CSE 373: Data Structures and Algorithms

Lecture 6: Sorting

Why Sorting?

- Practical application
  - People by last name
  - Countries by population
  - Search engine results by relevance

- Fundamental to other algorithms

- Different algorithms have different asymptotic and constant-factor trade-offs
  - No single 'best' sort for all scenarios
  - Knowing one way to sort just isn't enough

- Many to approaches to sorting which can be used for other problems

Problem statement

There are *n* comparable elements in an array and we want to rearrange them to be in increasing order

Pre:

- An array `A` of data records
- A value in each data record
- A comparison function
  - <, =, >, compareTo

Post:

- For each distinct position `i` and `j` of `A`, if `i < j` then `A[i] ≤ A[j]`
- `A` has all the same data it started with

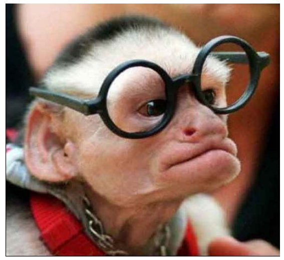
Sorting Classification

In memory sorting			External sorting
Comparison sorting $\Omega(N \log N)$		Specialized Sorting	
$O(N^2)$	$O(N \log N)$	$O(N)$	# of tape accesses
• Bubble Sort • Selection Sort • Insertion Sort • Shellsort Sort	• Merge Sort • Quick Sort • Heap Sort	• Bucket Sort • Radix Sort	• Simple External Merge Sort • Variations

in place? stable?

Comparison Sorting

comparison-based sorting: determine order through comparison operations on the input data:
<, >, compareTo, …



Bogo sort

- **bogo sort**: orders a list of values by repetitively shuffling them and checking if they are sorted

- more specifically:
  - scan the list, seeing if it is sorted
  - if not, shuffle the values in the list and repeat

- This sorting algorithm has terrible performance!
  - Can we deduce its runtime?

Bogo sort code

```
public static void bogoSort(int[] a) {
    while (!isSorted(a)) {
        shuffle(a);
    }
}

// Returns true if array a's elements
// are in sorted order.
public static boolean isSorted(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        if (a[i] > a[i+1]) {
            return false;
        }
    }

    return true;
}
```

Bogo sort code, helpers

```
// Shuffles an array of ints by randomly swapping each
// element with an element ahead of it in the array.
public static void shuffle(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // pick random number in [i+1, a.length-1] inclusive
        int range = a.length-1 - (i + 1) + 1;
        int j = (int)(Math.random() * range + (i + 1));
        swap(a, i, j);
    }
}

// Swaps a[i] with a[j].
private static void swap(int[] a, int i, int j) {
    if (i == j)
        return;

    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

Bogo sort runtime

- How long should we expect bogo sort to take?
  - related to probability of shuffling into sorted order
  - assuming shuffling code is fair, probability equals 1 / (number of permutations of *n* elements)

$$P_n^n = n!$$

  - bogo sort takes roughly factorial time to run
    - note that if array is initially sorted, bogo finishes quickly!
  - it should be clear that this is not satisfactory...

$O(n^2)$ Comparison Sorting

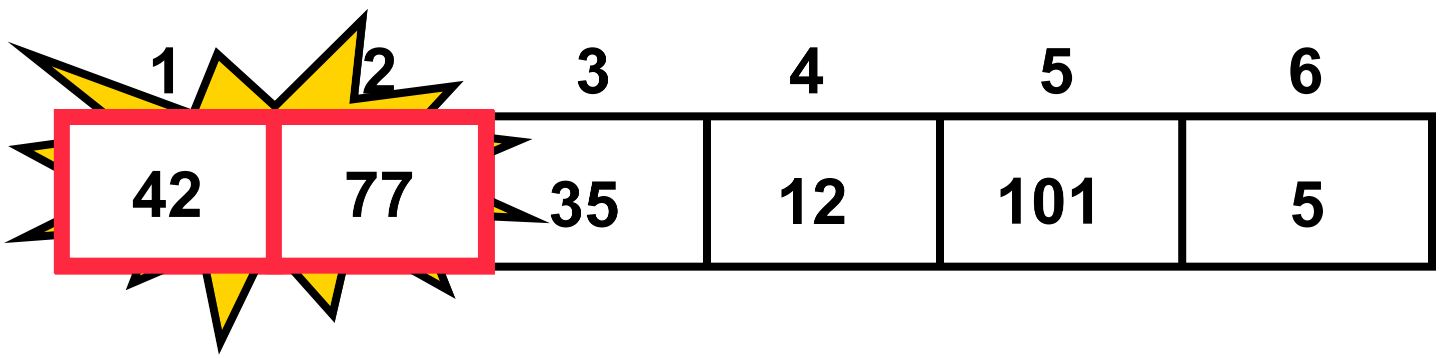
Bubble sort

- **bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary

- more specifically:
  - scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
  - scan the list again, bubbling up the second highest value
  - repeat until all elements have been placed in their proper order

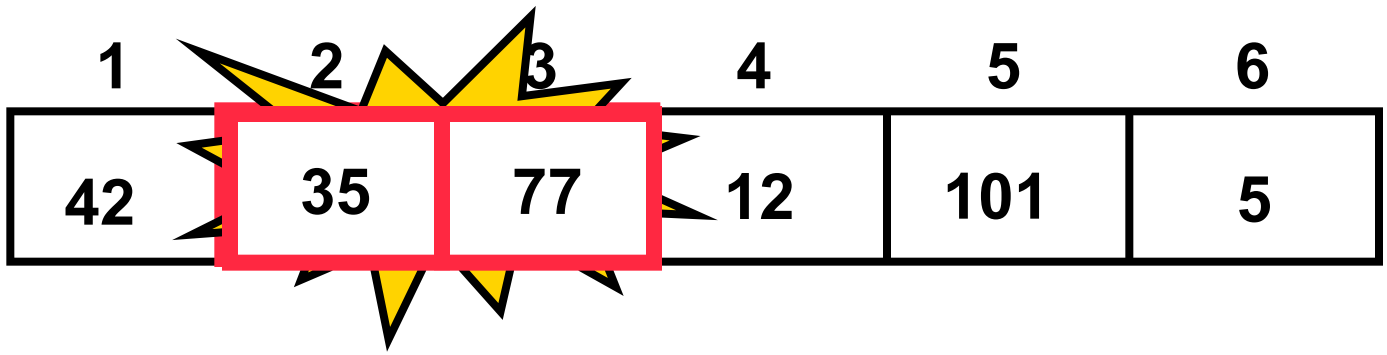
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



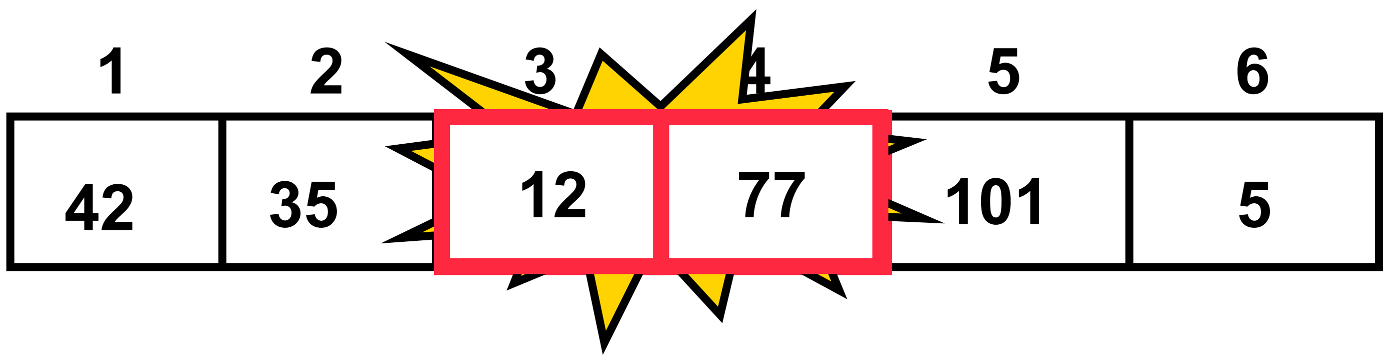
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling" largest element

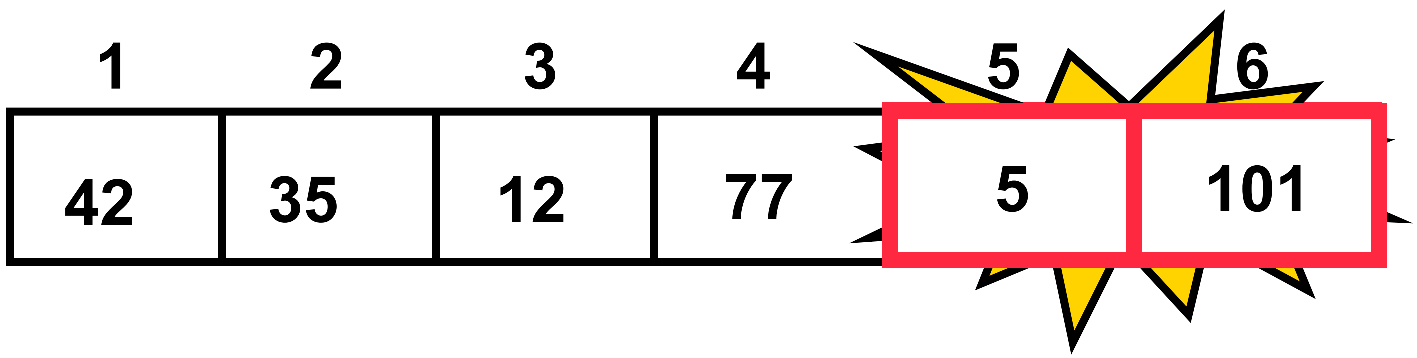
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
42	35	12	77	101	5

No need to swap

"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

Bubble sort code

```
public static void bubbleSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        for (int j = 1; j < a.length - i; j++) {
            // swap adjacent out-of-order elements
            if (a[j-1] > a[j]) {
                swap(a, j-1, j);
            }
        }
    }
}
```

Bubble sort runtime

- Running time (# comparisons) for input size *n*:

$$\sum_{i=0}^{n-1} \sum_{j=1}^{n-i} 1 = \sum_{i=0}^{n-1} (n-i)$$

$$= \sum_{i=0}^{n-1} i$$

$$= \frac{(n-1)n}{2}$$

$$= \mathrm{O}(n^2)$$

  – number of actual swaps performed depends on the data; out-of-order data performs many swaps

Selection sort

- **selection sort**: orders a list of values by repetitively putting a particular value into its final position

- more specifically:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places

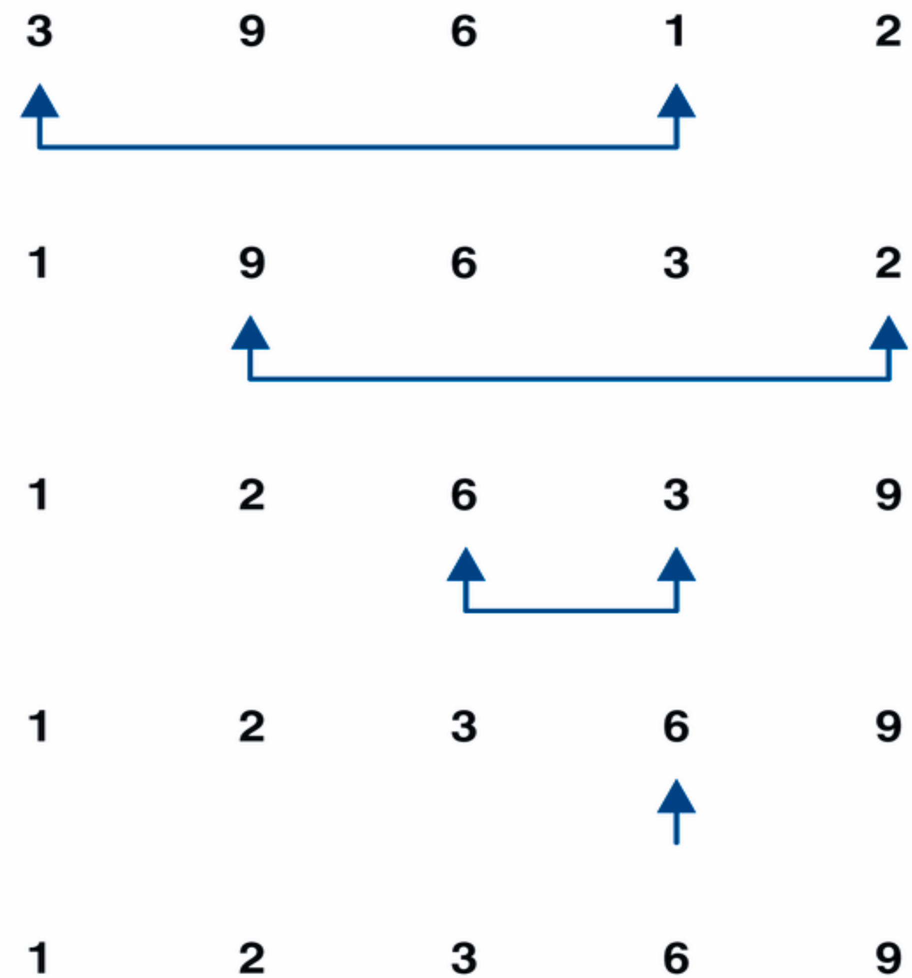
Selection sort example

Scan right starting with 3.
1 is the smallest. Exchange 1 and 3.

Scan right starting with 9.
2 is the smallest. Exchange 9 and 2.

Scan right starting with 6.
3 is the smallest. Exchange 6 and 3.

Scan right starting with 6.
6 is the smallest. Exchange 6 and 6.



Selection sort example 2

Index	0	1	2	3	4	5	6	7
Value	27	63	1	72	64	58	14	9
1st pass	**1**	63	27	72	64	58	14	9
2nd pass	1	**9**	27	72	64	58	14	63
3rd pass	1	9	**14**	72	64	58	27	63
…								

Selection sort code

```
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        // find index of smallest element
        int min = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }

        // swap smallest element with a[i]
        swap(a, i, min);
    }
}
```

Selection sort runtime

- Running time for input size *n*:
  - in practice, a bit faster than bubble sort. Why?

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=0}^{n-1} (n-(i+1)+1)$$

$$= \sum_{i=0}^{n-1} (n-i)$$

$$= \sum_{i=0}^{n-1} i$$

$$= \frac{(n-1)n}{2}$$

$$= \mathrm{O}(n^2)$$

Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert the second item into the sorted sublist, shifting the first item if needed
  - insert the third item into the sorted sublist, shifting the other items as needed
  - repeat until all values have been inserted into their proper positions

Insertion sort

- Simple sorting algorithm.
  – n-1 passes over the array
  – At the end of pass *i*, the elements that occupied A [0]…A[*i*] originally are still in those spots and in sorted order.

2	15	8	1	17	10	12	5
0	1	2	3	4	5	6	7

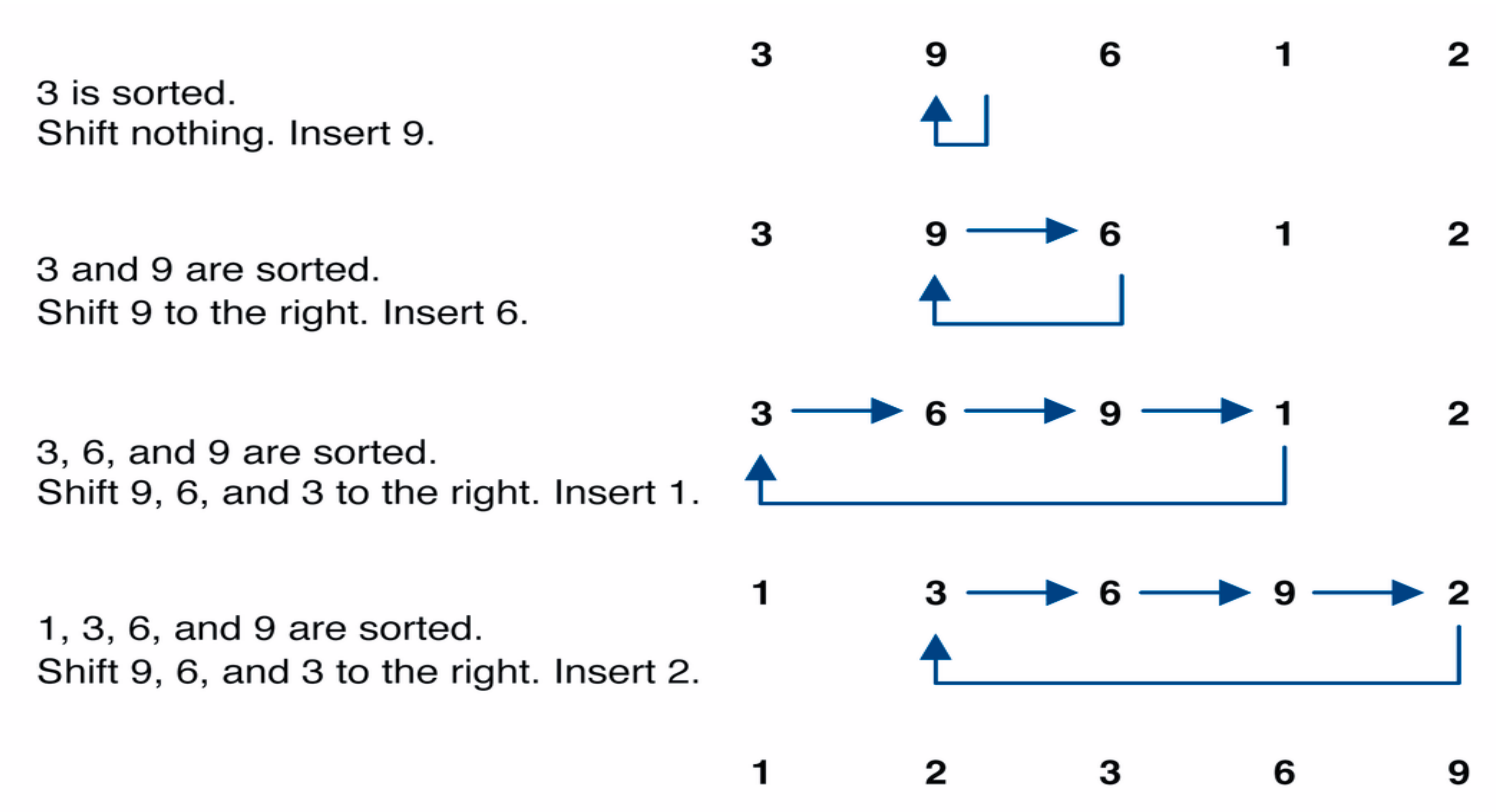
after pass 2

2	8	15	1	17	10	12	5
0	1	2	3	4	5	6	7

after pass 3

1	2	8	15	17	10	12	5
0	1	2	3	4	5	6	7

Insertion sort example



Insertion sort code

