Algorithms

Lec#2

Fall 2015

Review of Last Lecture

- Algorithm: is a step-by-step procedure for solving a problem in a finite amount of time.
- Running time : of an algorithm on a particular input is the number of primitive operation or steps executed.
- Experimental studies:
  - write a program implementing the algorithm.
  - Run the program with inputs of varying size .
  - Plot the results.

Review of Last Lecture

- Why it is impractical to do experimental studies to find T(n)?
  - It is necessary to implement the algorithm, which may be difficult
  - Results may not be indicative of the running time on other inputs not included in the experiment.
  - In order to compare two algorithms, the same hardware and software environments must be used.

Asymptotic analysis

Measuring the Efficiency of an algorithm:

Determine its scalability:

- Uses a high-level description of the algorithm instead of an implementation
- characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment.

Asymptotic Analysis

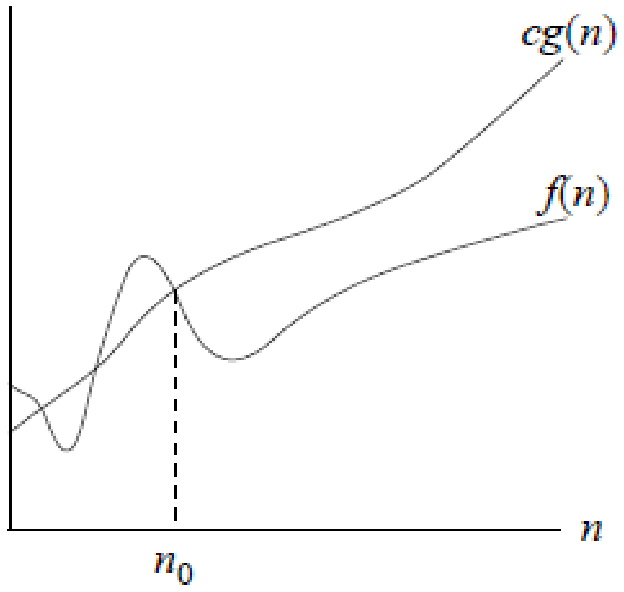
- A way to describe behavior of functions *in the limit*. We're studying ***asymptotic*** efficiency.
- Describe *growth* of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- How we indicate running times of algorithms.
- A way to compare "sizes" of functions:

O	$\approx$	$\leq$
Ω	$\approx$	$\geq$
Θ	$\approx$	$=$
o	$\approx$	$<$
ω	$\approx$	$>$

Big O

O-notation

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$



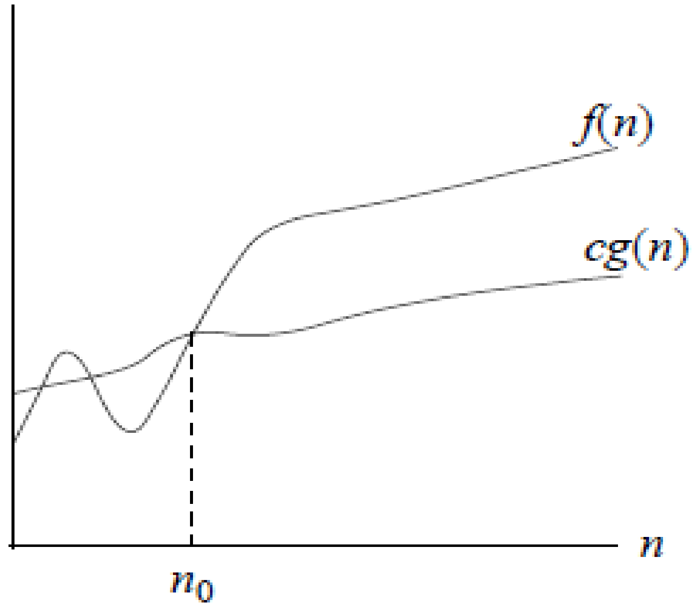
$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$

Big Ω

Ω-notation

$\Omega(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$.

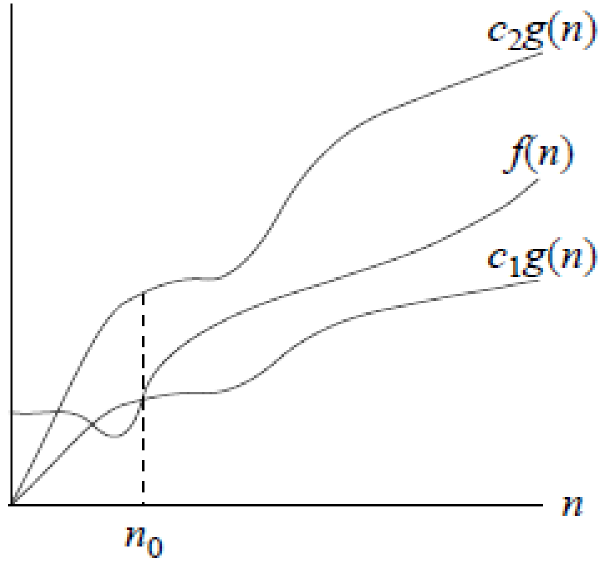


$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Big Θ

Θ-notation

$\Theta(g(n)) = \{ f(n) :$ there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Designing Algorithms

- There are many ways to design algorithms.
  - **Divide and conquer**
    - **Divide the problem into number of sub-problems that are smaller instances of the same problem.**
    - **Conquer the sub-problems by solving them recursively.**
    - **Combine the sub-problem solutions to give a solution to the original problem**

Merge Sort

- A sorting algorithm based on divide and conquer.
- Its worst case running time has lower order of growth than insertion sort.
- Because we are dealing with subproblems ,we state each subproblem as sorting a subarray A[p….r}. Initially, p=1 and r=n, but these values change as we recurse through subproblems.

