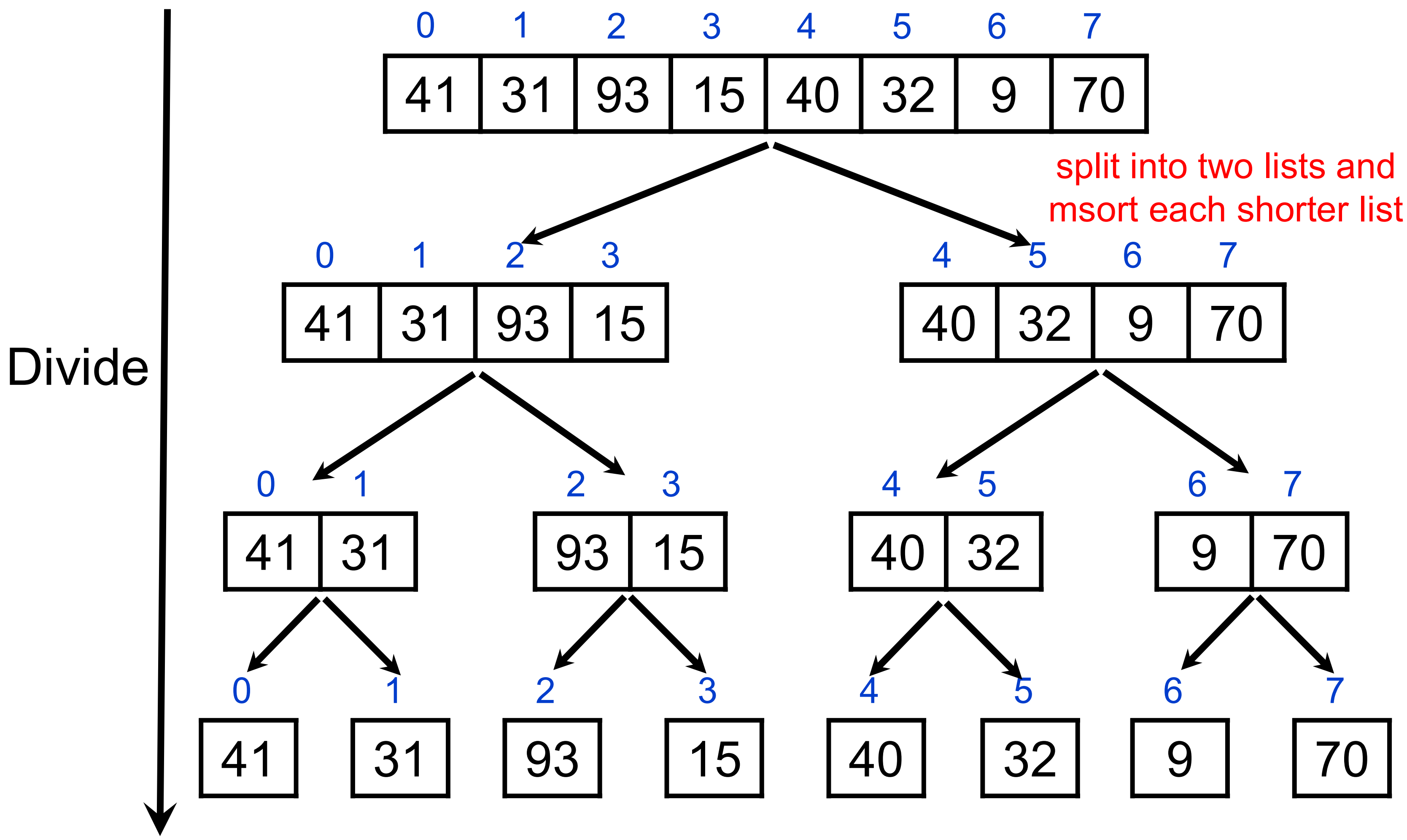