```
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int temp = a[i];

        // slide elements down to make room for a[i]
        int j = i;
        while (j > 0 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }

        a[j] = temp;
    }
}
```

Insertion sort runtime

- worst case: reverse-ordered elements in array.

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1)n}{2}$$

$$= O(n^2)$$

- best case: array is in sorted ascending order.

$$\sum_{i=1}^{n-1} 1 = n - 1 = O(n)$$

- average case: each element is about halfway in order.

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2}(1 + 2 + 3 \ldots + (n-1)) = \frac{(n-1)n}{4}$$

$$= O(n^2)$$

Comparing sorts

- We've seen "simple" sorting algos. so far, such as:
  - selection sort
  - insertion sort

	comparisons	swaps
selection	$n^2/2$	n
insertion	worst: $n^2/2$ best: n	worst: $n^2/2$ best: n

- They all use nested loops and perform approximately n^2 comparisons
- They are relatively inefficient

Average case analysis

- Given an array A of elements, an *inversion* is an ordered pair (i, j) such that i < j, but A[i] > A[j]. (out of order elements)
- Assume no duplicate elements.

- Theorem: The average number of inversions in an array of n distinct elements is n (n - 1) / 4.
- Corollary: Any algorithm that sorts by exchanging adjacent elements requires O(n^2) time on average.

Shell sort description

- **shell sort**: orders a list of values by comparing elements that are separated by a gap of >1 indexes
  - a generalization of insertion sort
  - invented by computer scientist Donald Shell in 1959

- based on some observations about insertion sort:
  - insertion sort runs fast if the input is almost sorted
  - insertion sort's weakness is that it swaps each element just one step at a time, taking many swaps to get the element into its correct position

Shell sort example

- Idea: Sort all elements that are 5 indexes apart, then sort all elements that are 3 indexes apart, ...

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	72	82	93	95	15 swaps

Shell sort code

```
public static void shellSort(int[] a) {
    for (int gap = a.length / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < a.length; i++) {
            // slide element i back by gap indexes
            // until it's "in order"
            int temp = a[i];
            int j = i;
            while (j >= gap && temp < a[j - gap]) {
                a[j] = a[j - gap];
                j -= gap;
            }
            a[j] = temp;
        }
    }
}
```

Sorting practice problem

- Consider the following array of int values.