Merge Sort

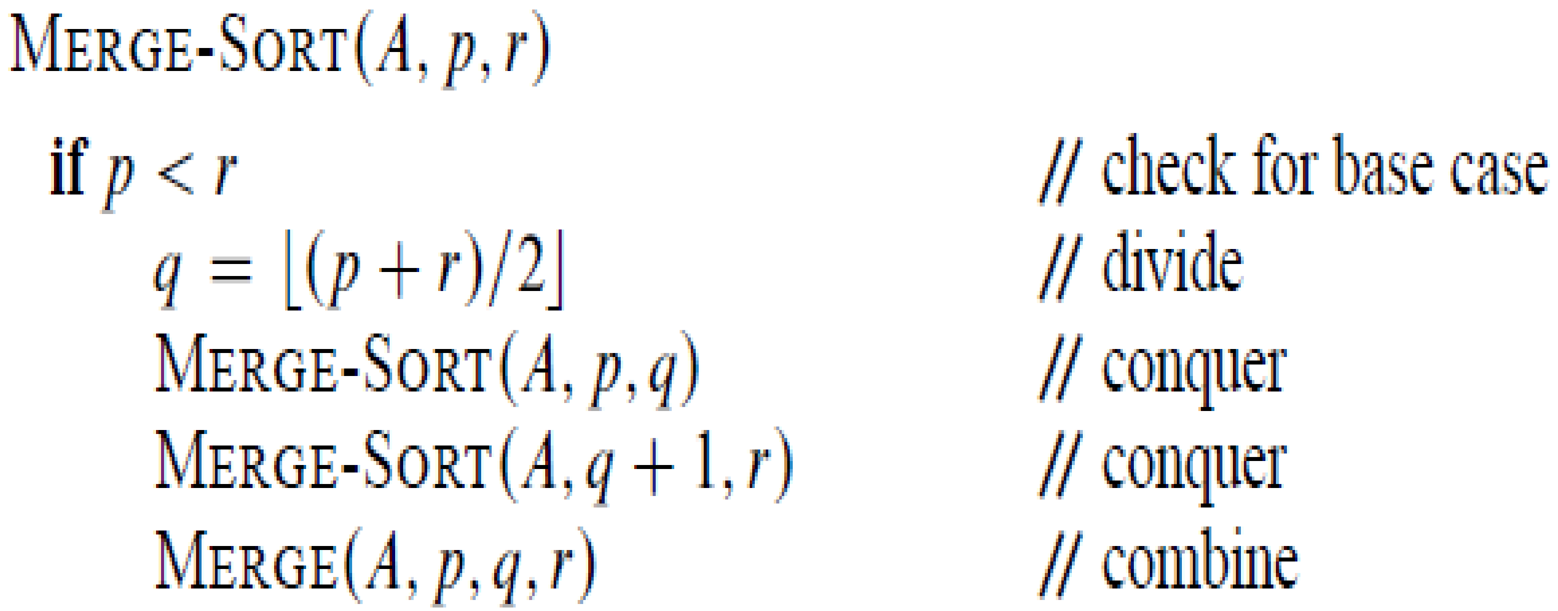
To Sort A[p…r]:

Divide by splitting into two sub-arrays A[p..q] and A[q+1...r].where q is the halfway point of A[p…r]

Conquer by recursively sorting the two sub-arrays A[p…q] and A[q+1……r].

Combine by merging the two sorted subarrays A[p….q] and A[q+1….r] to produce a single sorted subarray A[p…r]. To accomplish this step, we'll deifne a procedure Merge(A,p,q,r)

Merge Sort



```
MERGE-SORT$(A, p, r)$
  if $p < r$                                  // check for base case
      $q = \lfloor (p + r)/2 \rfloor$                 // divide
      MERGE-SORT$(A, p, q)$                   // conquer
      MERGE-SORT$(A, q + 1, r)$               // conquer
      MERGE$(A, p, q, r)$                     // combine
```

Merge Sort

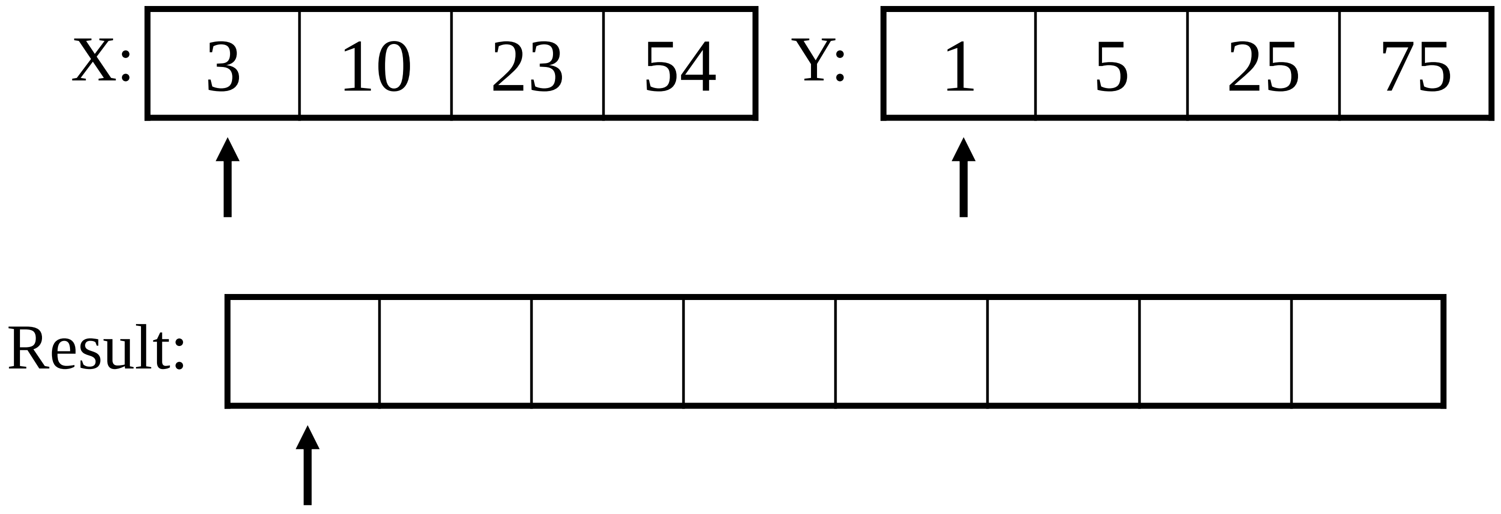
Merging

- The key to Merge Sort is merging two sorted lists into one, such that if you have two lists X $(x_1 \leq x_2 \leq \cdots \leq x_m)$ and Y$(y_1 \leq y_2 \leq \cdots \leq y_n)$ the resulting list is Z$(z_1 \leq z_2 \leq \cdots \leq z_{m+n})$
- Example:

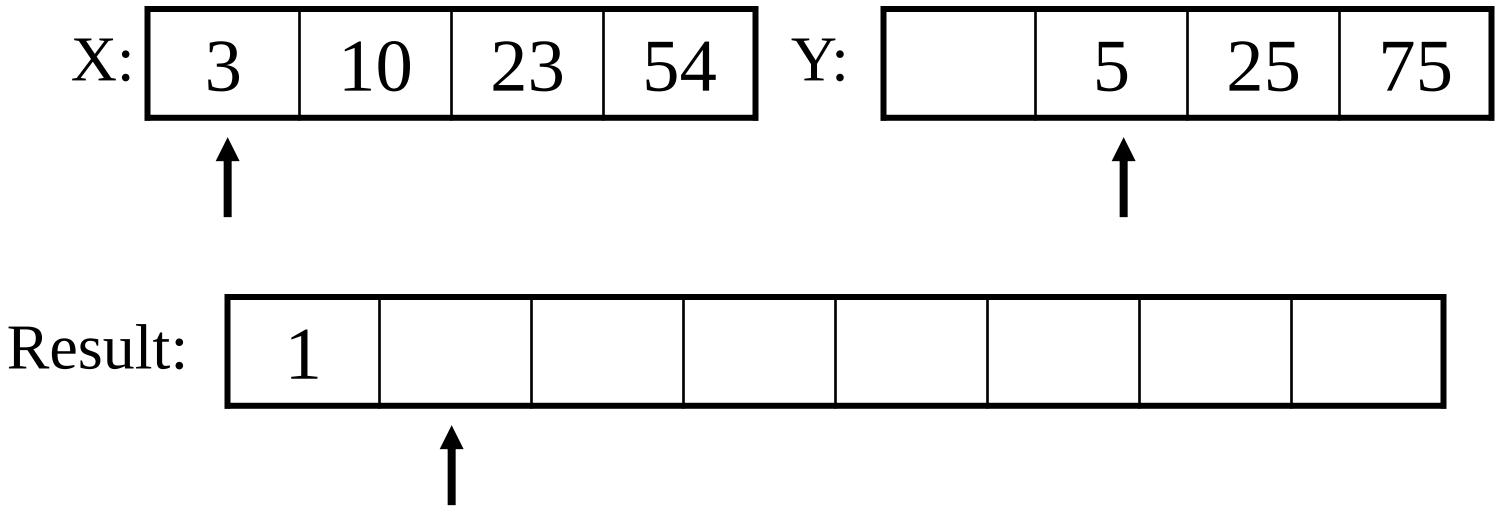
L_1 = { 3 8 9 } L_2 = { 1 5 7 }

merge(L_1, L_2) = { 1 3 5 7 8 9 }

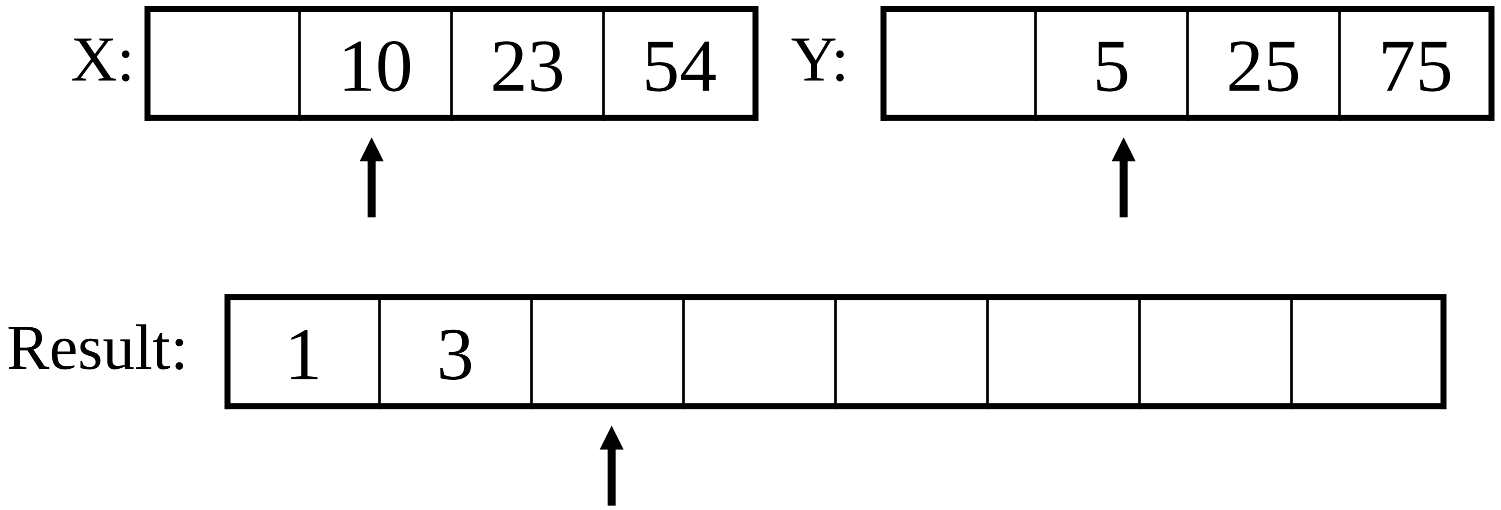
Merging (cont.)



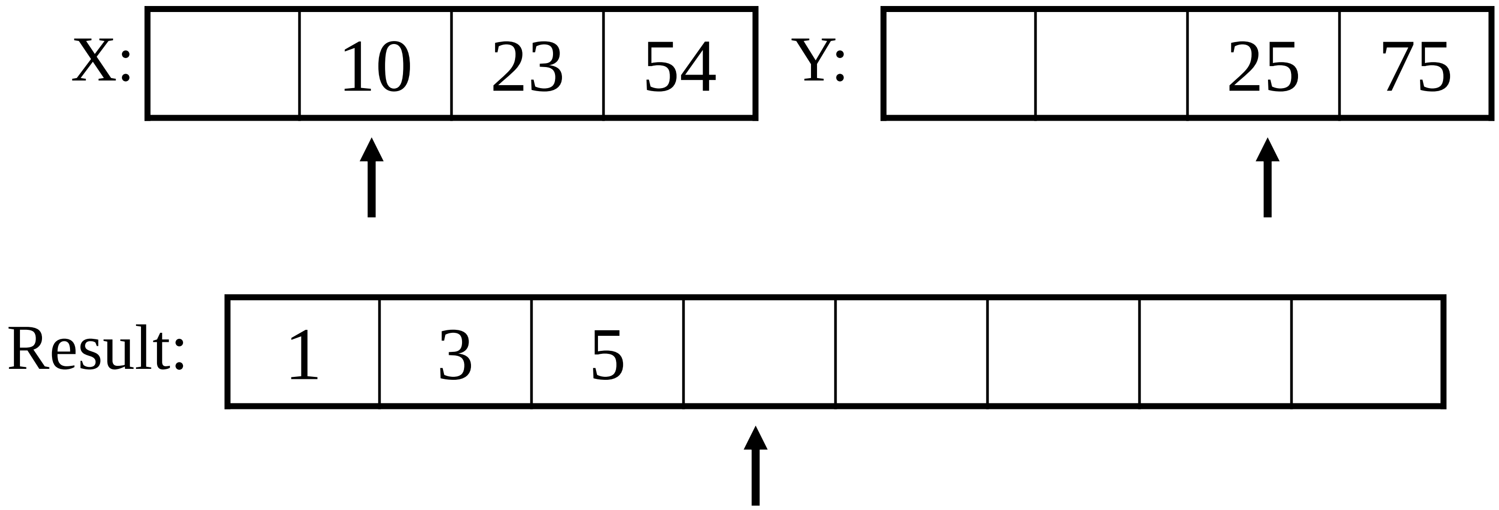
Merging (cont.)