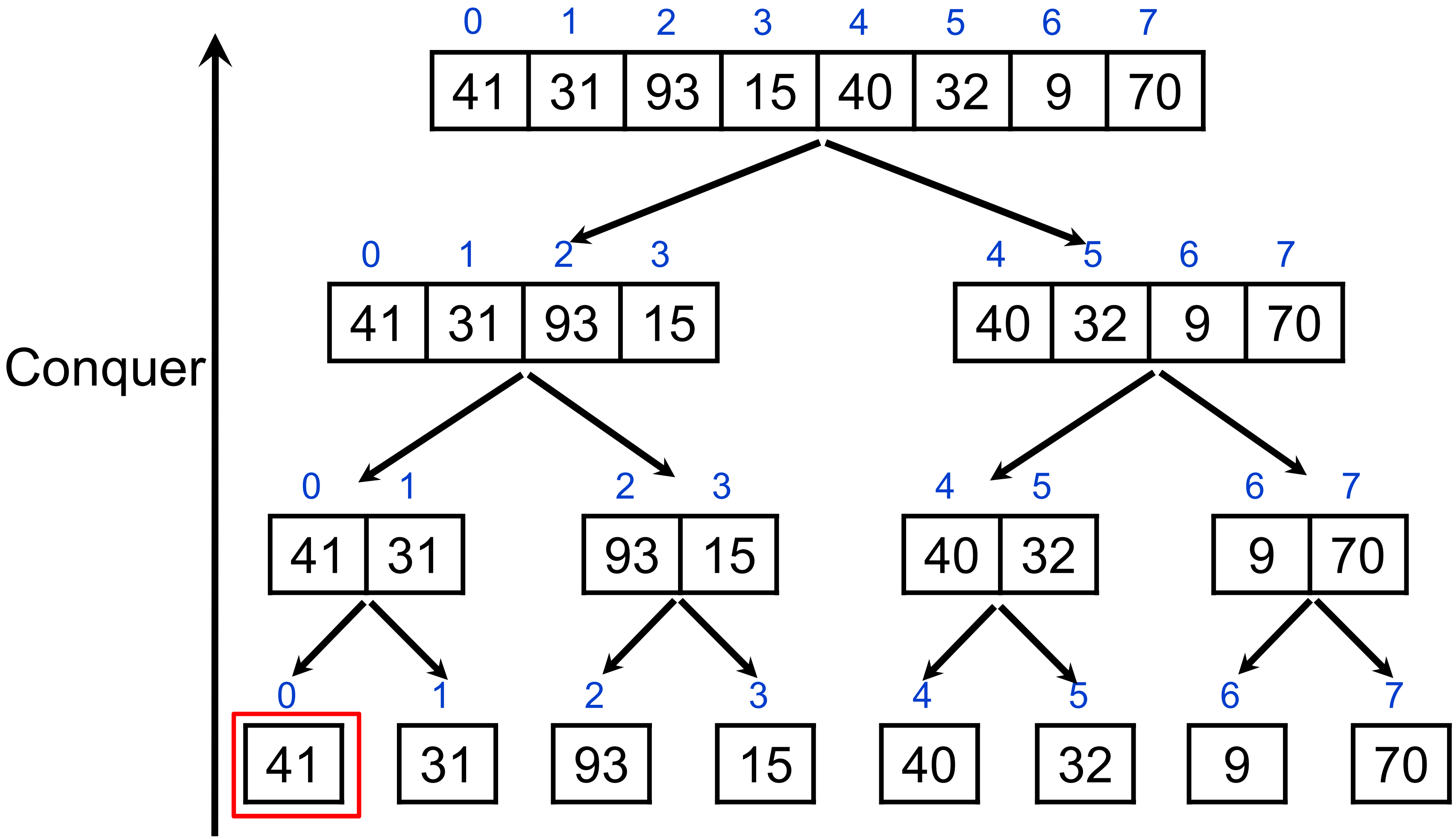
Merge Sort