```
[22, 11, 34, -5,  3, 40,  9, 16,  6]
```

(a) Write the contents of the array after 3 passes of the outermost loop of bubble sort.

(b) Write the contents of the array after 5 passes of the outermost loop of insertion sort.

(c) Write the contents of the array after 4 passes of the outermost loop of selection sort.

(d) Write the contents of the array after a pass of bogo sort. (Just kidding.)

O(n log n) Comparison Sorting

Merge sort

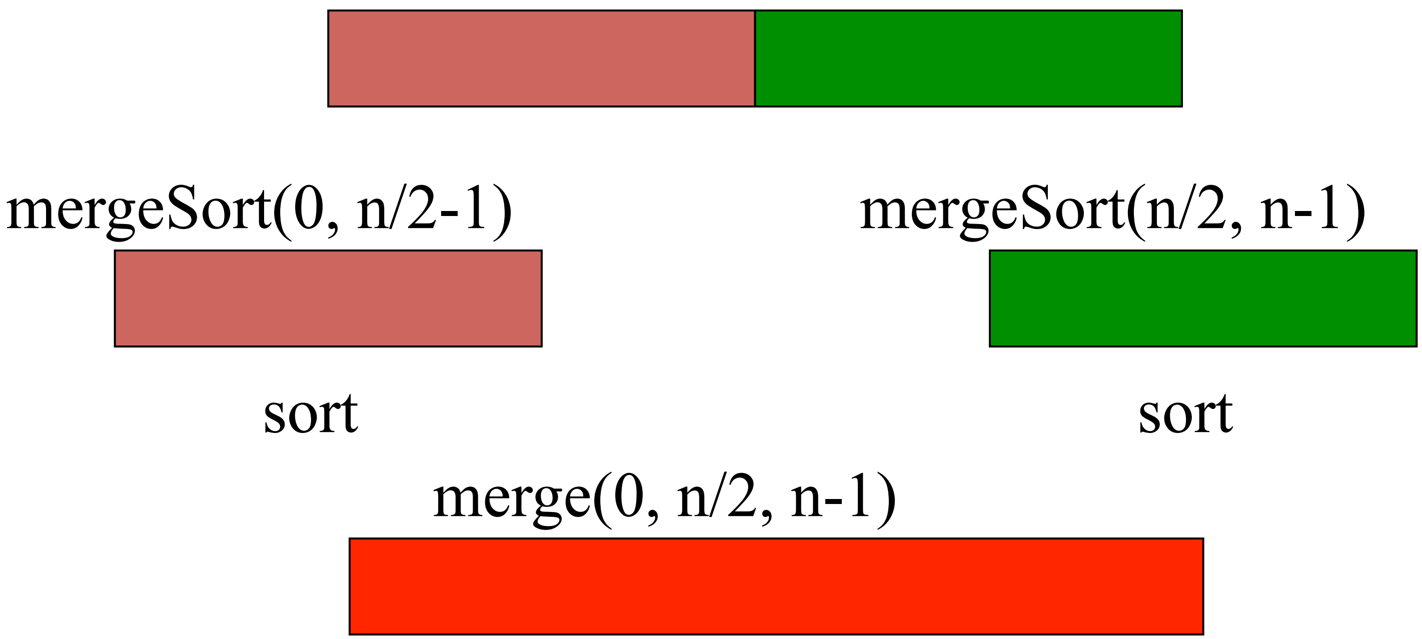
- **merge sort**: orders a list of values by recursively dividing the list in half until each sub-list has one element, then recombining
  - Invented by John von Neumann in 1945

- more specifically:
  - divide the list into two roughly equal parts
  - recursively divide each part in half, continuing until a part contains only one element
  - merge the two parts into one sorted list
  - continue to merge parts as the recursion unfolds

- This is a "divide and conquer" algorithm.

Merge sort

- Merge sort idea:
  - Divide the array into two halves.
  - Recursively sort the two halves (using merge sort).
  - Use merge to combine the two arrays.

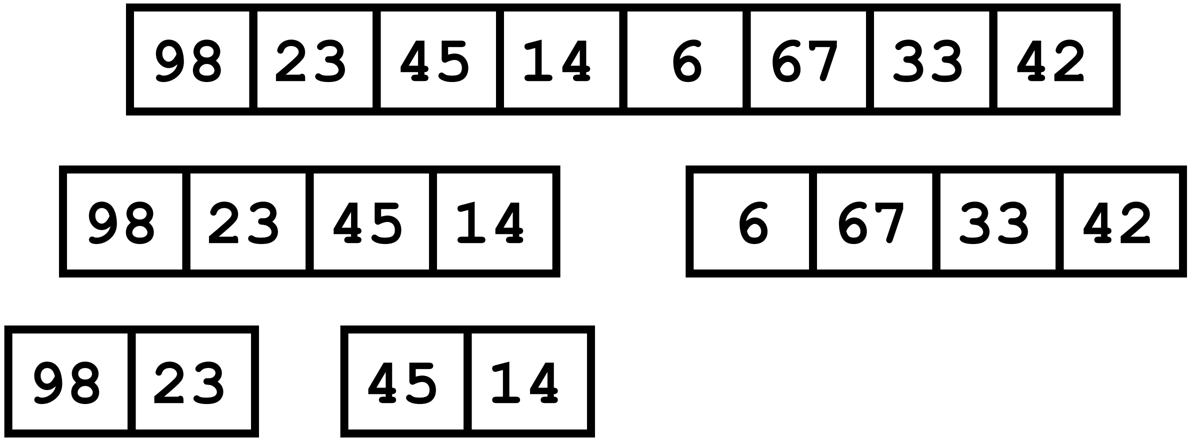


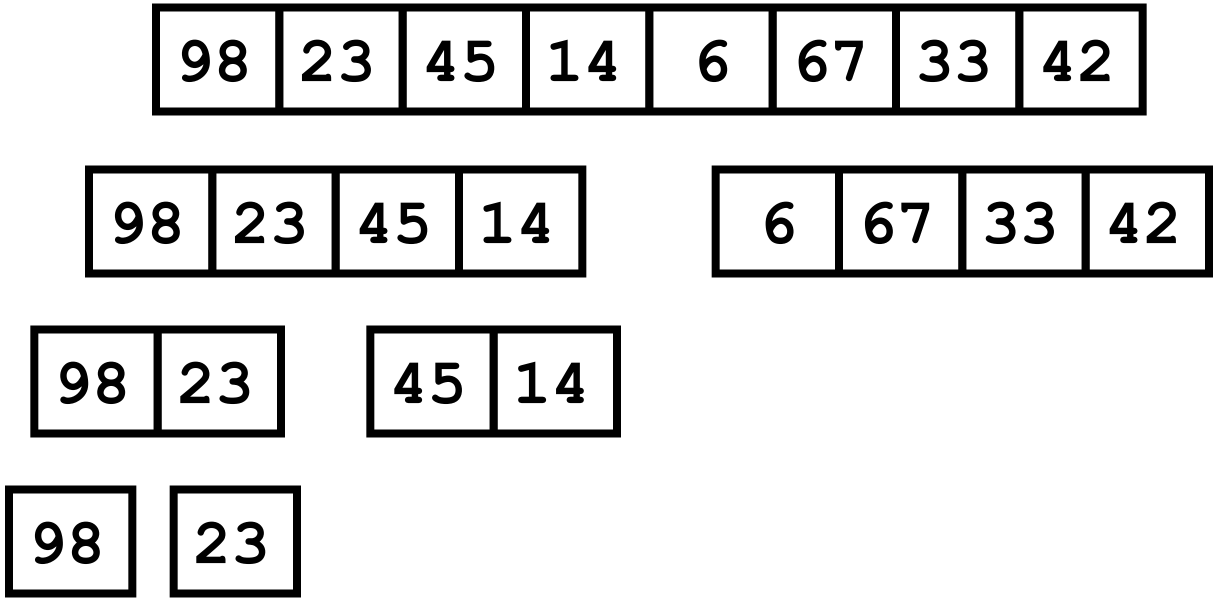
98	23	45	14	6	67	33	42

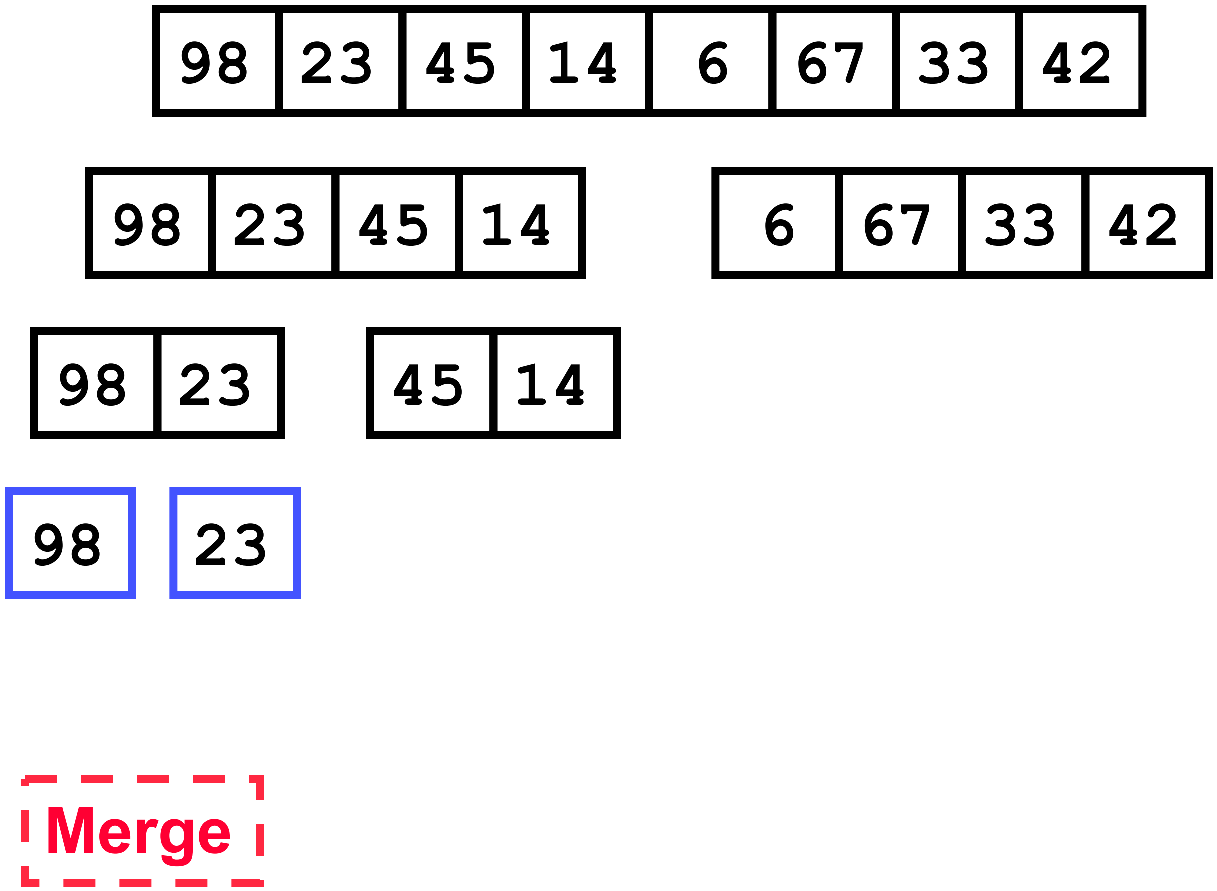
98	23	45	14	6	67	33	42

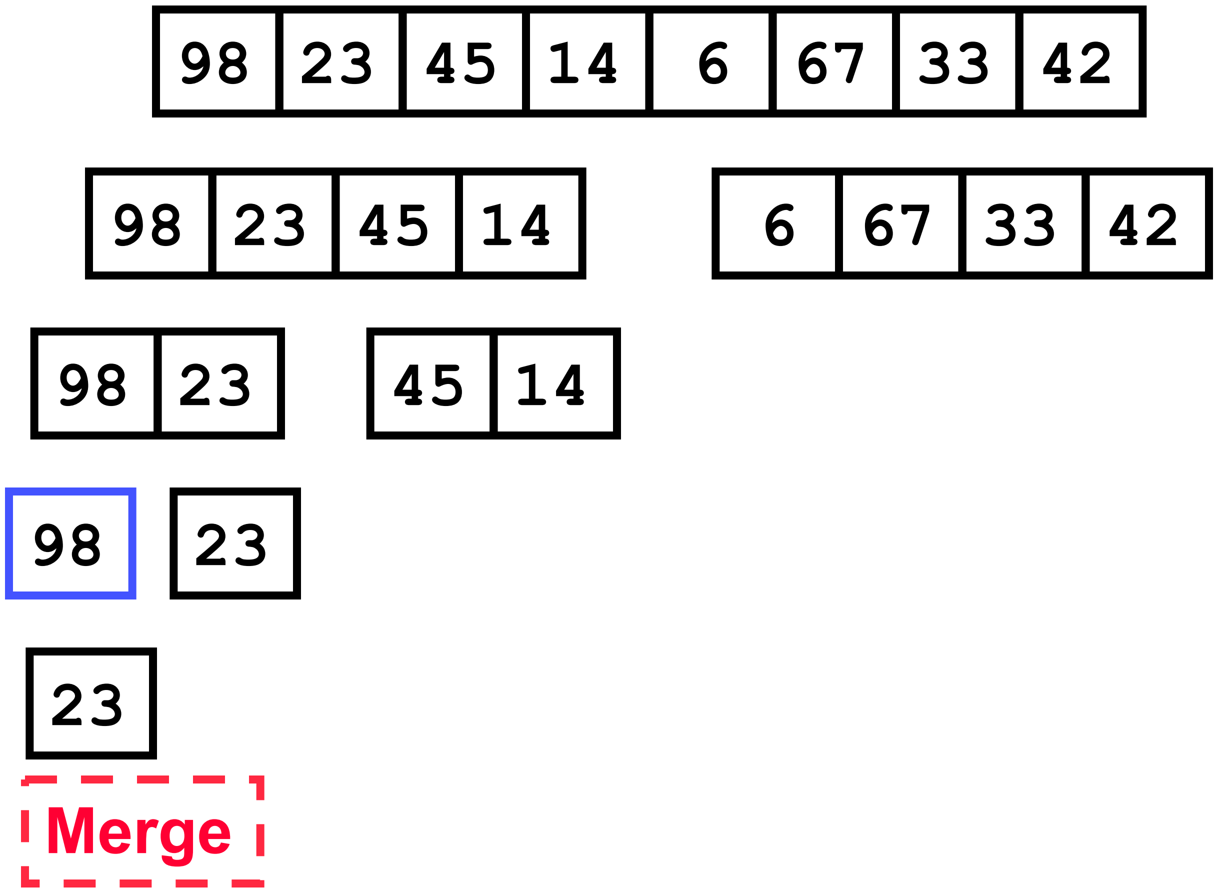
98	23	45	14

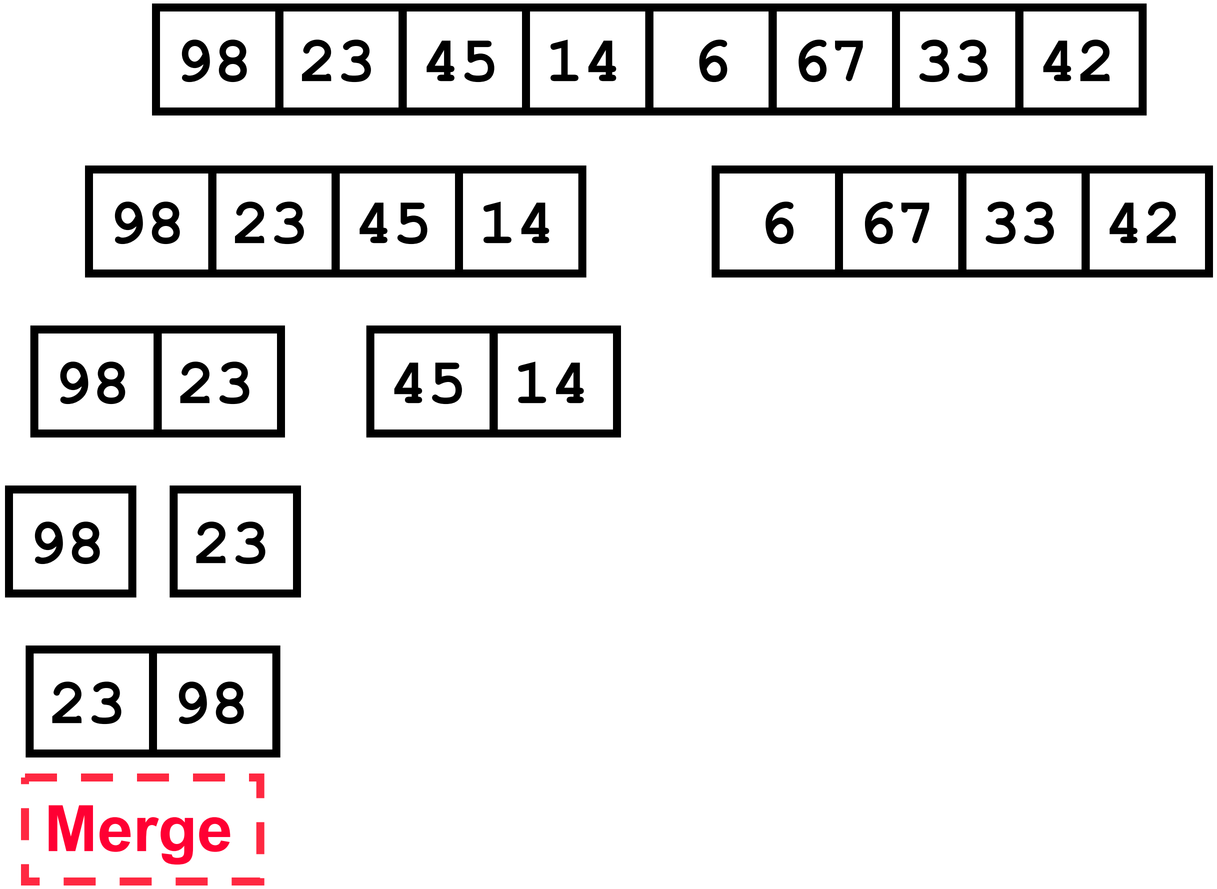