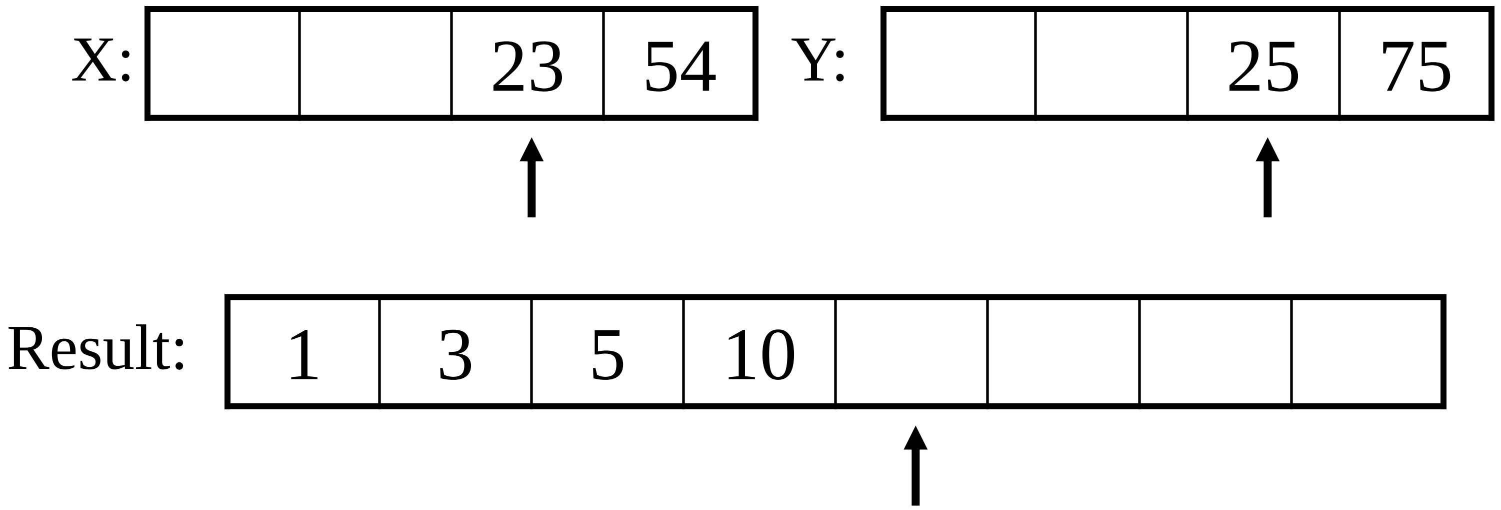
Merging (cont.)



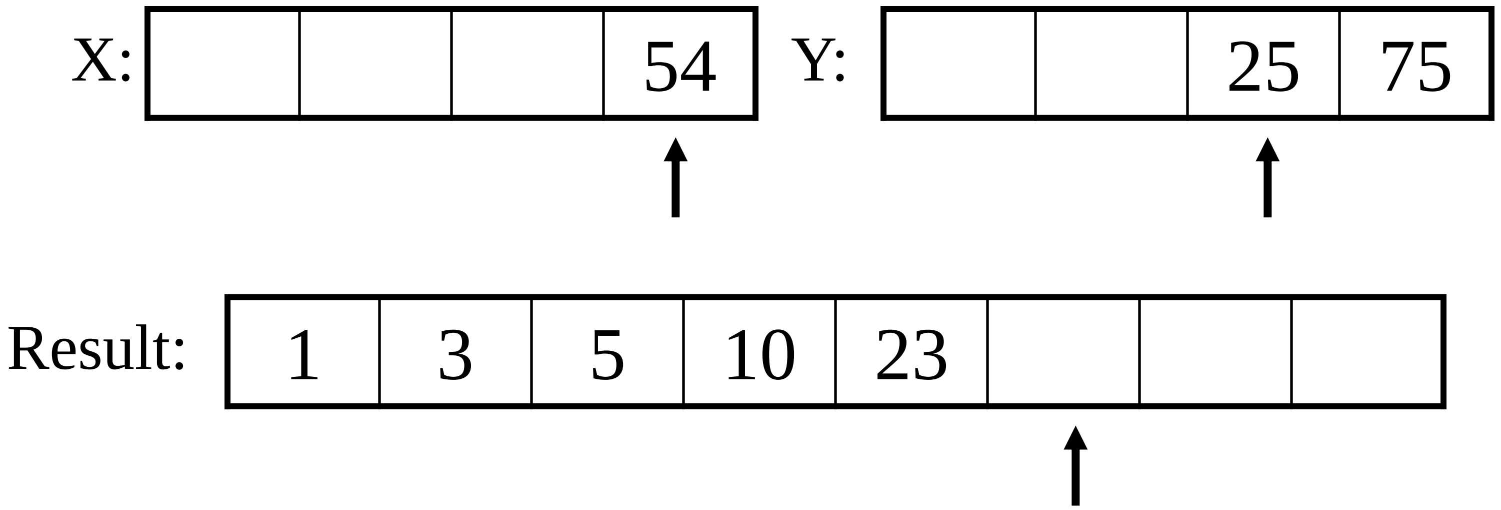
Merging (cont.)



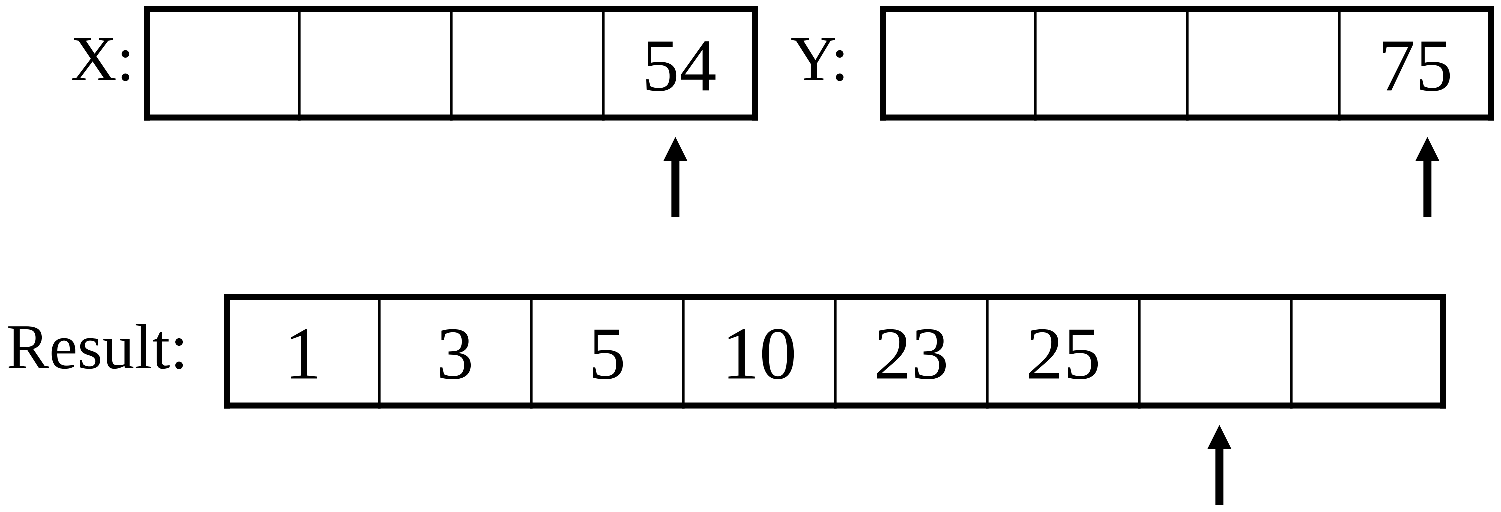
Merging (cont.)