- Last week we have seen the iterative version of merge sort
- The iterative definition is based on bottom-up strategy
  - Begin with individual elements
  - Iteratively merge groups of increasing sizes
- Merge sort can also be described recursively
- The recursive definition is based on top-down strategy
  - Begin with the full array to be sorted
  - Divide the array into shorter arrays to be sorted
  - Sort the shorter arrays recursively

Merge Sort: Divide

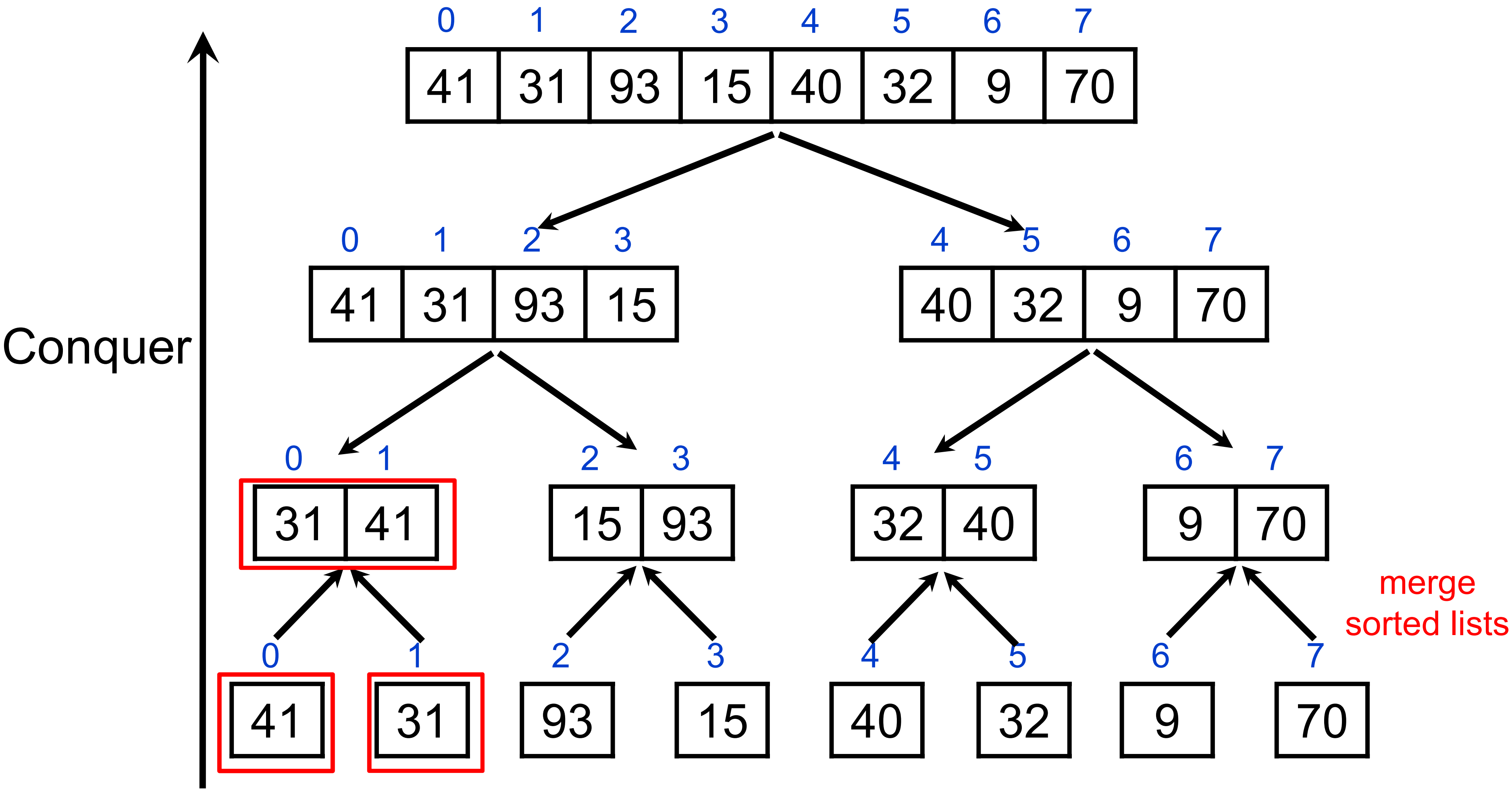


Merge Sort: Conquer

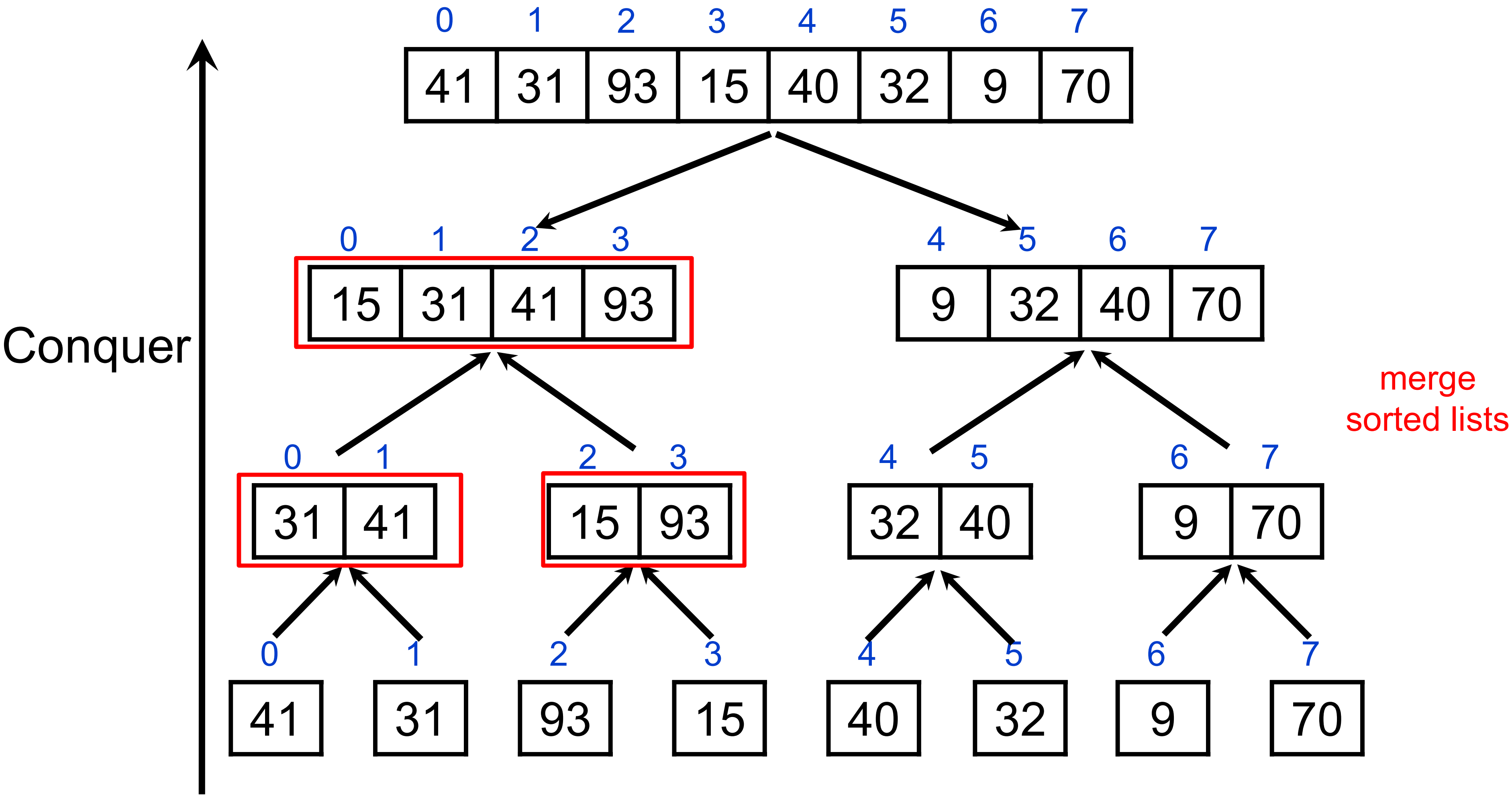


base case only 1 item,
consider it already sorted

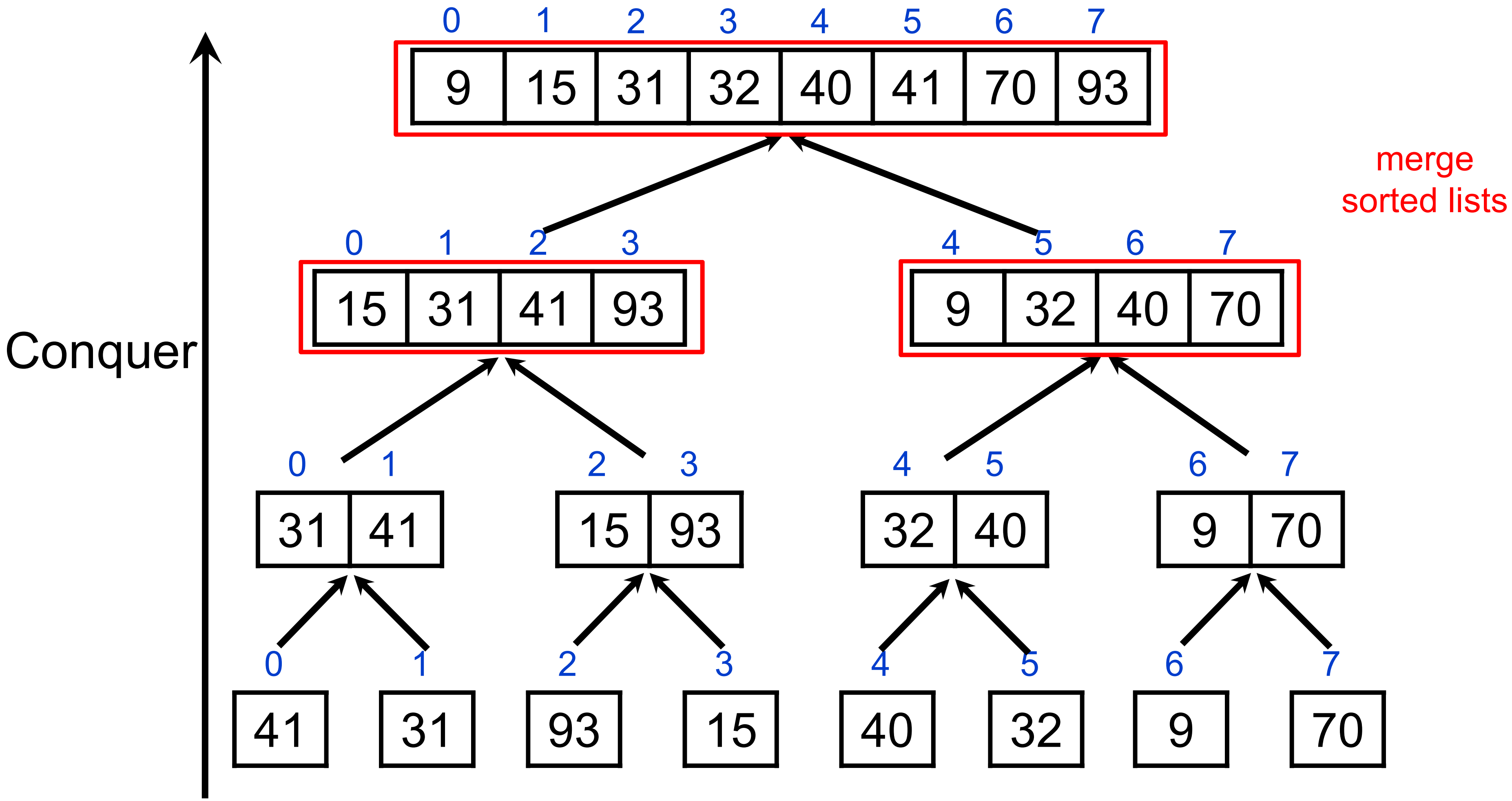
Merge Sort: Conquer



Merge Sort: Conquer



Merge Sort: Conquer



Recursive Version of Merge Sort

```python
def rmSort(a):
    if len(a) == 1:
        return a
    mid = len(a)//2
    a1 = rmSort(a[0:mid])
    a2 = rmSort(a[mid:len(a)])
    return merge2(a1, a2)
```

Recursive Version of Merge Sort

```python
def rmSort(a):
    if len(a) == 1:
        return a
    mid = len(a)//2
    a1 = rmSort(a[0:mid])
    a2 = rmSort(a[mid:len(a)])
    return merge2(a1, a2)
```