6	67	33	42

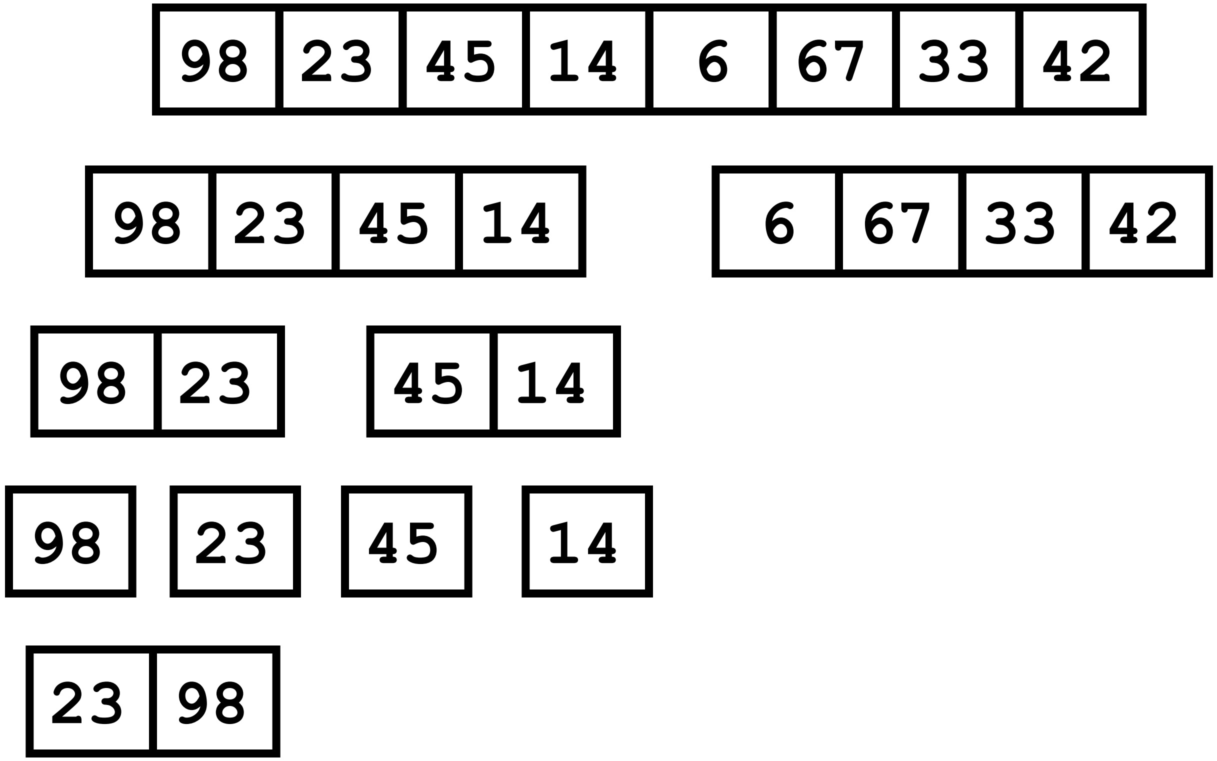

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

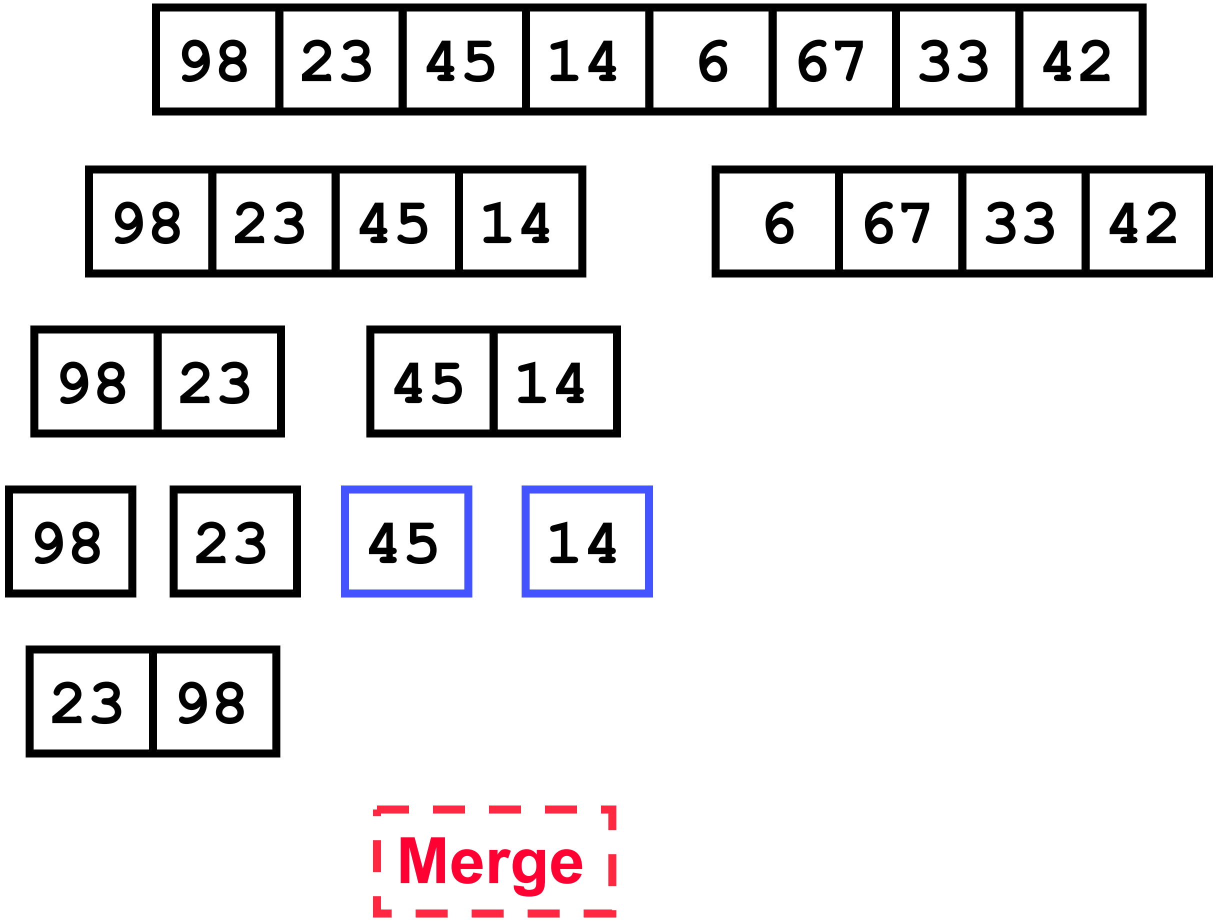

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

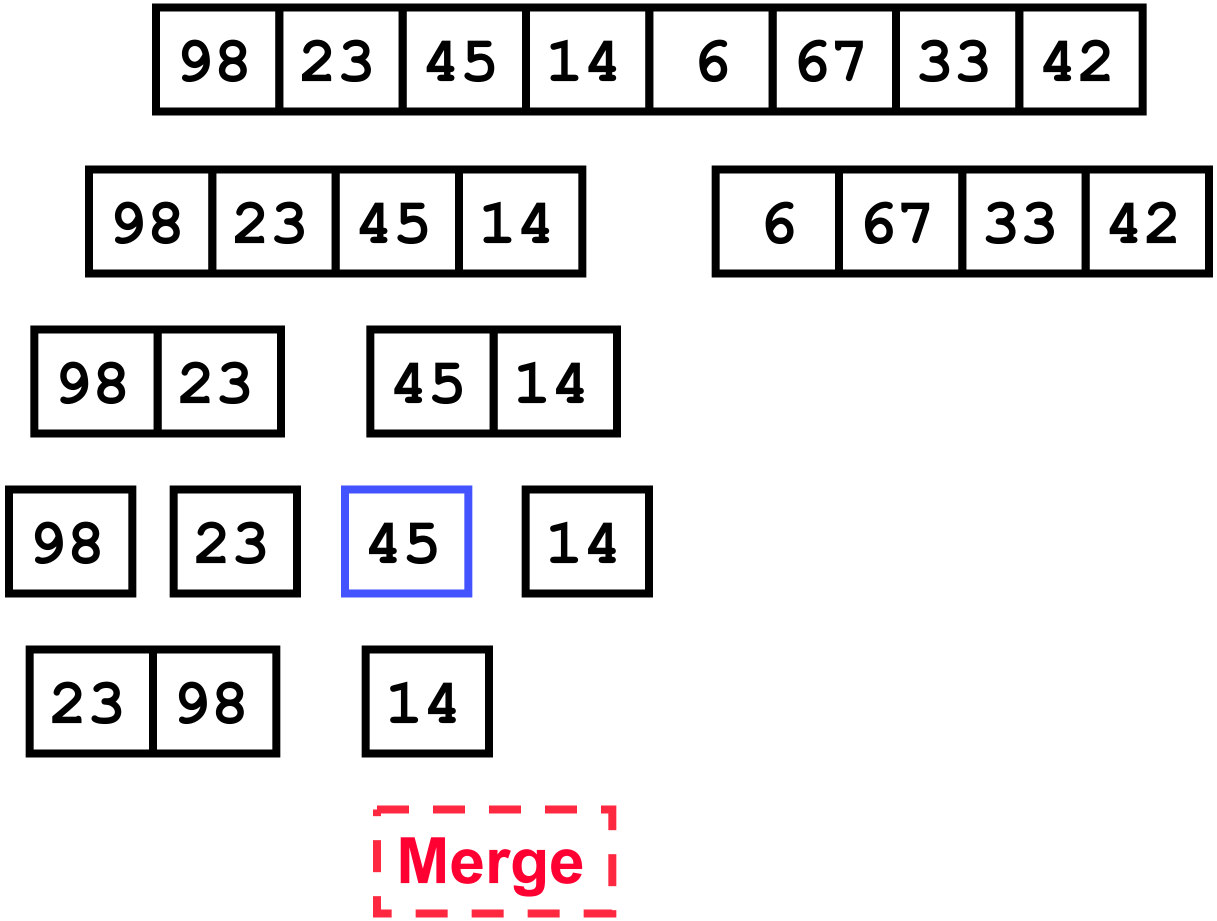

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

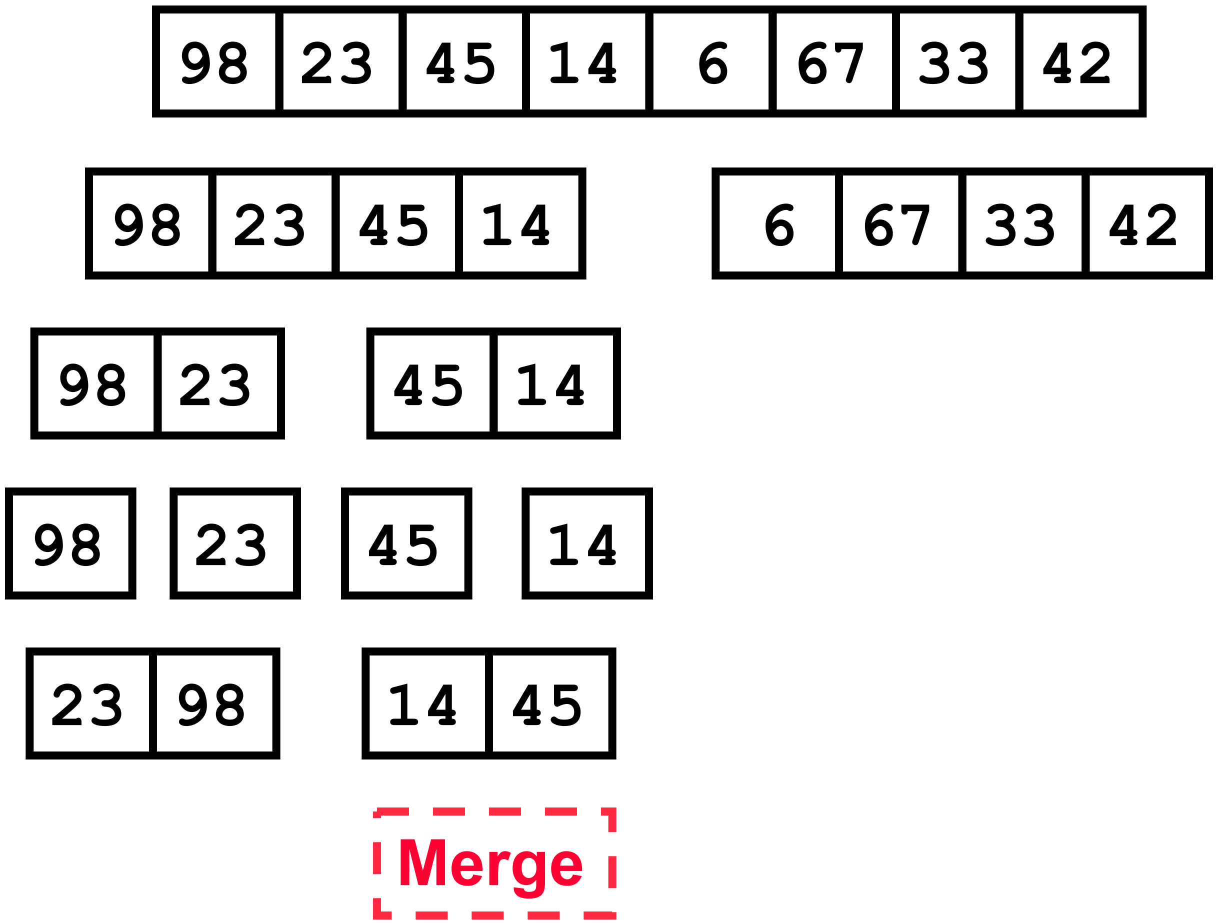

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

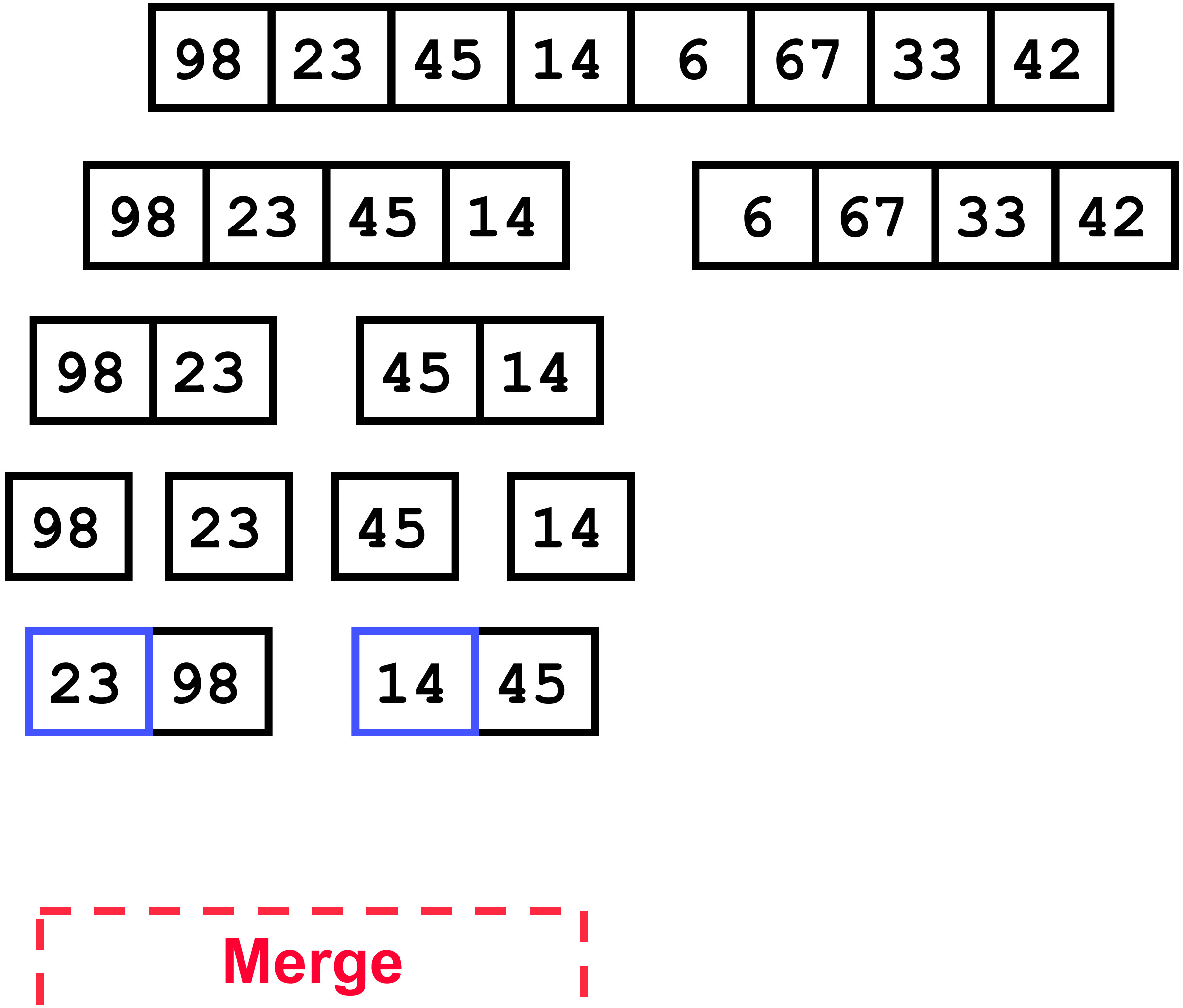

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge

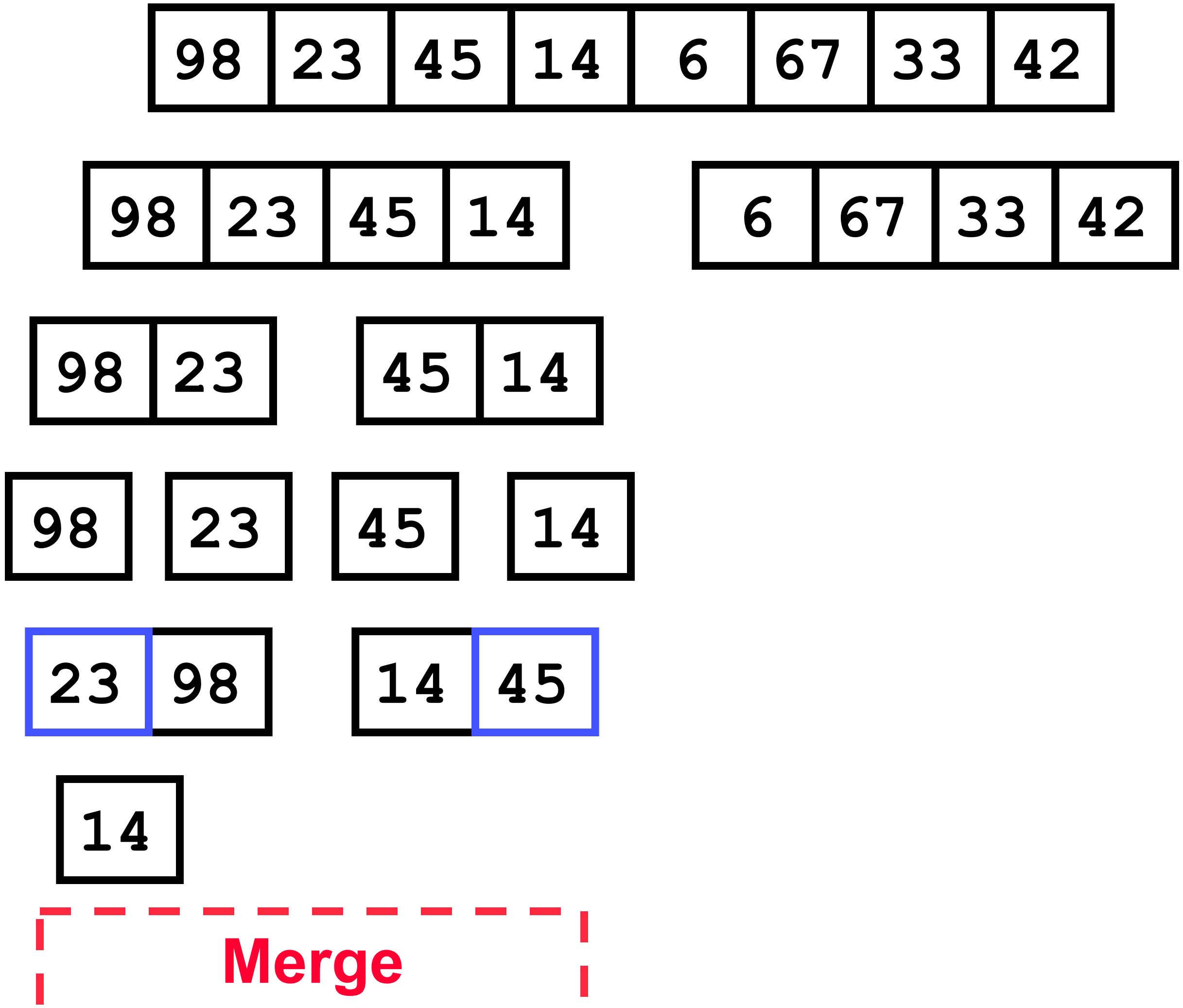

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98

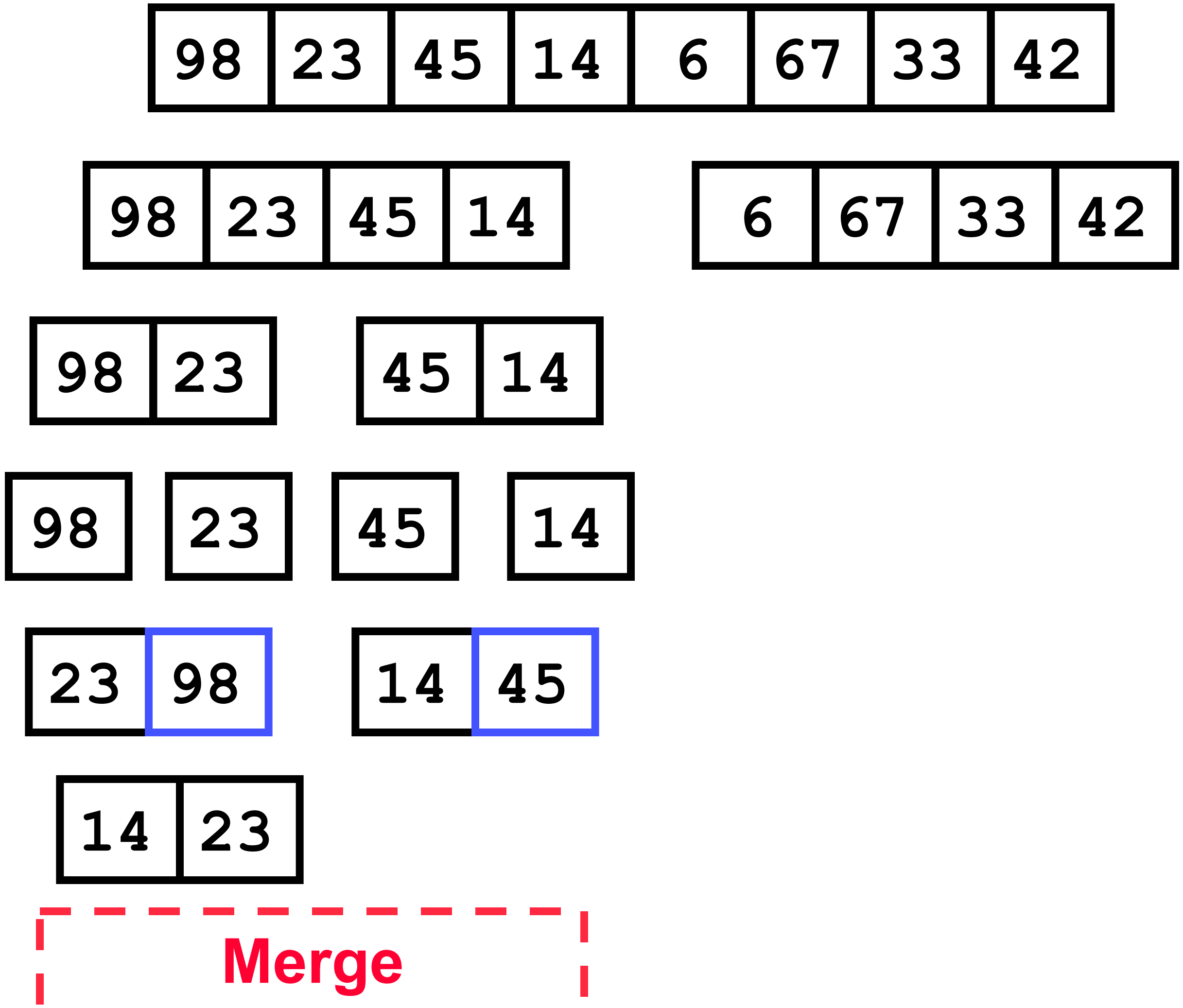

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
Merge