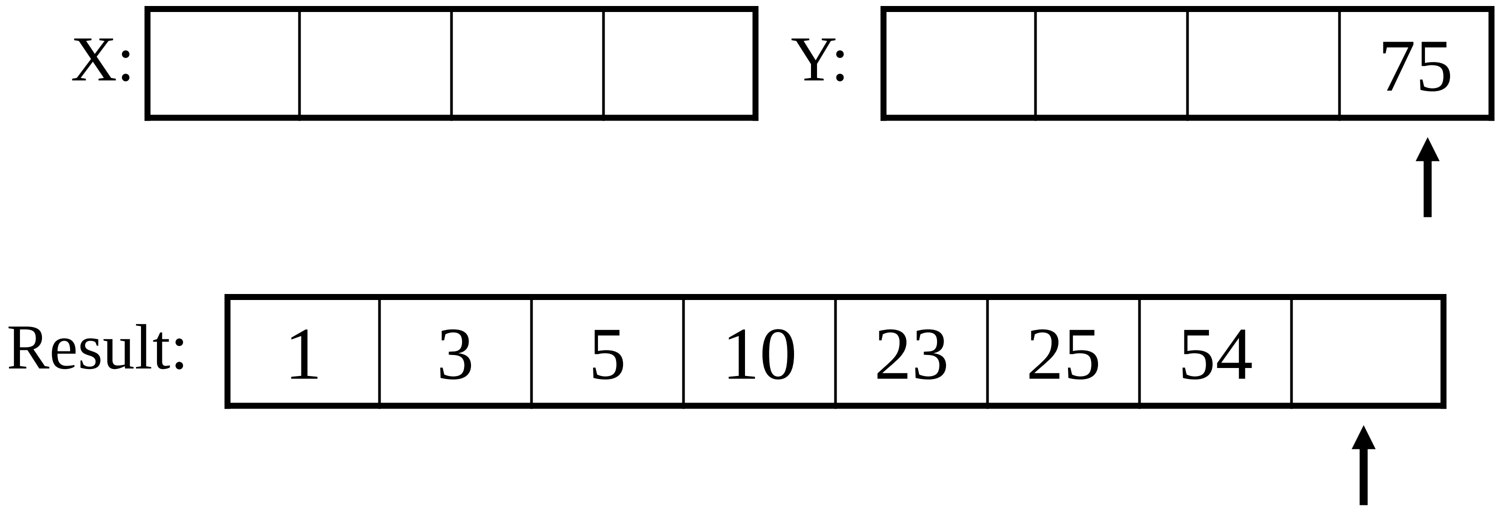
Merging (cont.)



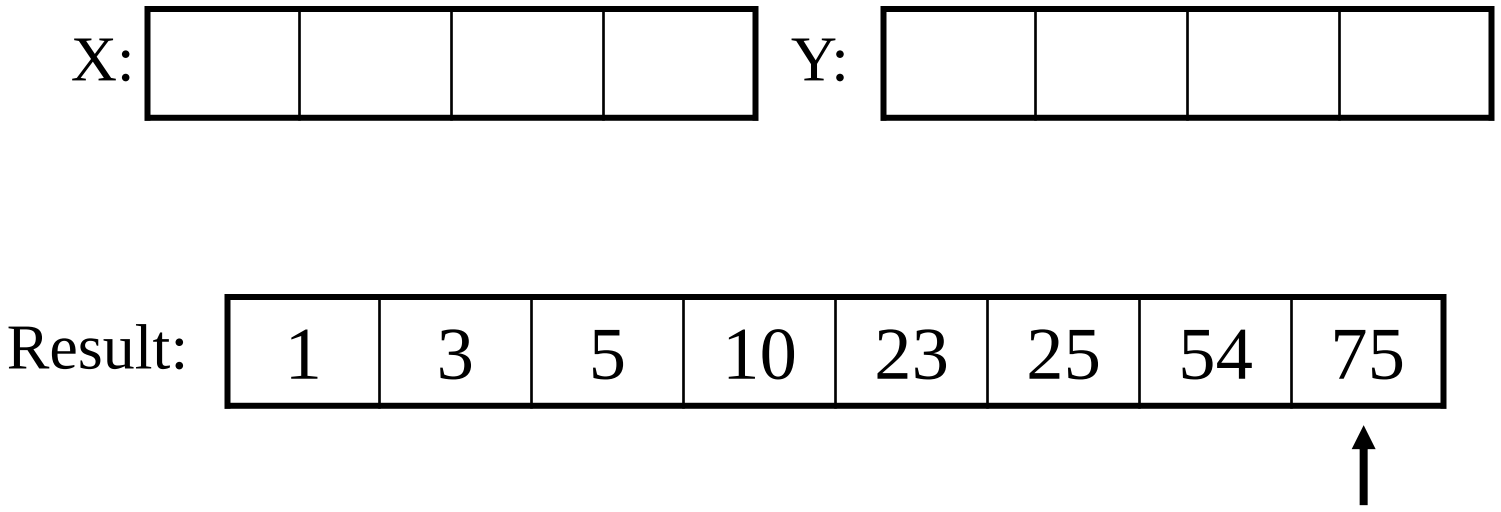
Merging (cont.)



Merging (cont.)



Merging (cont.)



Divide And Conquer

- Merging a two lists of one element each is the same as sorting them.
- Merge sort divides up an unsorted list until the above condition is met and then sorts the divided parts back together in pairs.
- Specifically this can be done by recursively dividing the unsorted list in half, merge sorting the right side then the left side and then merging the right and left back together.

