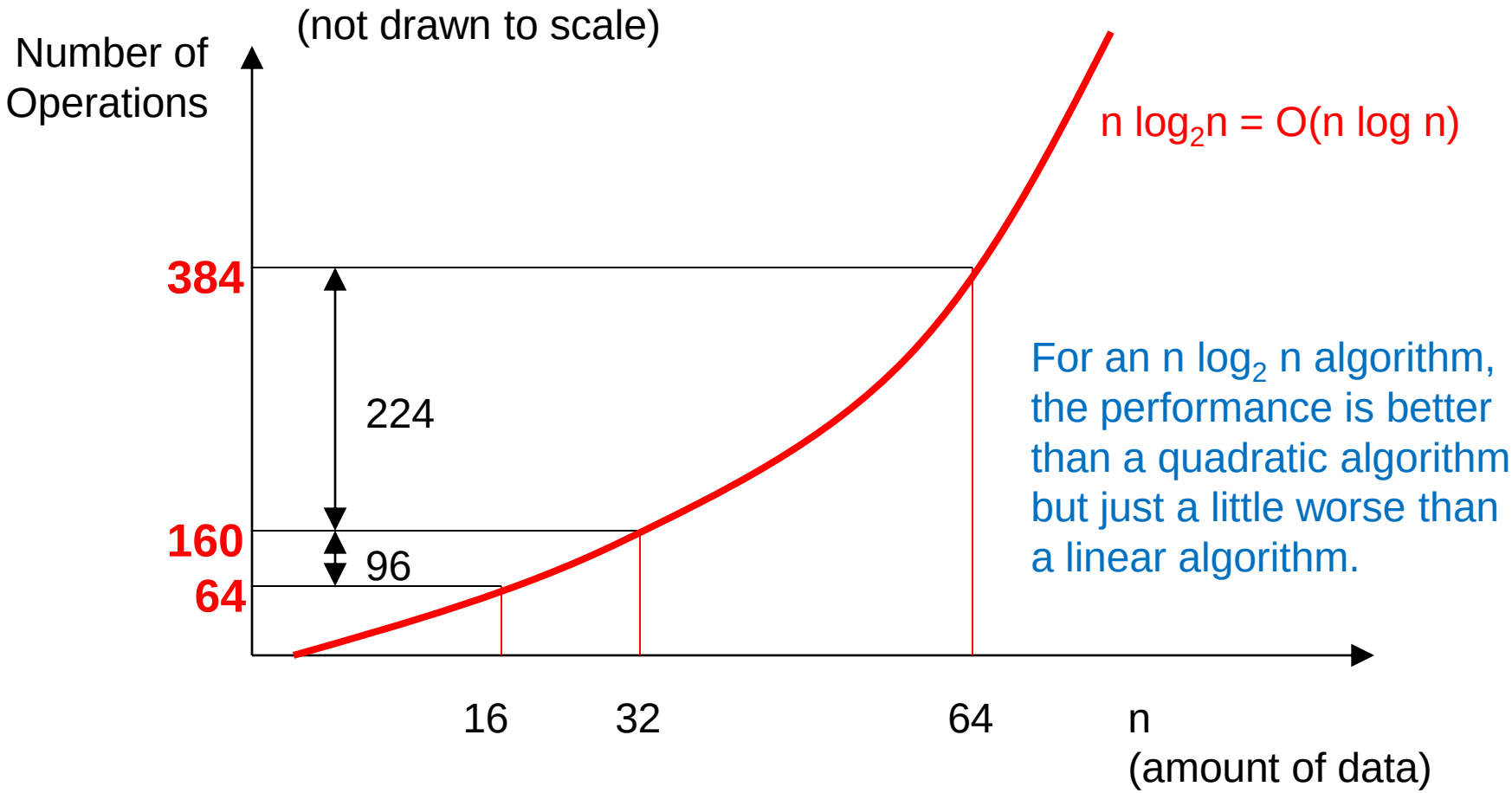
It takes n appends to merge all pairs to the next higher level.

Big O

- In the worst case, merge sort requires O(n log n) time to sort an array with n elements.

Number of operations	Order of Complexity
$n \log_2 n$	O(n log n)
$4n \log_{10} n$	O(n log n)
$n \log_2 n + 2n$	O(n log n)

O(N log N)


(not drawn to scale)
Number of Operations
n $\log_2$n = O(n log n)
384
224
160
96
64
16
32
64
n
(amount of data)
For an n $\log_2$ n algorithm, the performance is better than a quadratic algorithm but just a little worse than a linear algorithm.