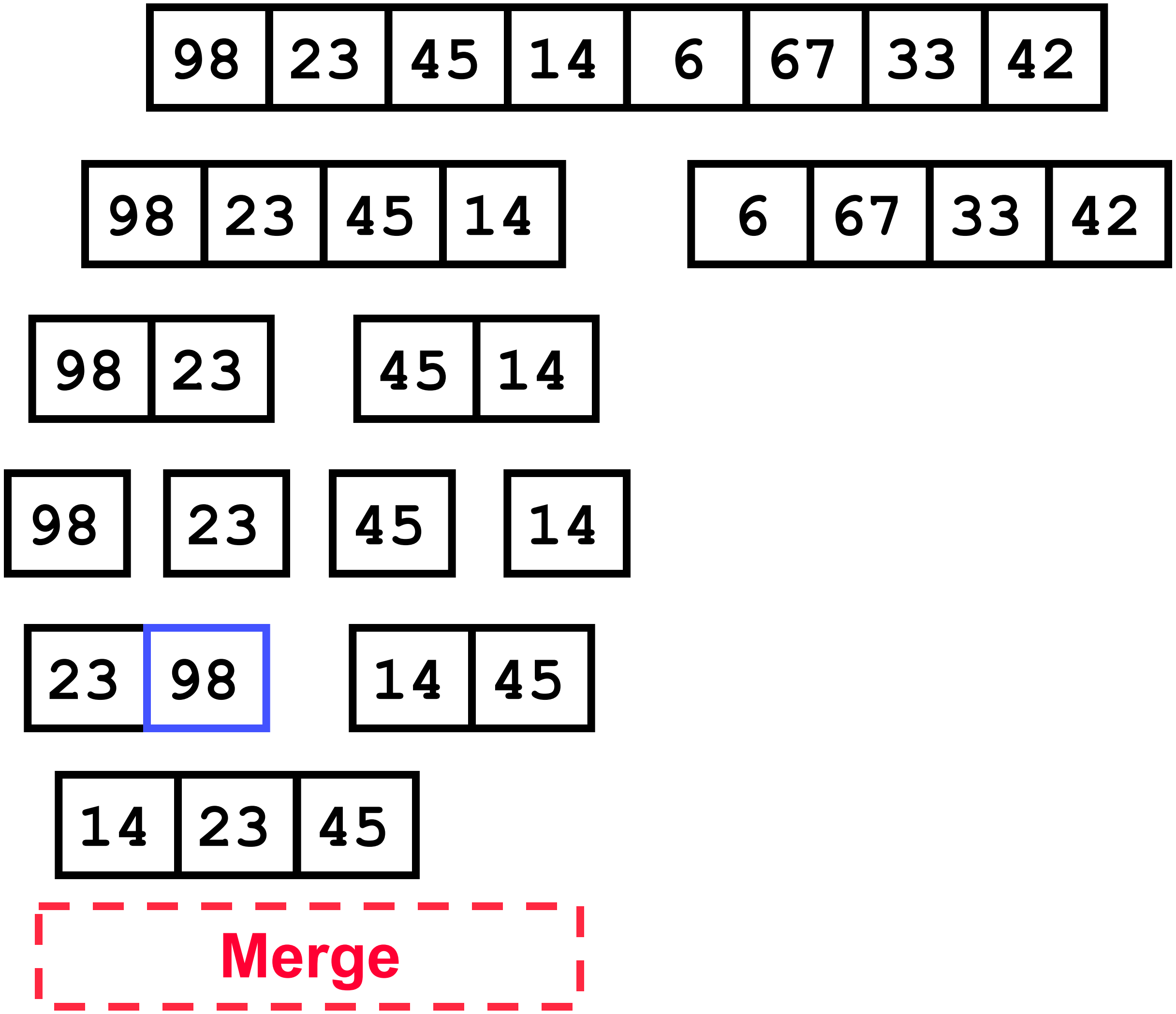

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14
Merge

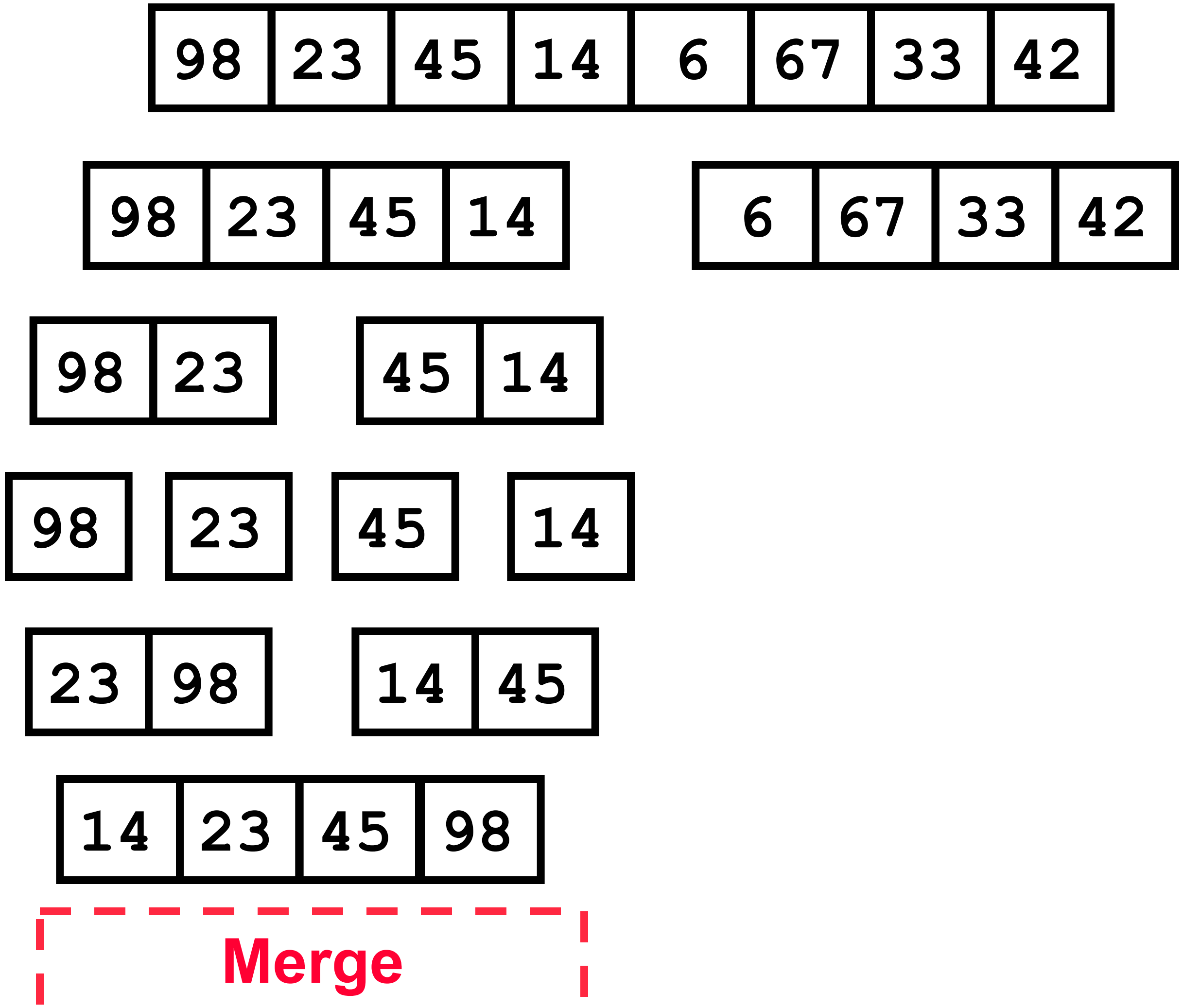

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

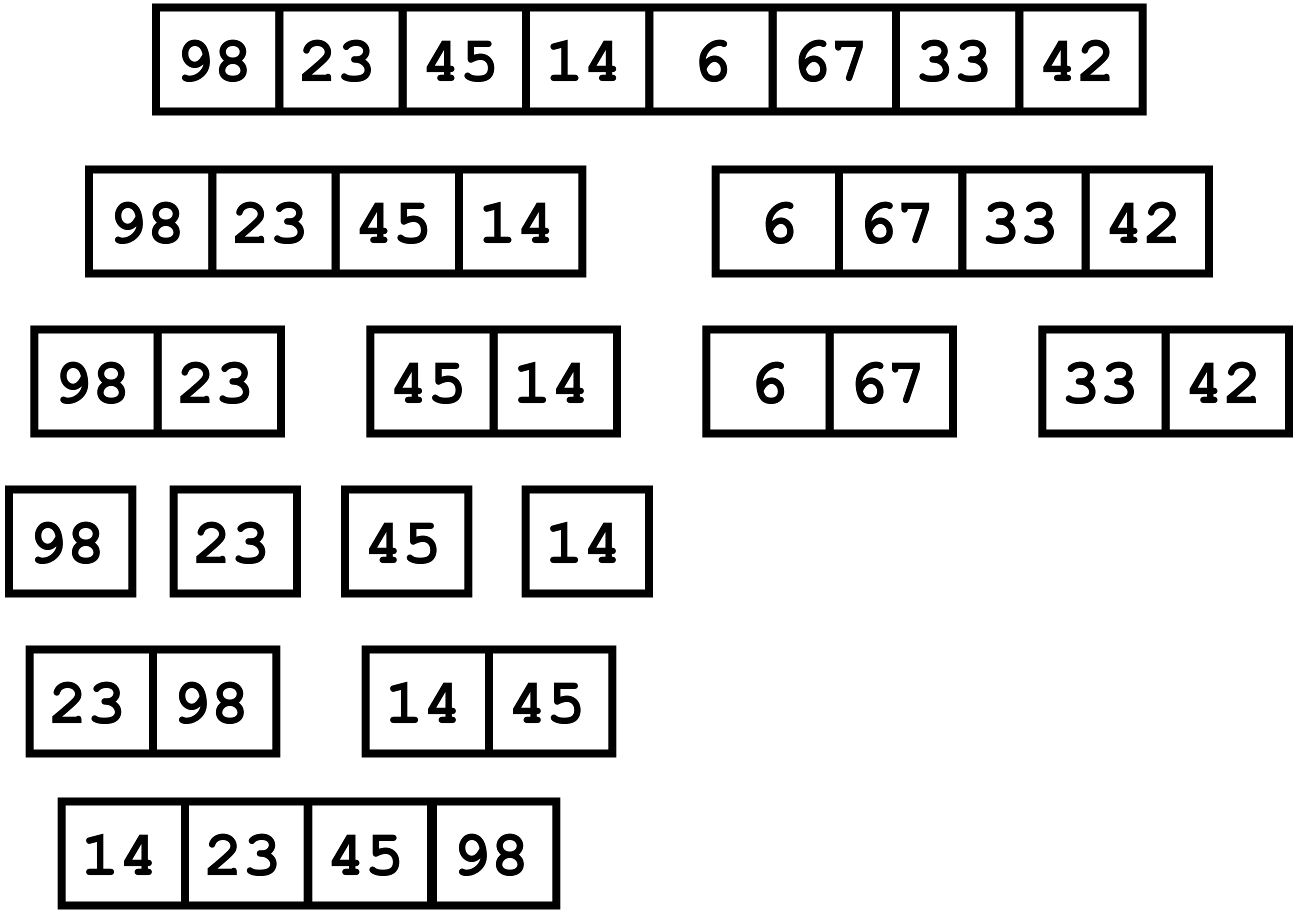

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

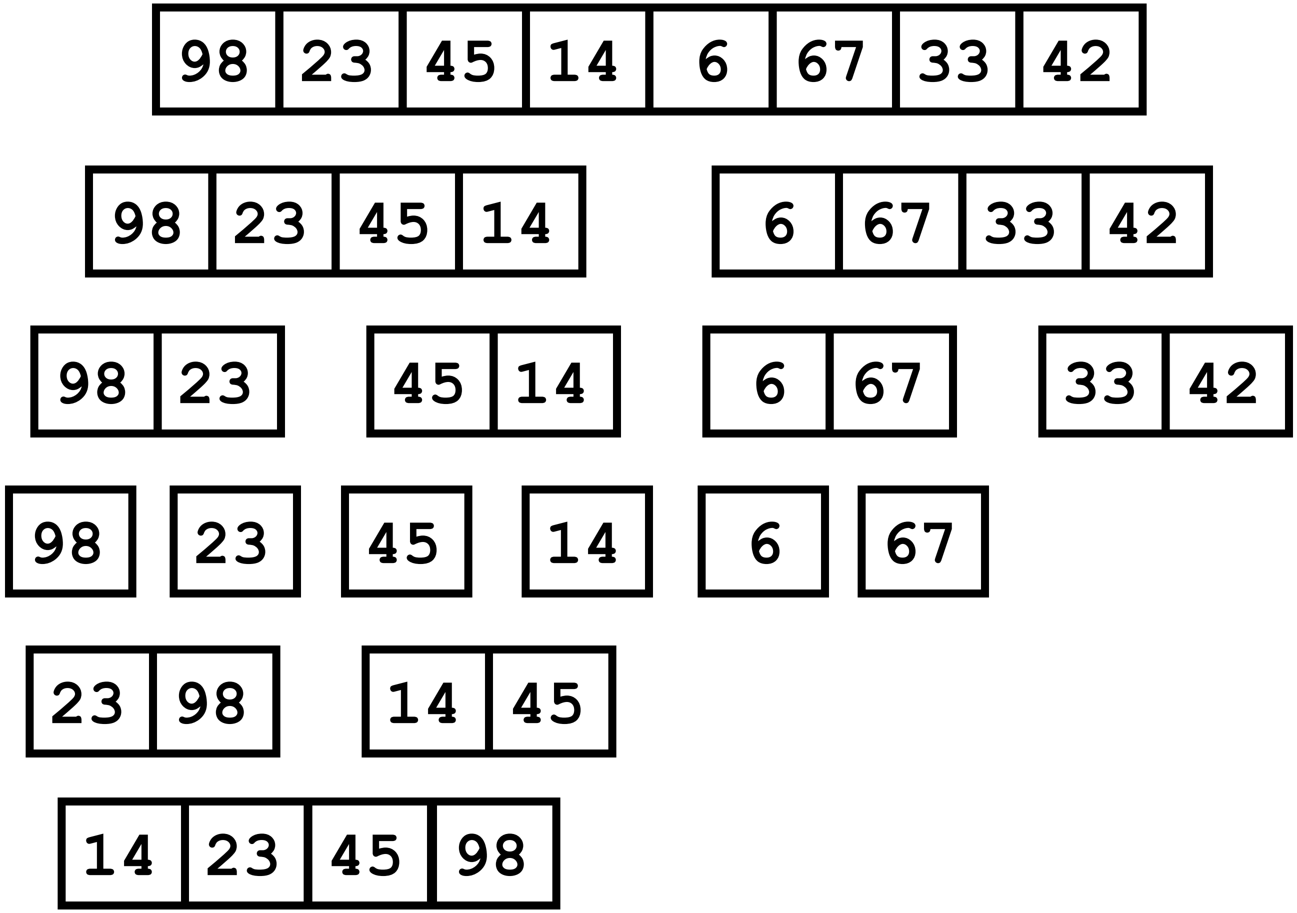

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

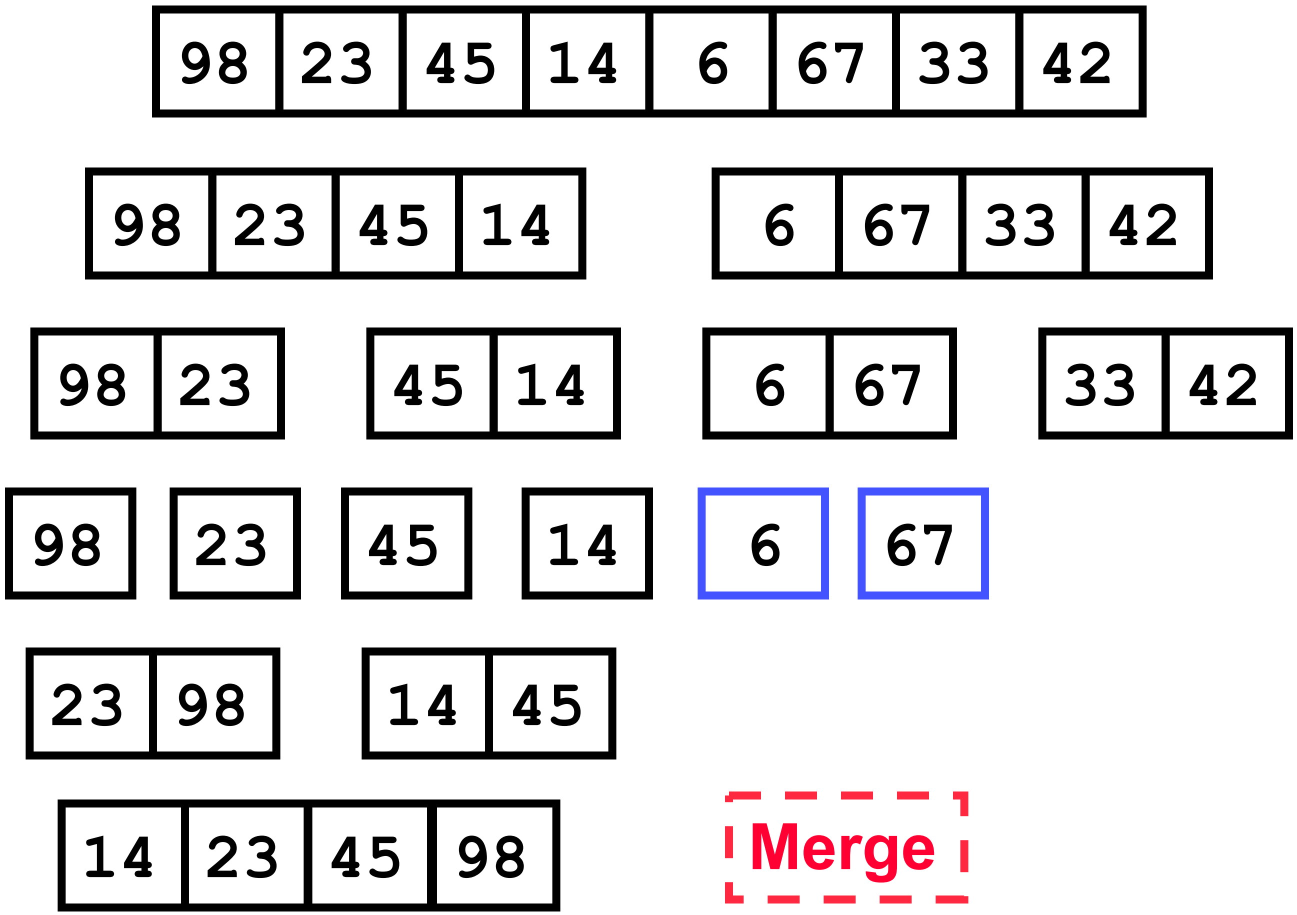

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

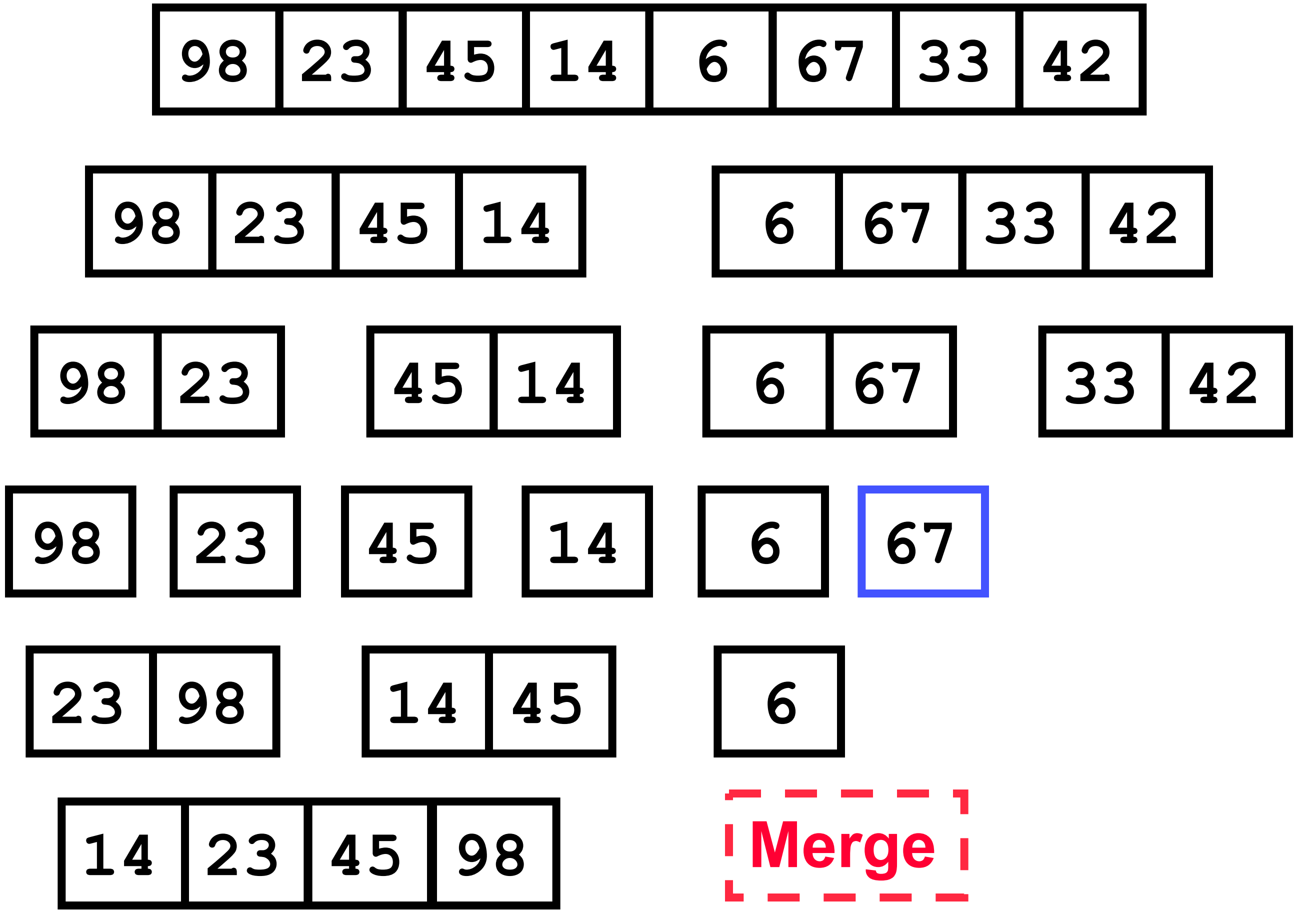

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

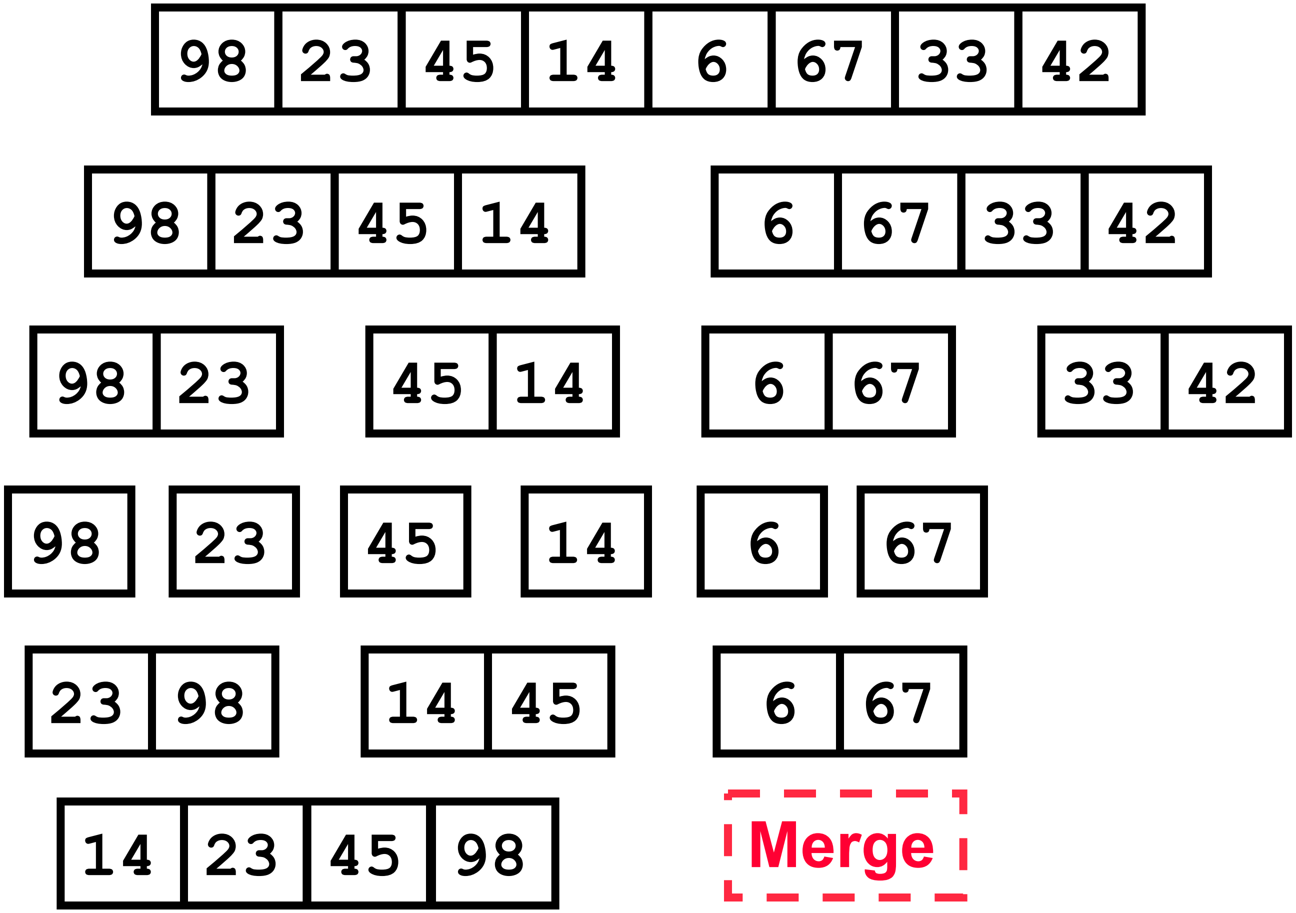

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