Merge Sort Algorithm

Given a list L with a length k:

- If k == 1 → the list is sorted
- Else:
  - Merge Sort the left side (1 thru k/2)
  - Merge Sort the right side (k/2+1 thru k)
  - Merge the right side with the left side

Merge Sort Example

99	6	86	15	58	35	86	4	0

Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

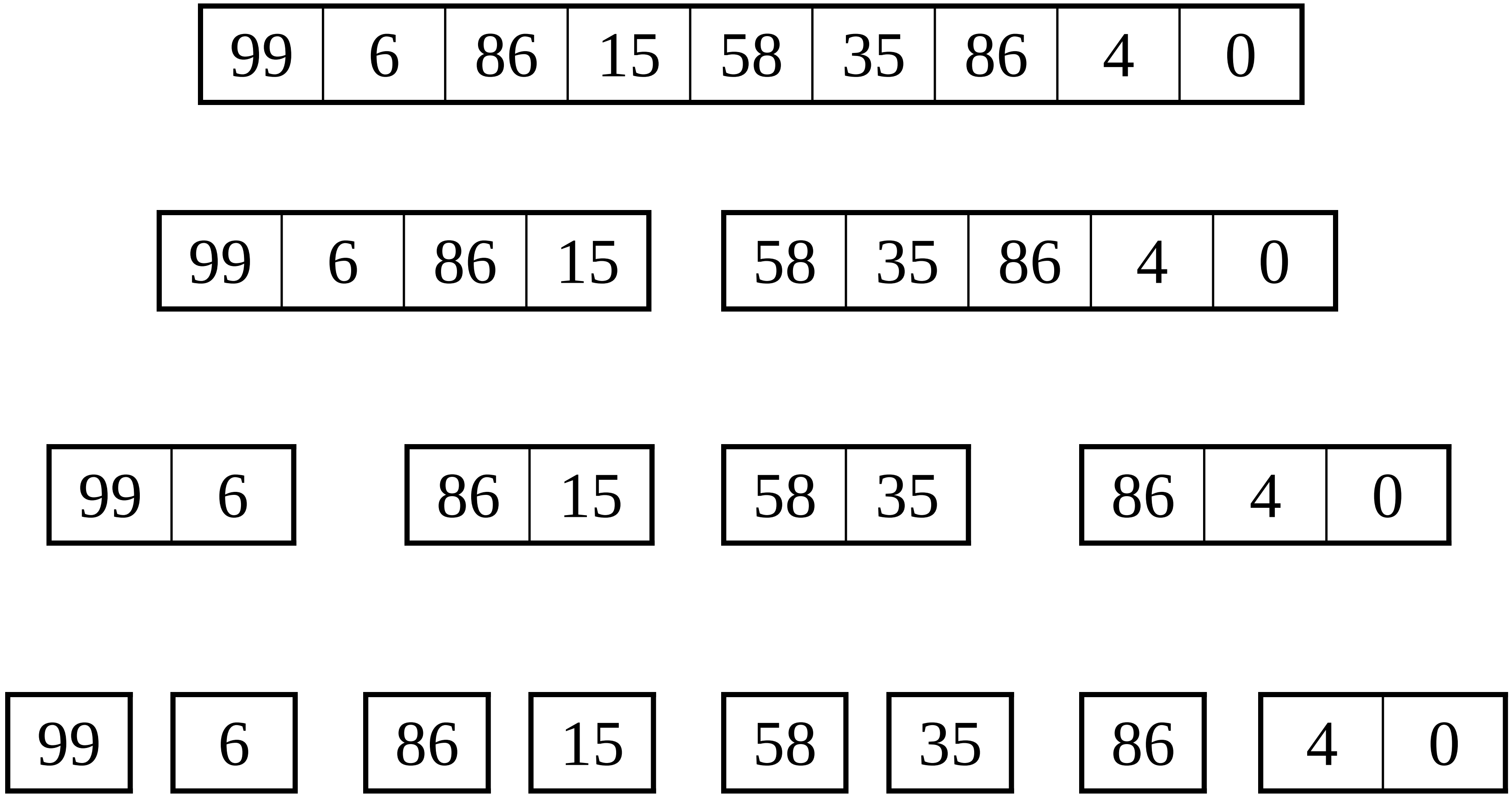
99	6

86	15

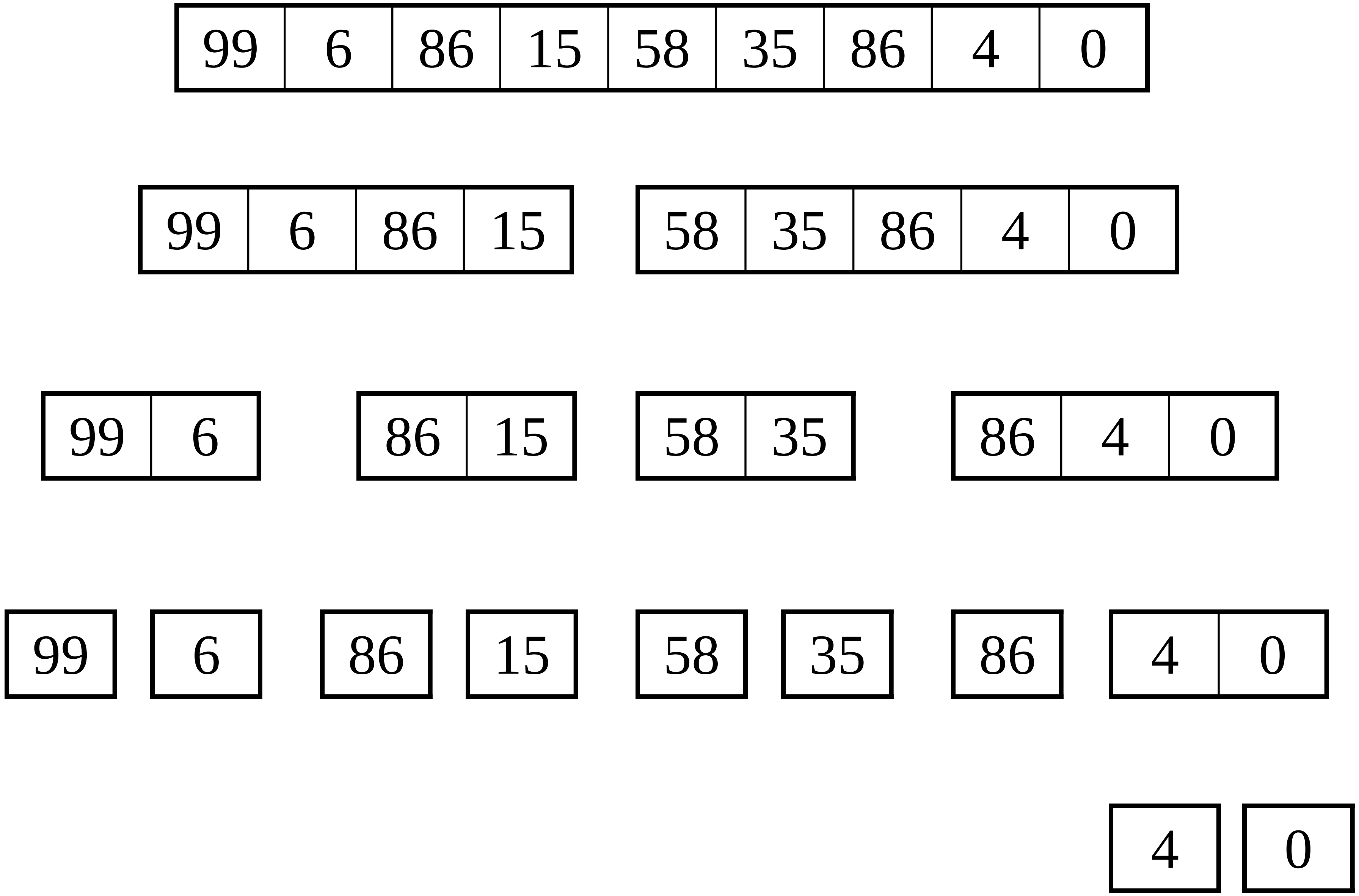
58	35

86	4	0

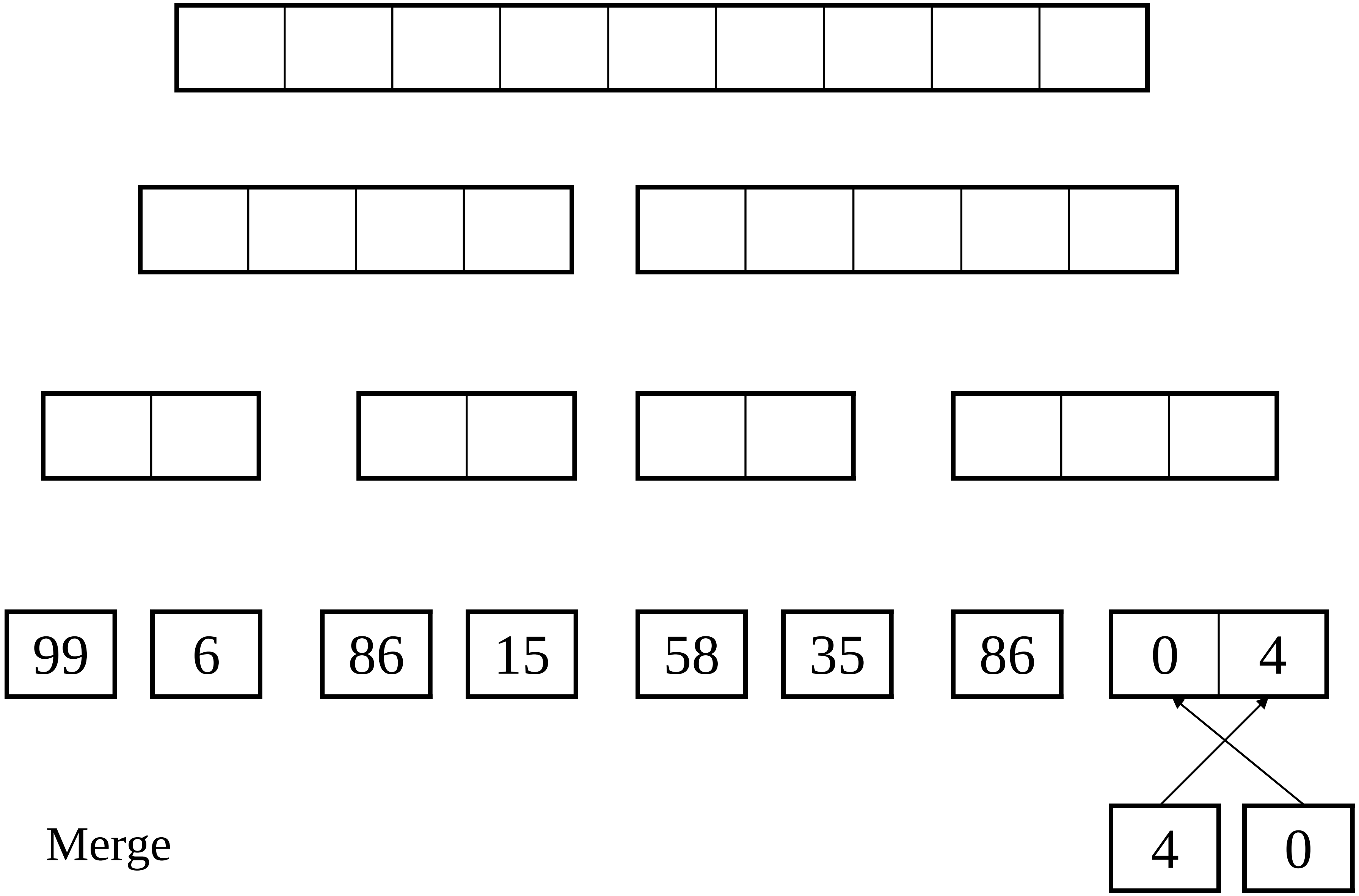
Merge Sort Example



Merge Sort Example