base case – single element

recursive call to smaller instances of the original problem

merge results from two sub-problems

Recursive Version of Merge Sort

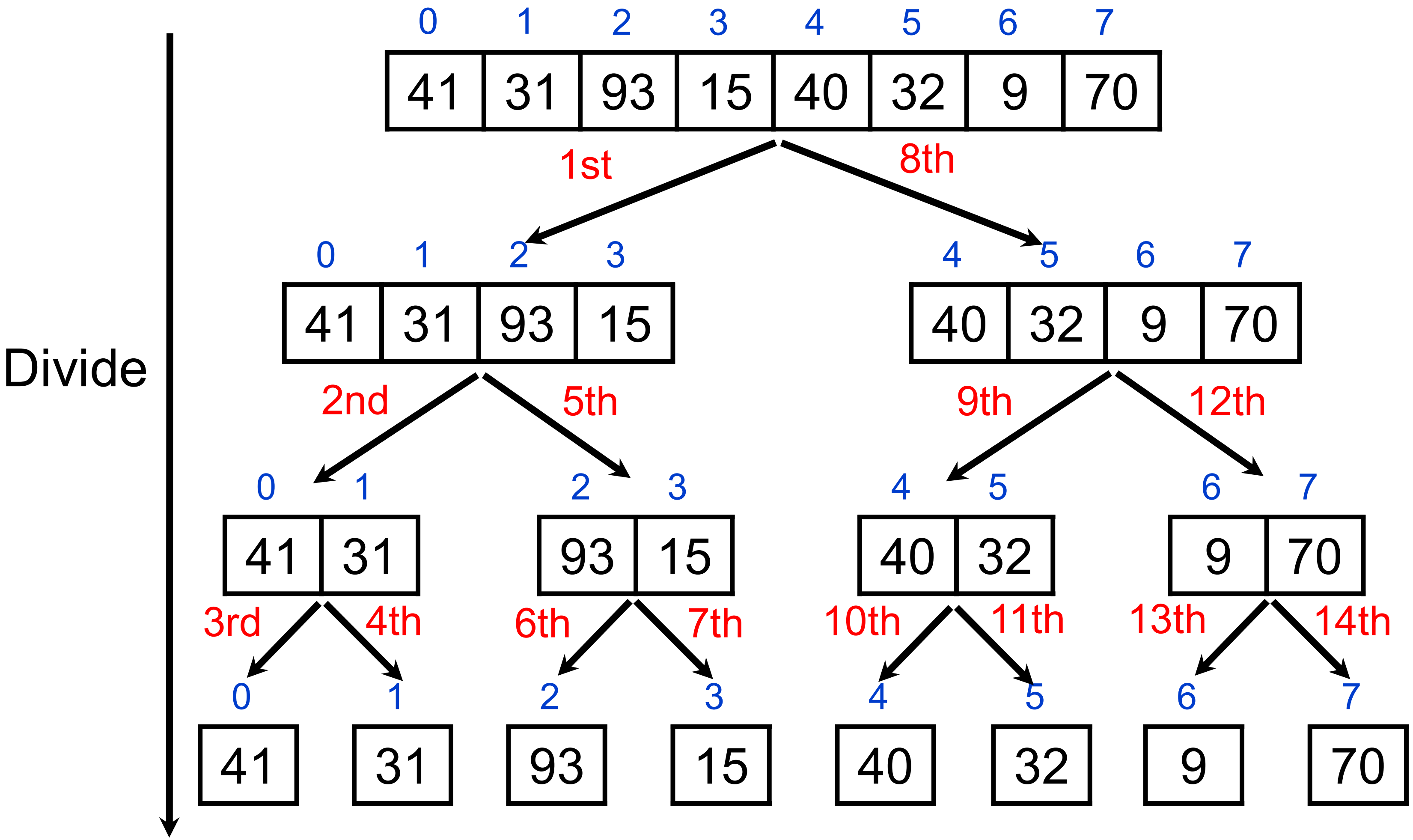
```
# takes in 2 sorted lists (a1 and a2)
# returns a1 + a2 sorted
def merge2(a1, a2):
  i = 0
  j = 0
  ret = []
  while i < len(a1) or j < len(a2):
    if (j == len(a2)) or (i < len(a1) and a1[i] < a2[j]):
      ret.append(a1[i]) # pick item from a1
      i += 1
    else:
      ret.append(a2[j]) # pick item from a2
      j += 1
  return ret
```

*Repeat as long as there is at least 1 element in **a1** OR there is at least 1 element in **a2***