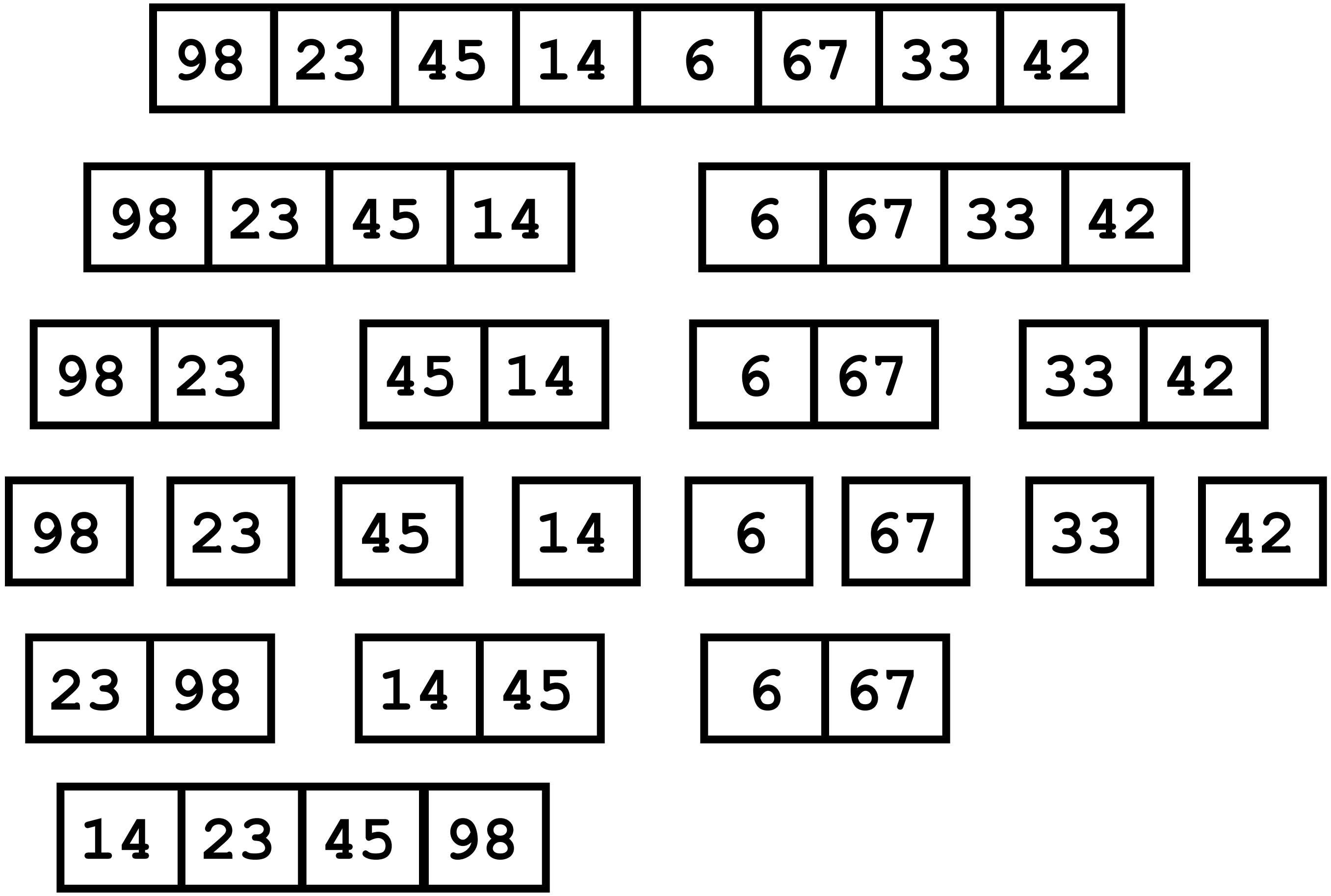

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

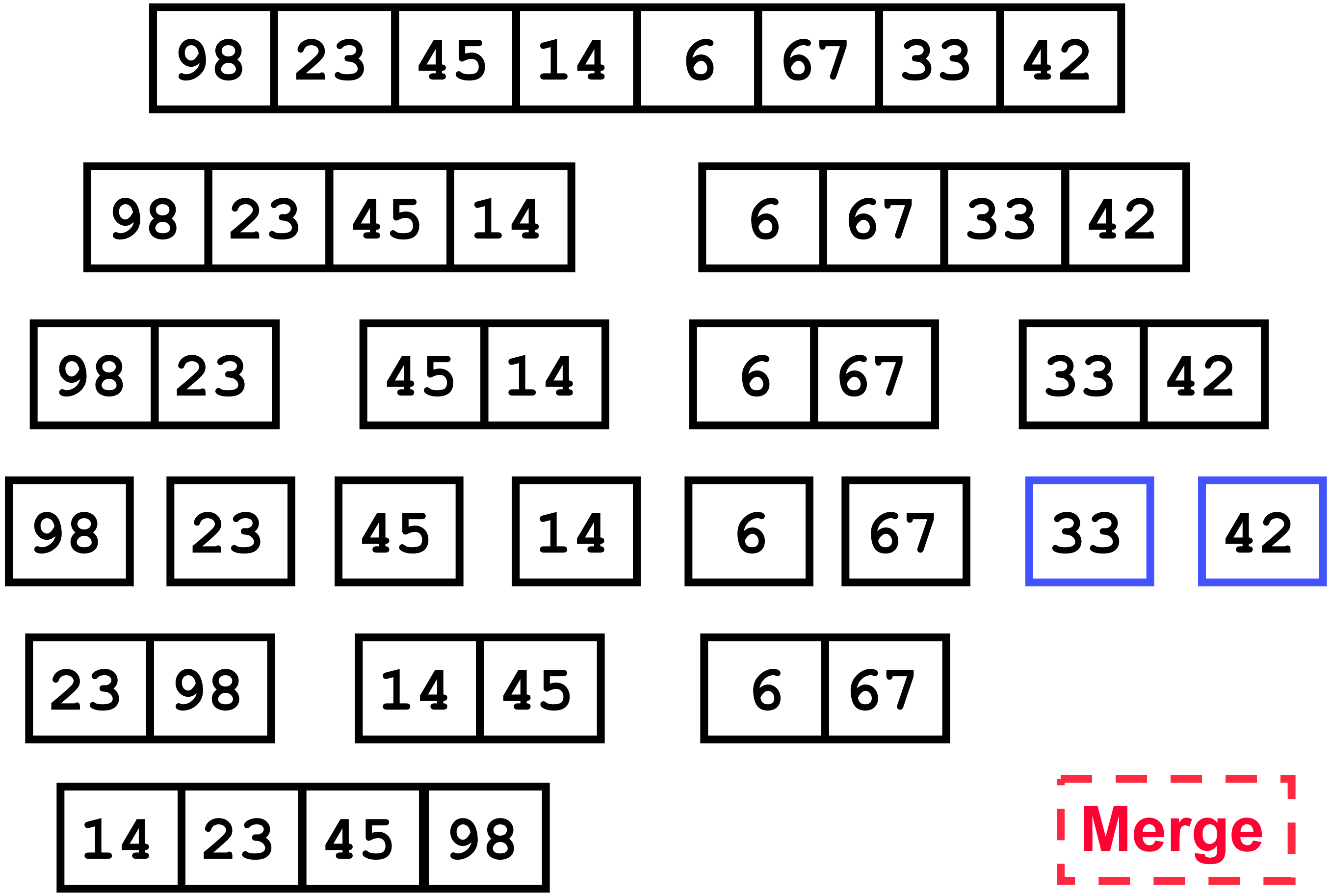

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

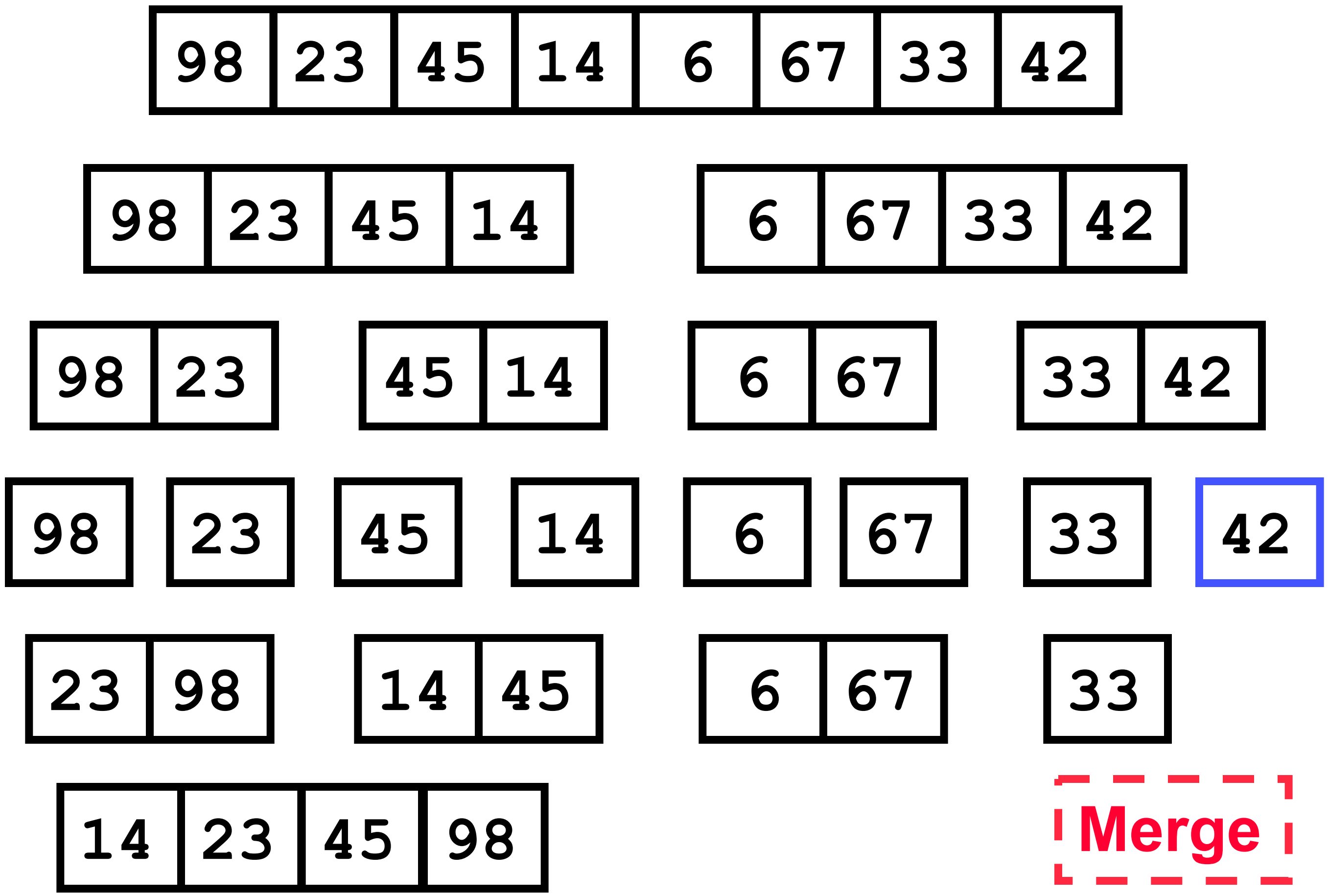

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

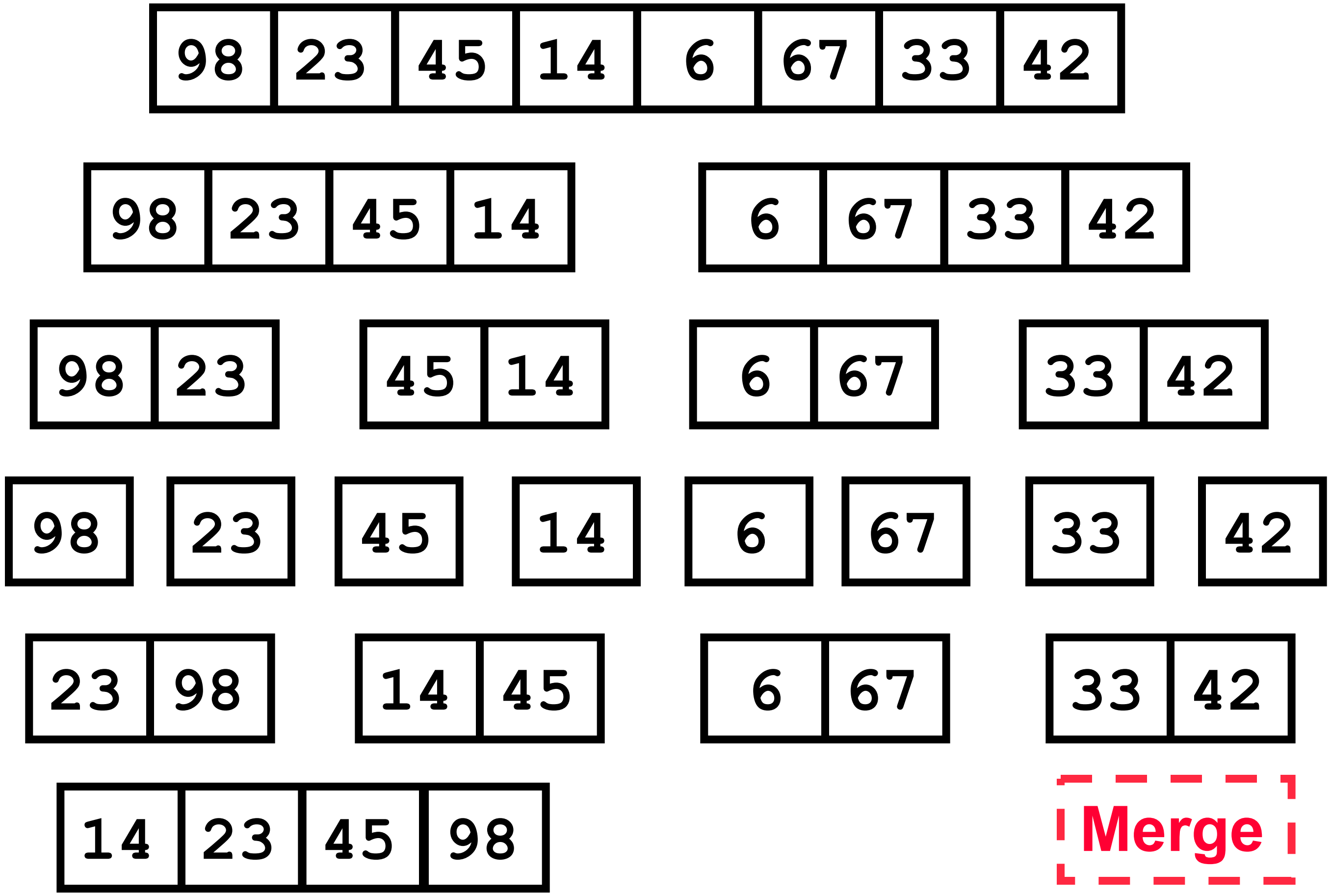

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge

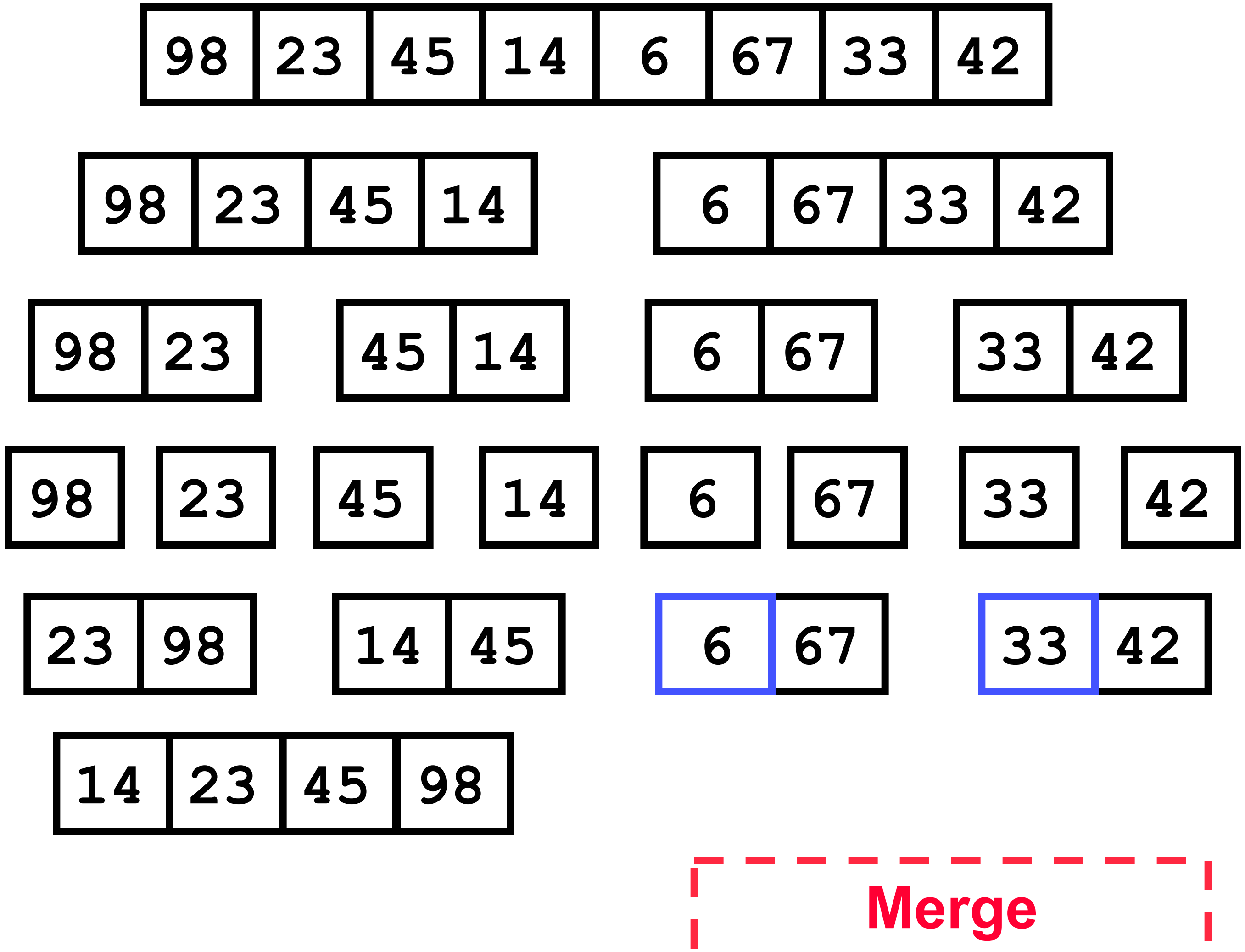

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6 67
14 23 45 98
Merge

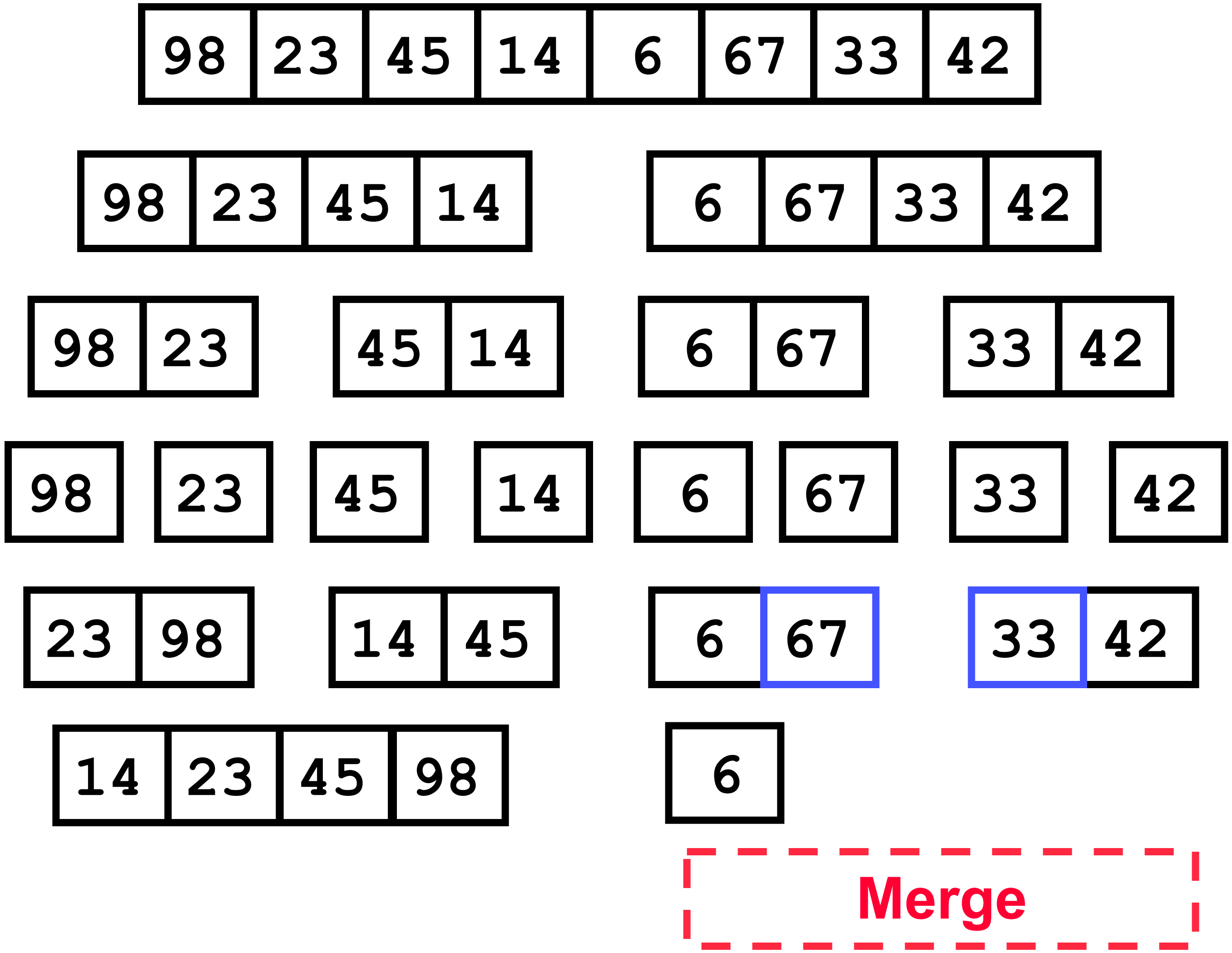

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98

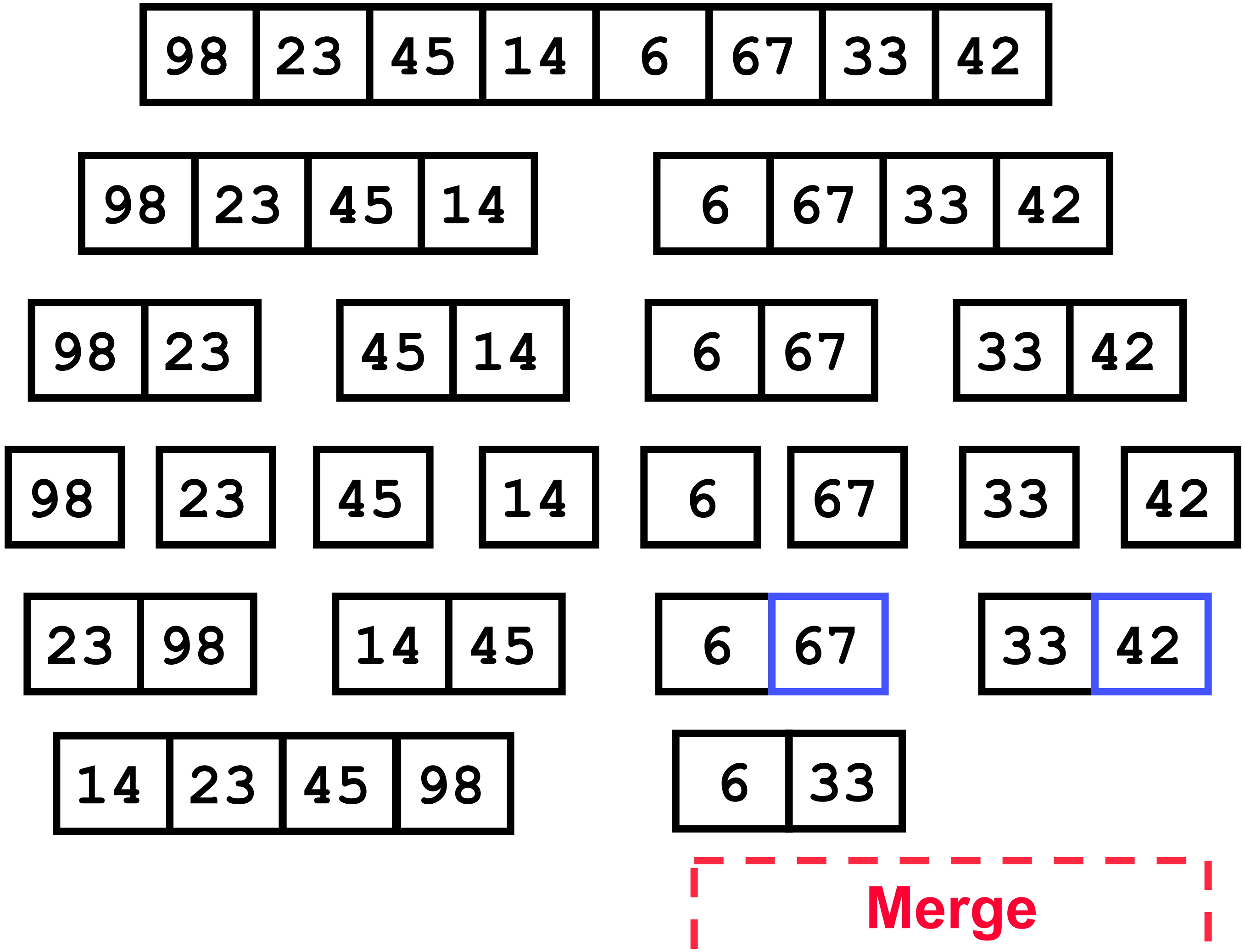

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98
Merge