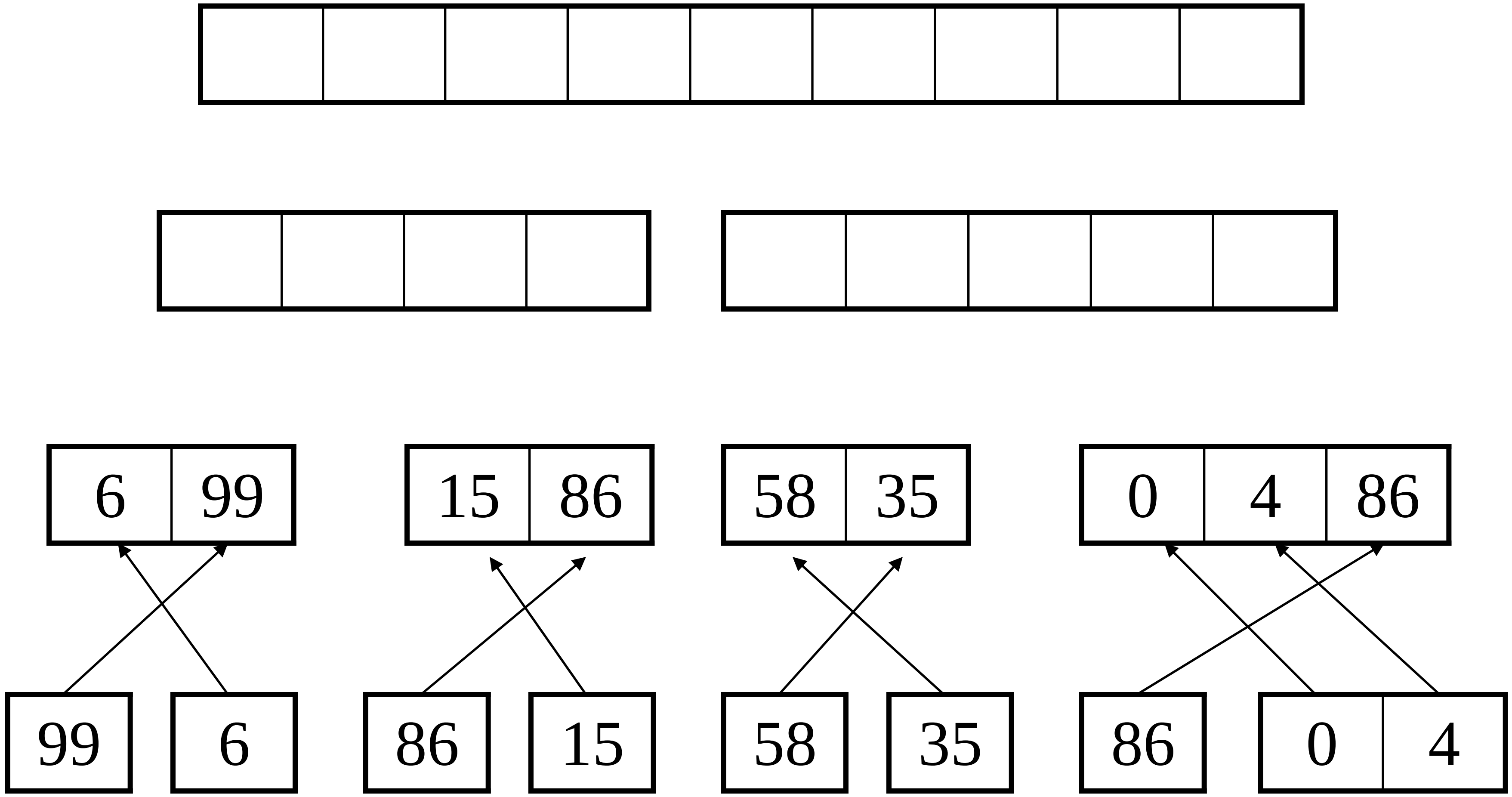


Merge Sort Example



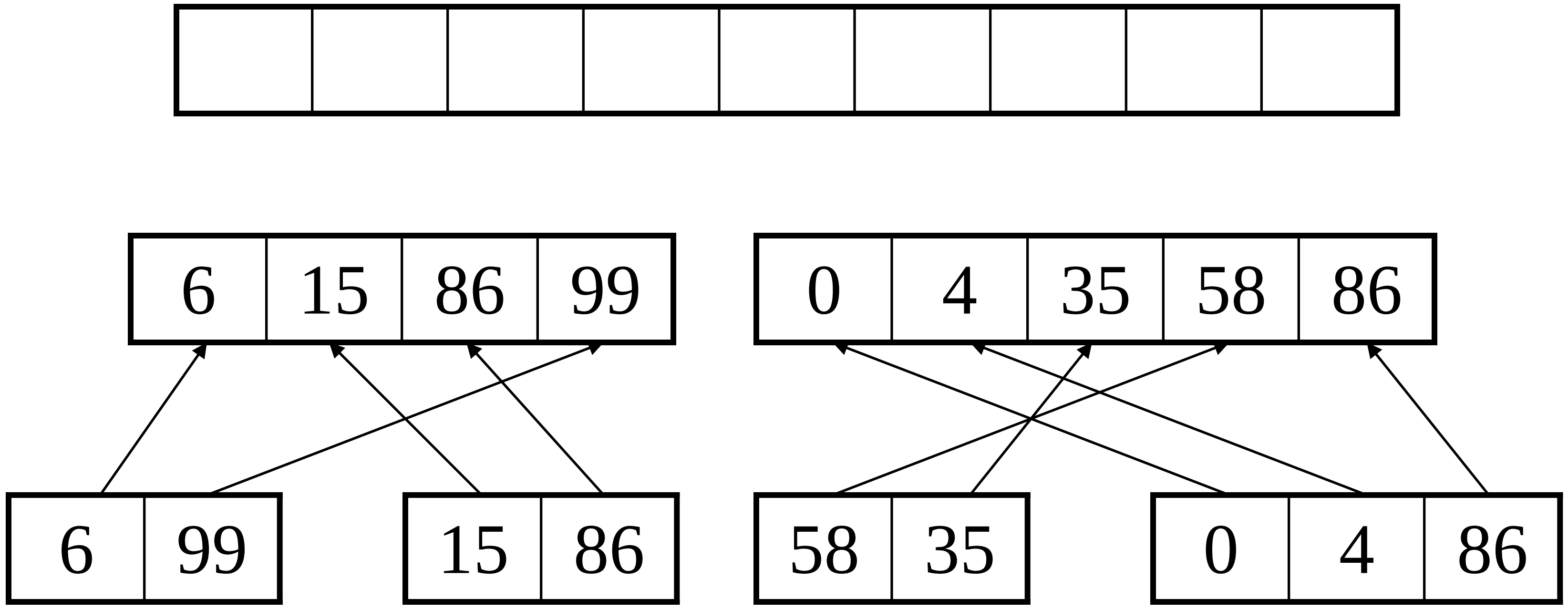
Merge

Merge Sort Example



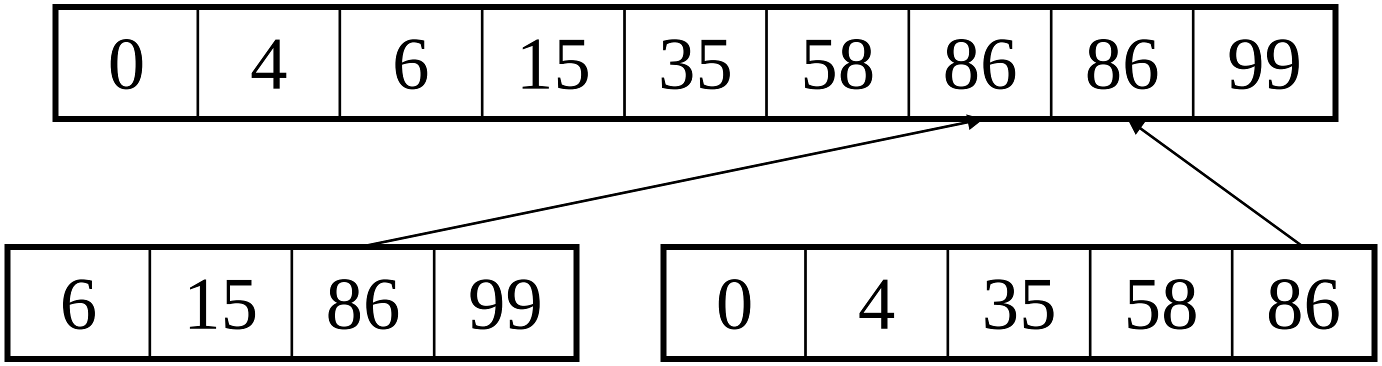
Merge

Merge Sort Example



Merge

Merge Sort Example



Merge

Merge Sort Example

0	4	6	15	35	58	86	86	99