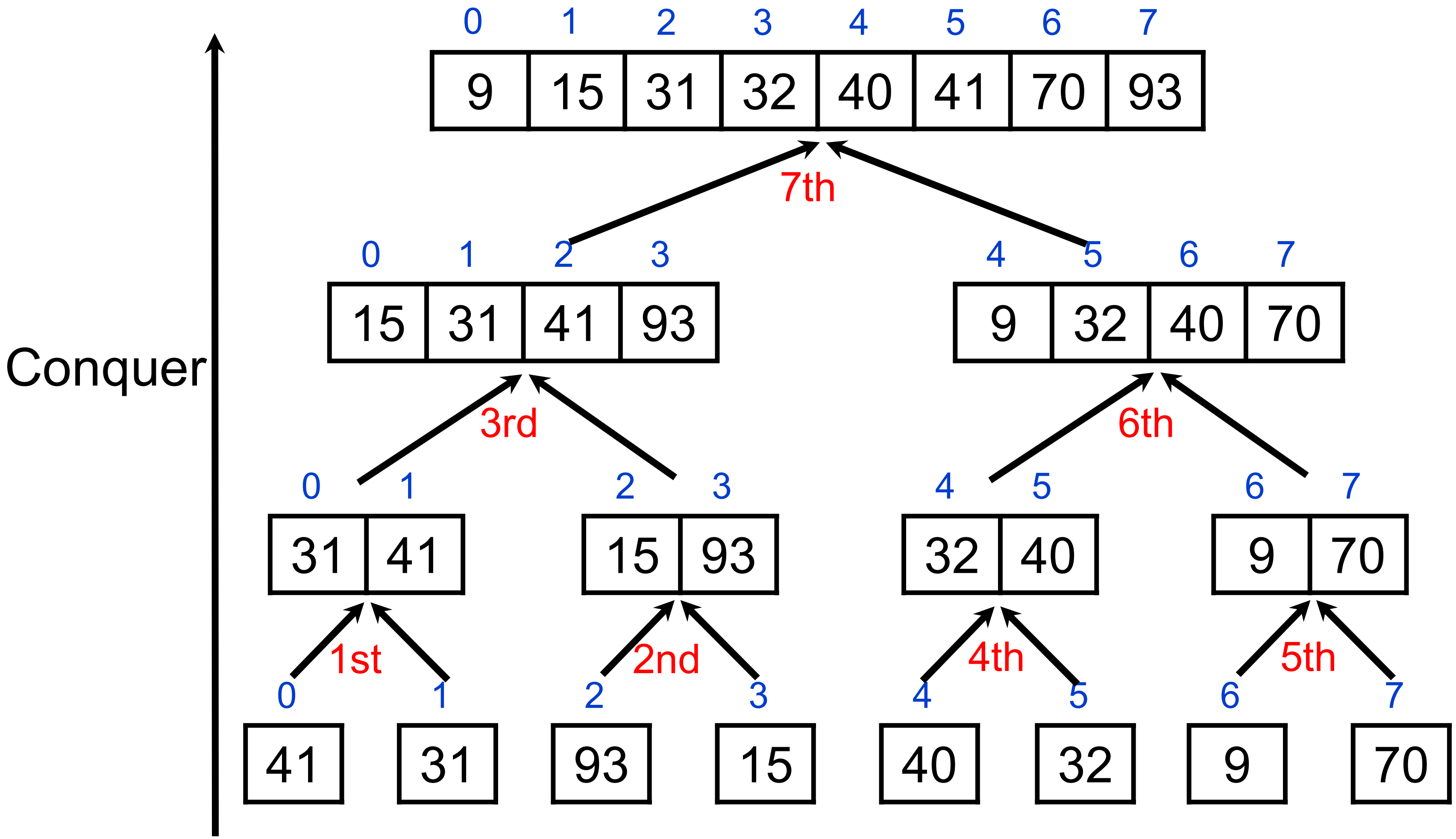
*There is at least 1 element in **a1** AND the current element in **a1** is smaller than the current element in **a2***