Comparing Insertion Sort to Merge Sort

(Worst Case)

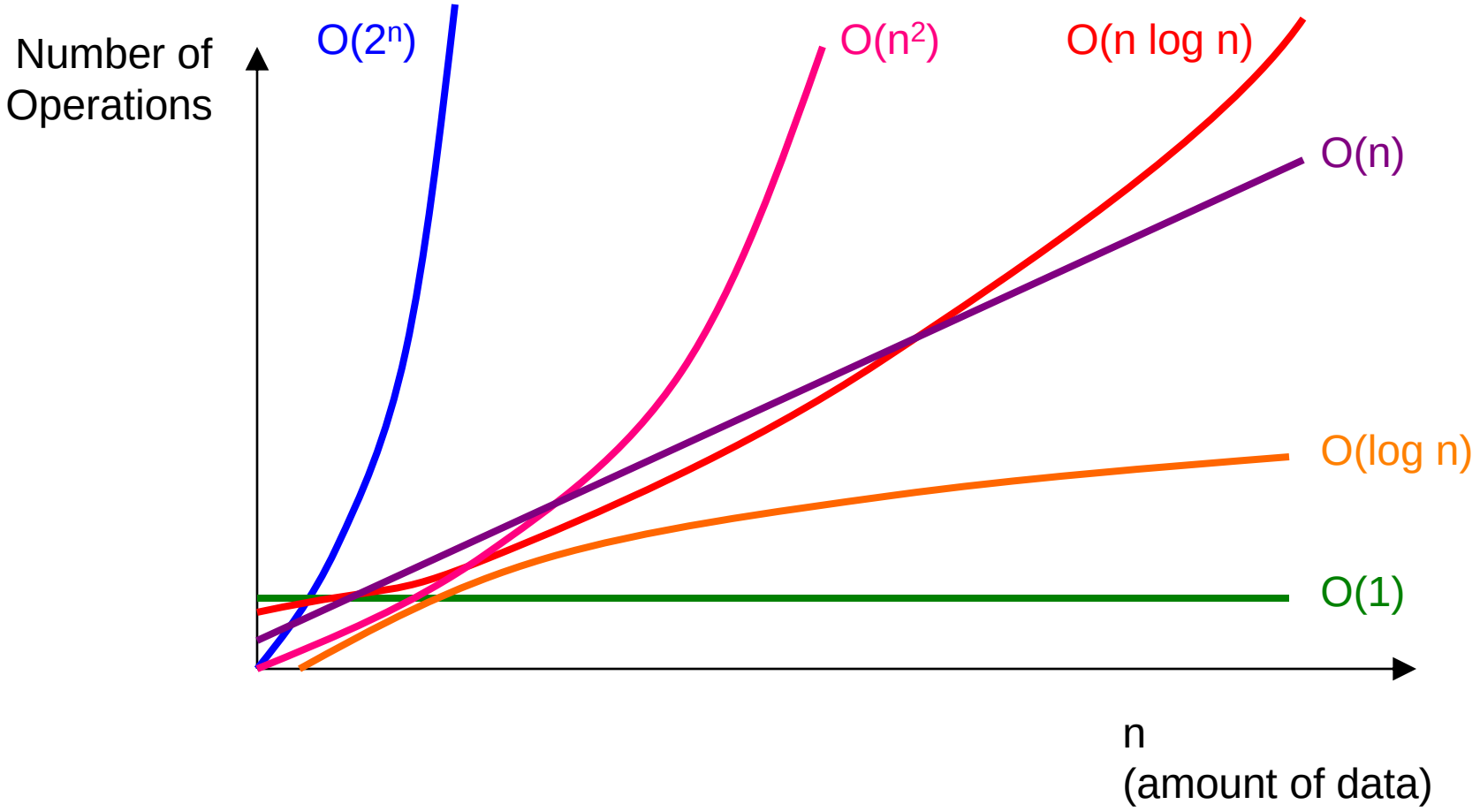
n	isort (n(n+1)/2)	msort ($n \log_2 n$)
8	36	24
16	136	64
32	528	160
2^{10}	524,800	10,240
2^{20}	549,756,338,176	20,971,520

For array sizes less than 100, there's not much difference between these sorts, but for larger arrays sizes, there is a clear advantage to merge sort.

Sorting and Searching

- Recall that if we wanted to use binary search, the array must be sorted.
  - What if we sort the array first using merge sort?
    - Merge sort O(n log n) (worst case)
    - Binary search O(log n) (worst case)
    - Total time: (worst case) O(n log n) + O(log n) = O(n log n)

Comparing Big O Functions



Merge Sort: Iteratively
(optional)

- *If you are interested, the textbook discusses an iterative version of merge sort which you can read on your own.*
- *This version uses an alternate version of the* `merge` *function that is not shown in the textbook but is given in the RubyLabs gem.*