*No more elements in **a2***

Sequence of Calls to `rmsort`

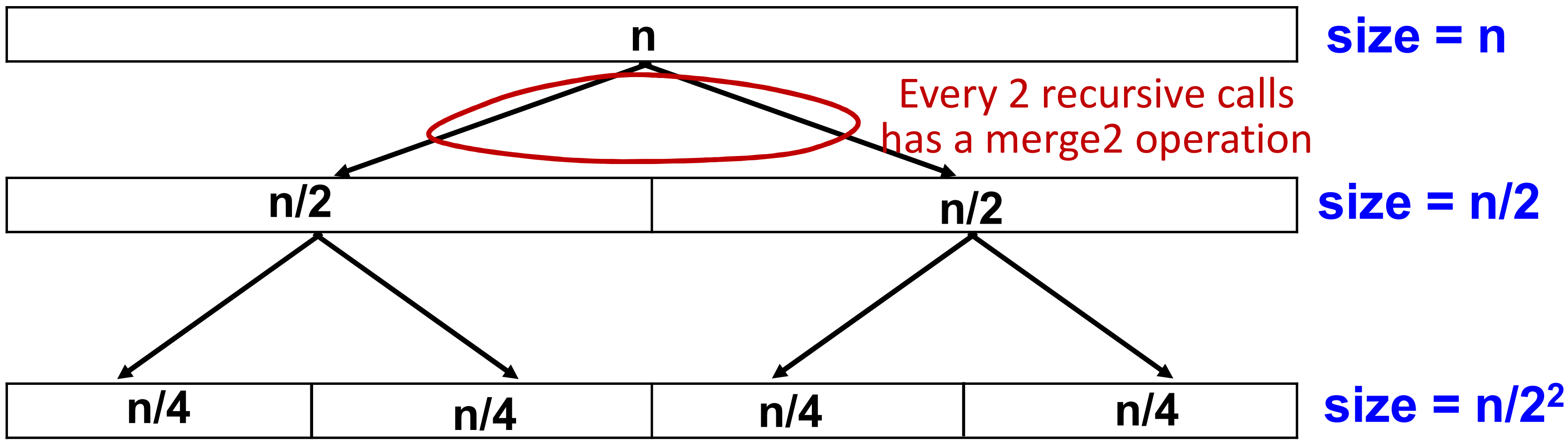


Sequence of Calls to `merge2`



Complexity of Recursive Merge Sort

- Number of comparisons is the same for both iterative and recursive versions
- In the recursive version *(refer next slide for illustration)*:
  - There are `log n` levels of recursion with decreasing group size `size` (in the above case, it's 8, 4, and 2).
  - For each level, there are `(2 x n/size)` recursive function calls.
  - For each level, every pair of recursive function calls has a `merge2` operation. That is, each level has `(n/size)` calls to the `merge2` function.
  - Each `merge2` operation conducts up to `size` comparisons in the worst case.
- Complexity of recursive merge sort is $O(n \log n)$.



$n/2^k = 1$

$\Rightarrow \quad n = 2^k$

$\Rightarrow \quad k = \log_2 n$

i.e. there are $\log_2 n$ levels of recursive calls

In-Class Exercise Q3

For the following input arrays, determine the state of the array just before the final merge step for the two versions of merge sort:

- bottom-up non-recursive version
- top-down recursive version

(a) [26, 34, 64, 83, 2, 5]

(b) [52, 30, 98, 59, 67, 4, 64, 12, 79]

Another Example: Binary Search

```
# iterative version of bsearch
def bsearch(array, target):
    lower = -1
    upper = len(array)

    while not (lower + 1 == upper):
        mid = (lower + upper)//2
        if target == array[mid]:  #success
            return mid
        elif target < array[mid]:
            upper = mid             #search lower region
        else:
            lower = mid             #search upper region
    return -1                       #not found
```

…Another Example: Binary Search

```
# recursive version of bsearch
def rbsearch(array, target, lower=None, upper=None):
    if lower == None:
        lower = -1
        upper = len(array)

    if lower + 1 == upper:     # base case
        return -1              # not found
    mid = (lower + upper)//2
    if array[mid] == target:   # success
        return mid
    elif array[mid] < target: # search upper region
        return rbsearch(array, target, mid, upper)
    else:                      # search lower region
        return rbsearch(array, target, lower, mid)
```

only happens when rbsearch is called the 1st time

Summary

- An algorithm that uses divide and conquer can be written using ***iteration*** or ***recursion***
  - recursive = “self-similar”
  - a problem that can be divided into smaller subproblems
  - a recursive function calls itself
- Fundamentals of recursion:
  - reduction step
  - base case
- Recursive versions of:
  - merge sort

Further reading materials

- Online sources:
  - http://en.wikipedia.org/wiki/Recursion_(computer_science)
  - http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/01-recursion.pdf
  - http://www-cs-faculty.stanford.edu/~eroberts/courses/cs106b/chapters/05-intro-to-recursion.pdf
  - http://introcs.cs.princeton.edu/java/23recursion/

- Optional supplementary text (available in Library):
  - Prichard and Carrano, Data Abstraction and Problem Solving with Java
    - Chapter 3 "Recursion: The Mirrors"

Optional Reading:
What is “Tail Recursion”?

- https://stackoverflow.com/questions/33923/what-is-tail-recursion

Road Map

Algorithm Design and Analysis

(Weeks 1 - 5)

Fundamental Data Structures

Next week →

- Week 6: Linear data structures (stack, queue)
- Week 7: Hierarchical data structure (binary tree)
- Week 9: Networked data structure (graph)
- Week 10: Graph Algorithms

Computational Intractability and Heuristic Reasoning

(Weeks 11 - 13)