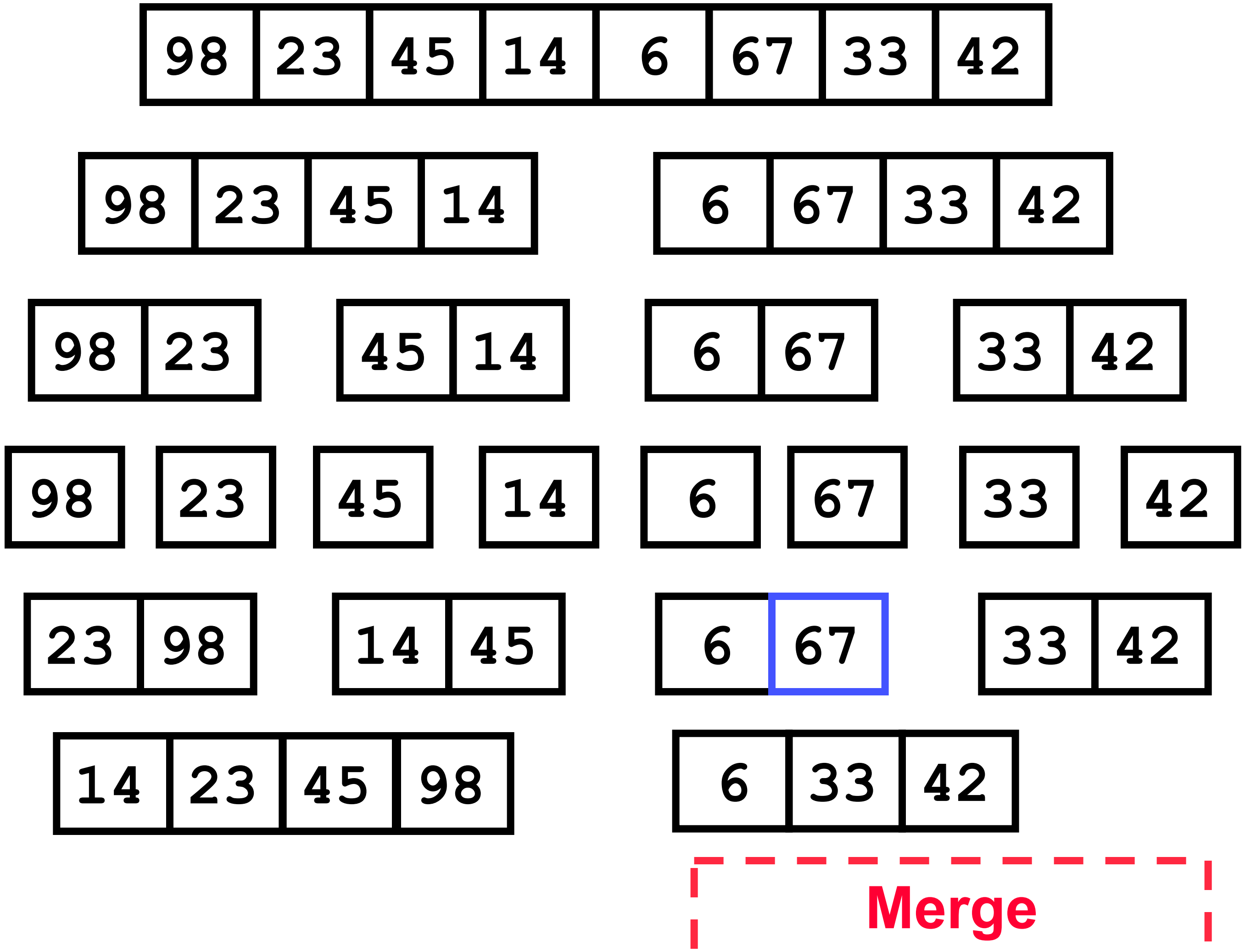

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge

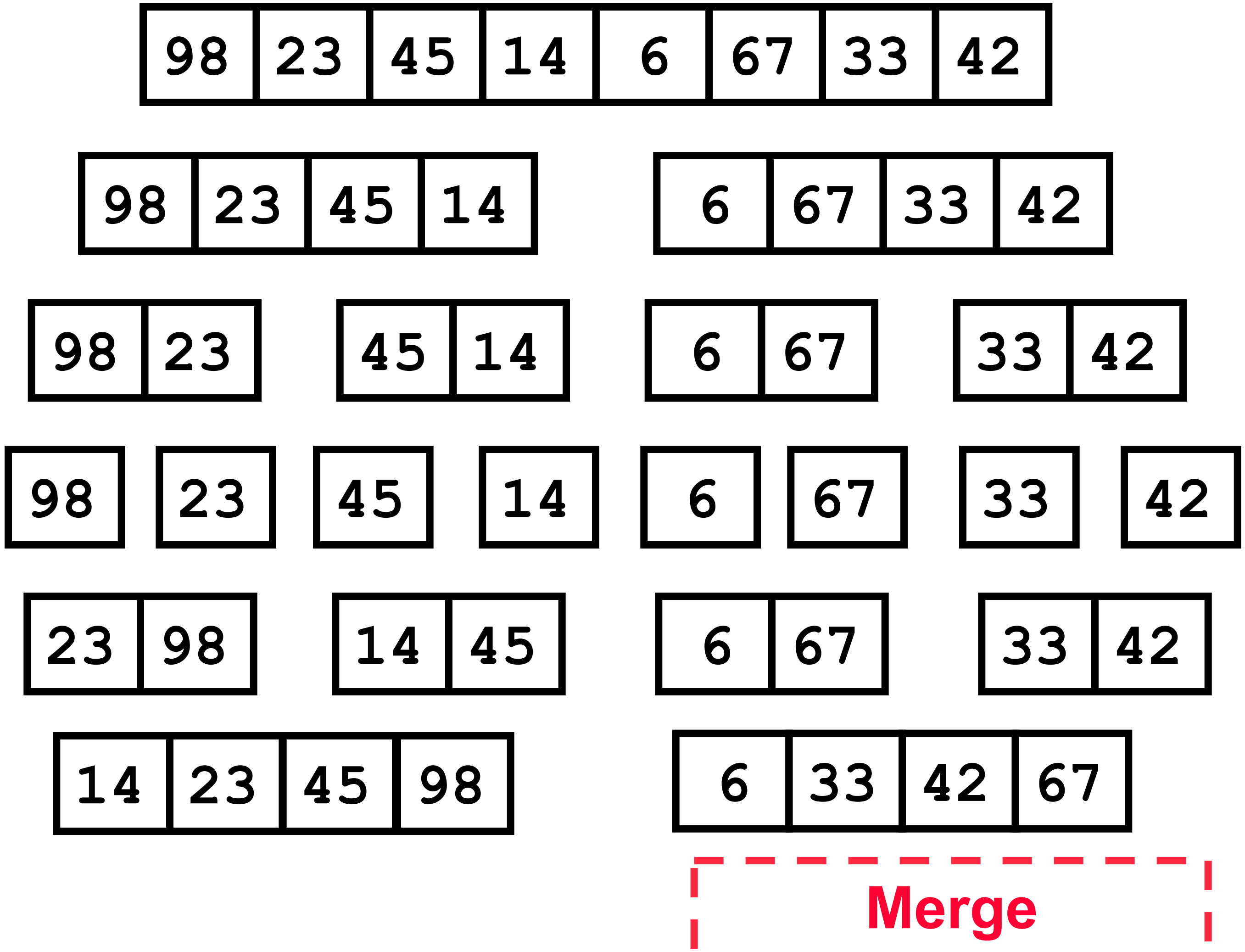

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

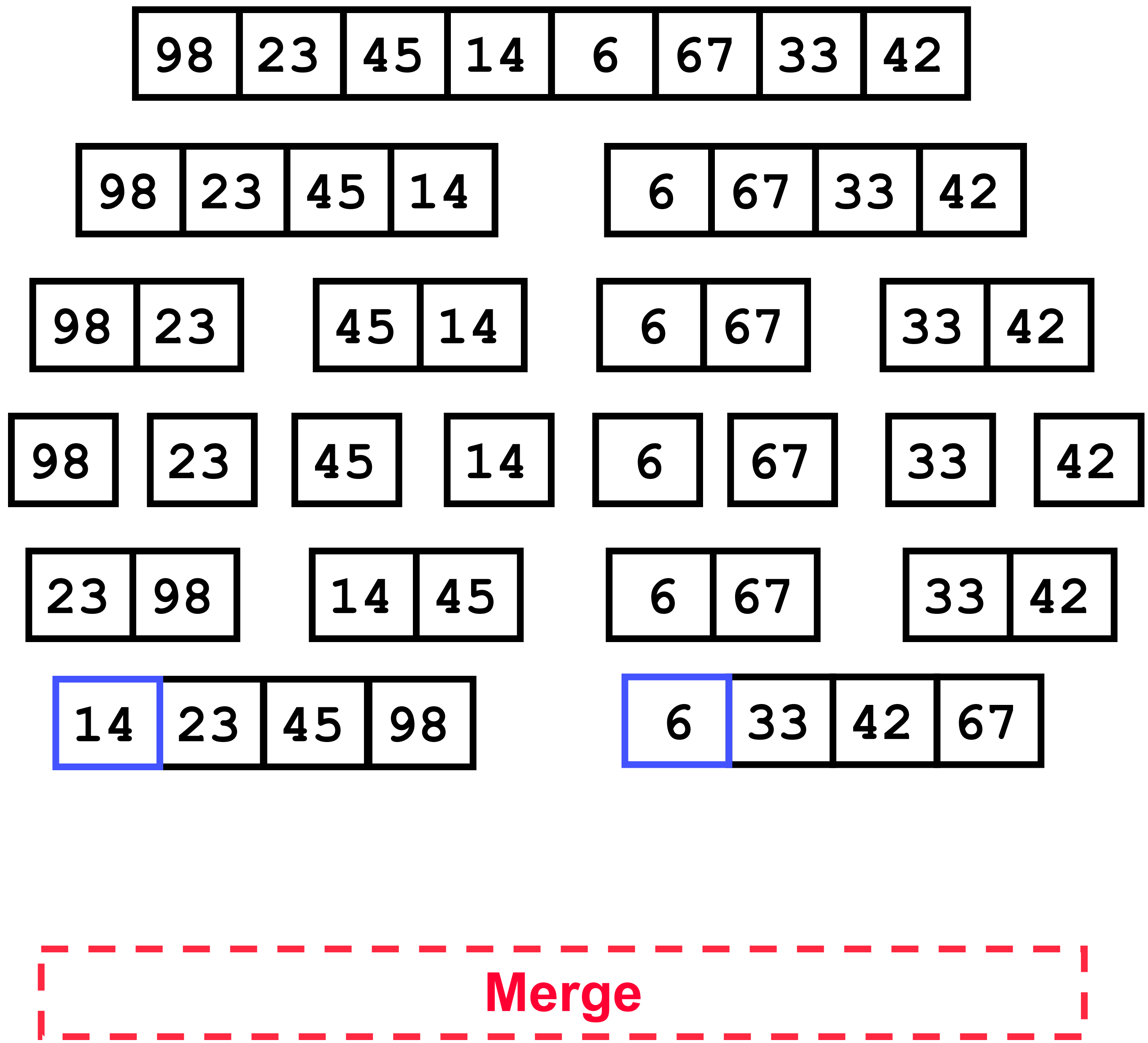

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

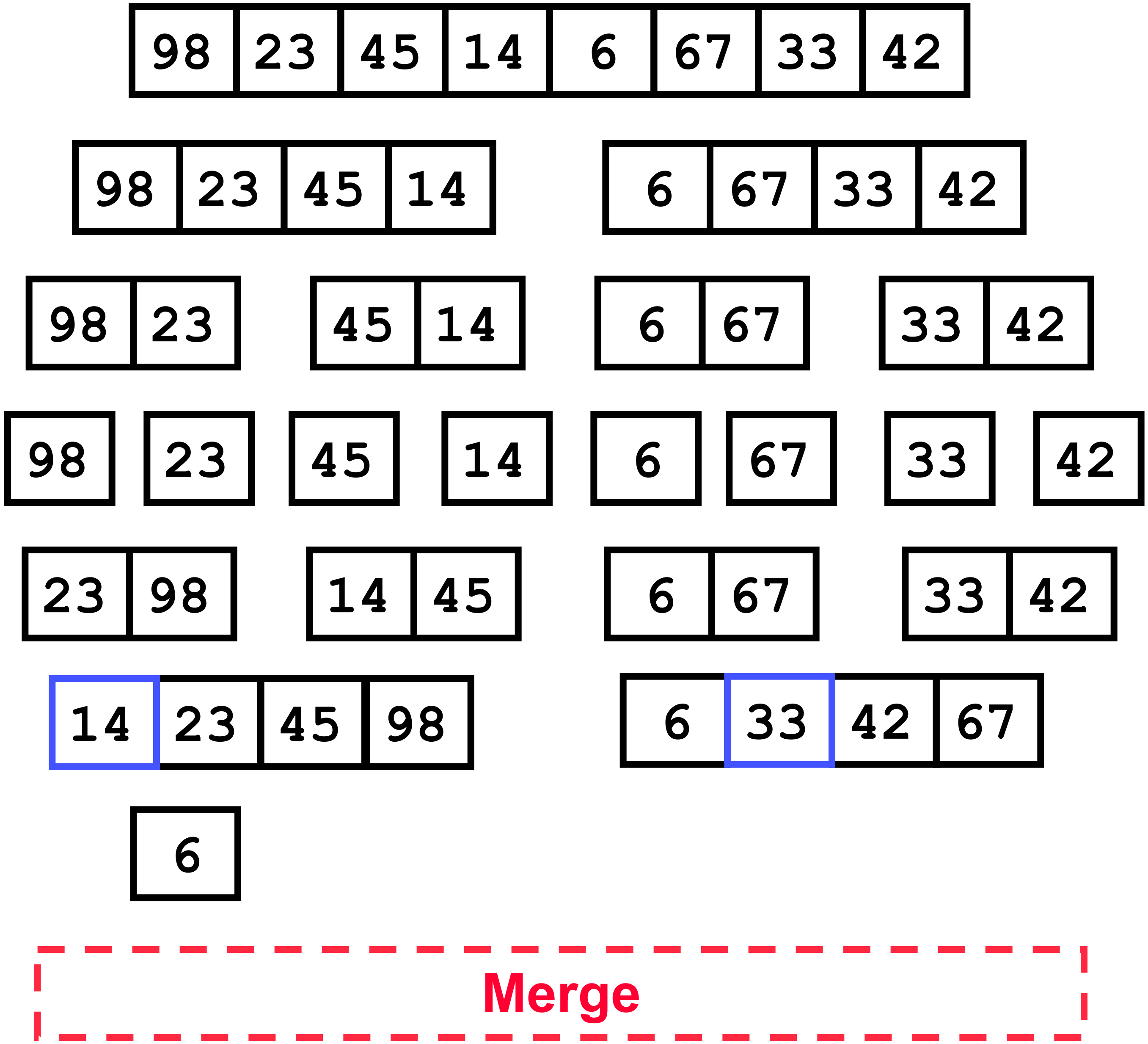

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6
Merge

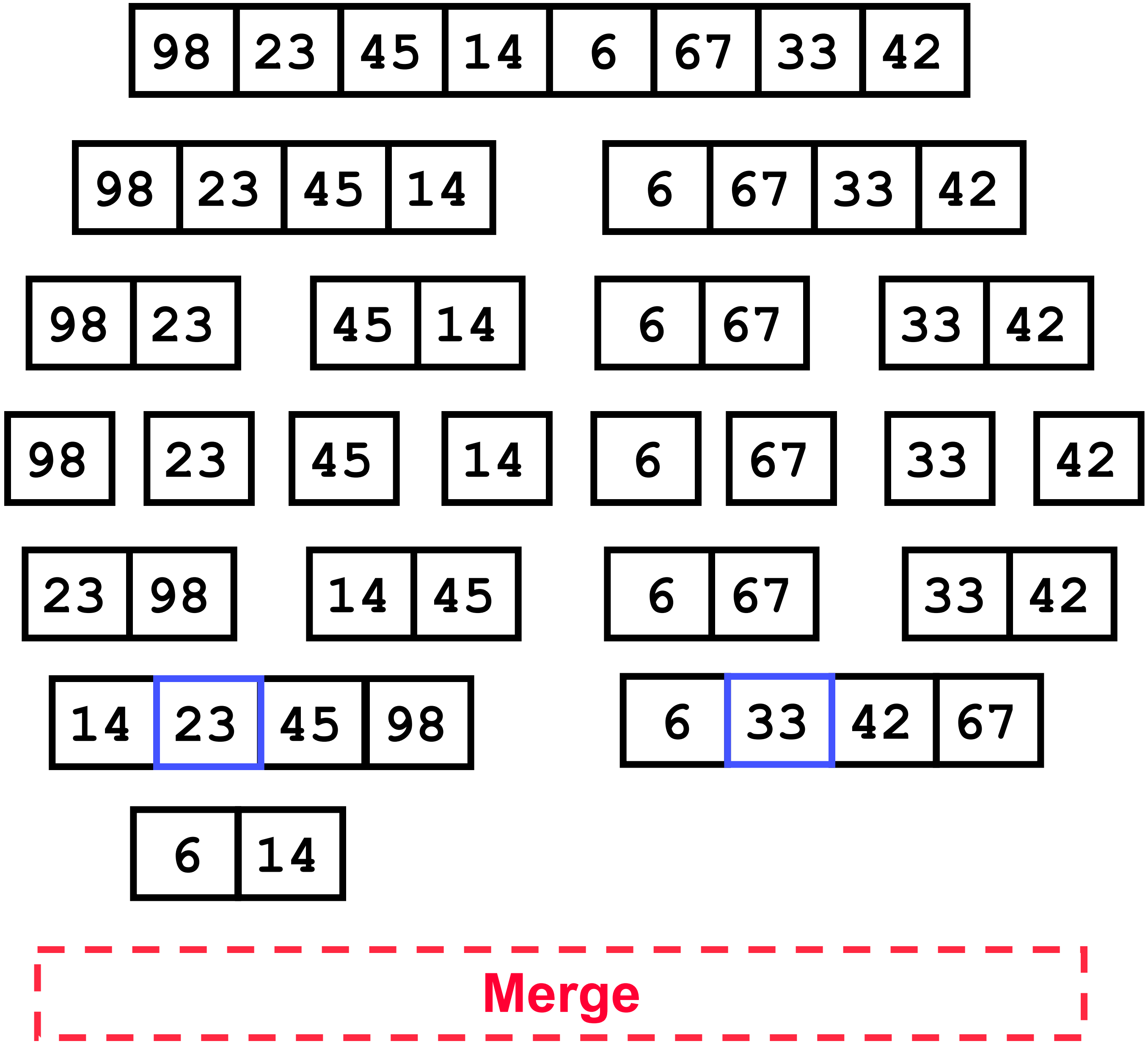

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33
Merge

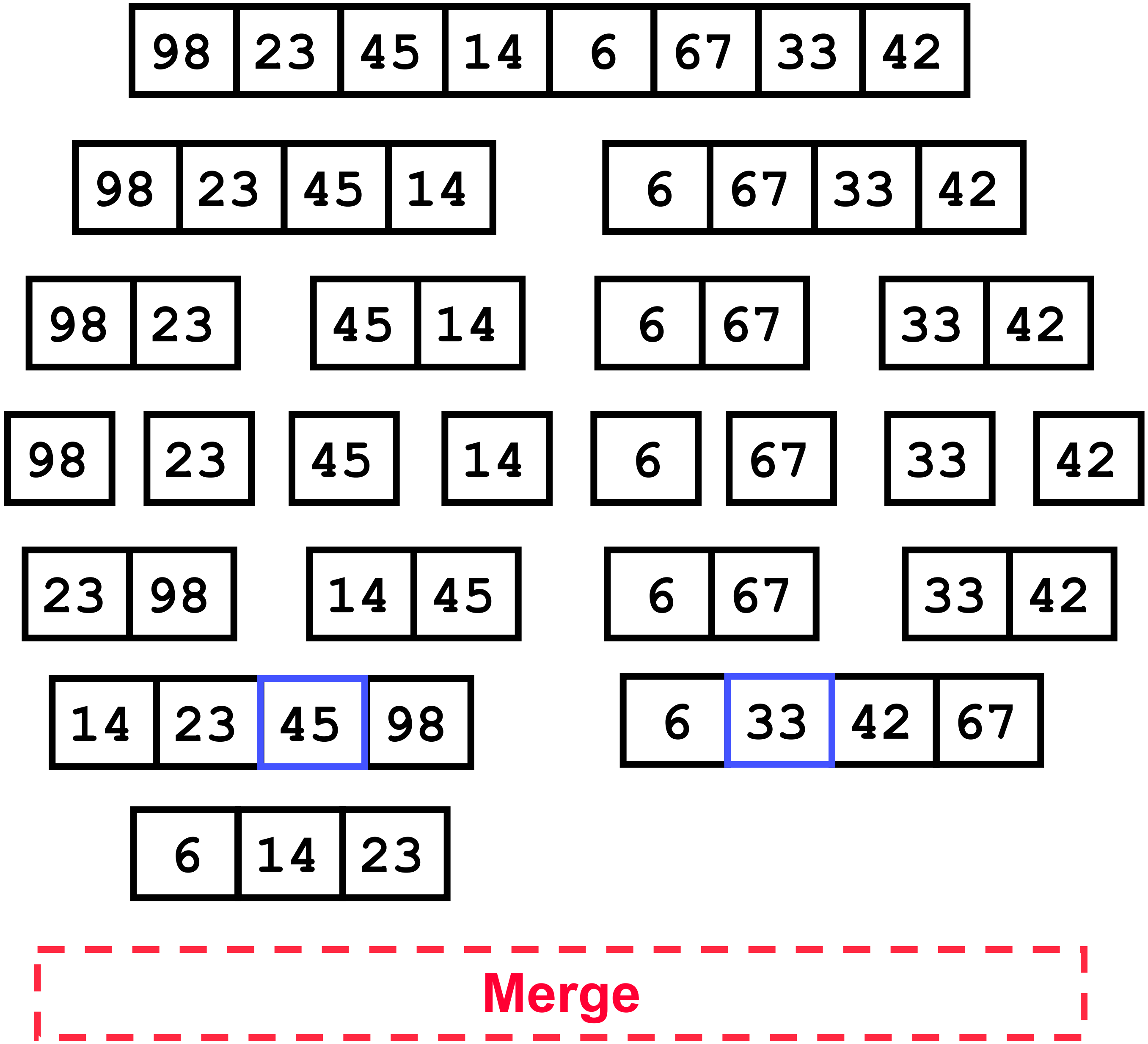

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42
Merge

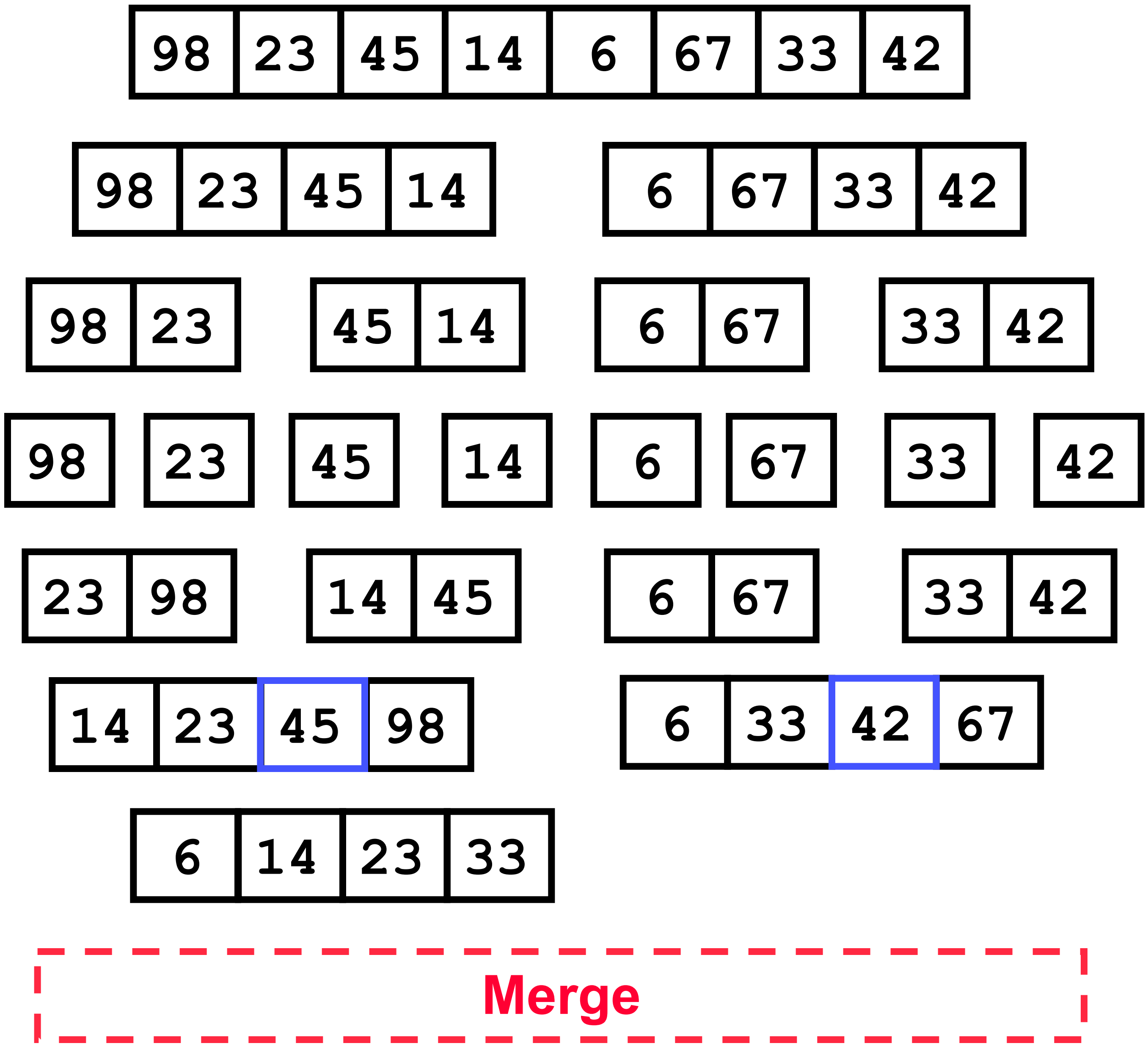

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

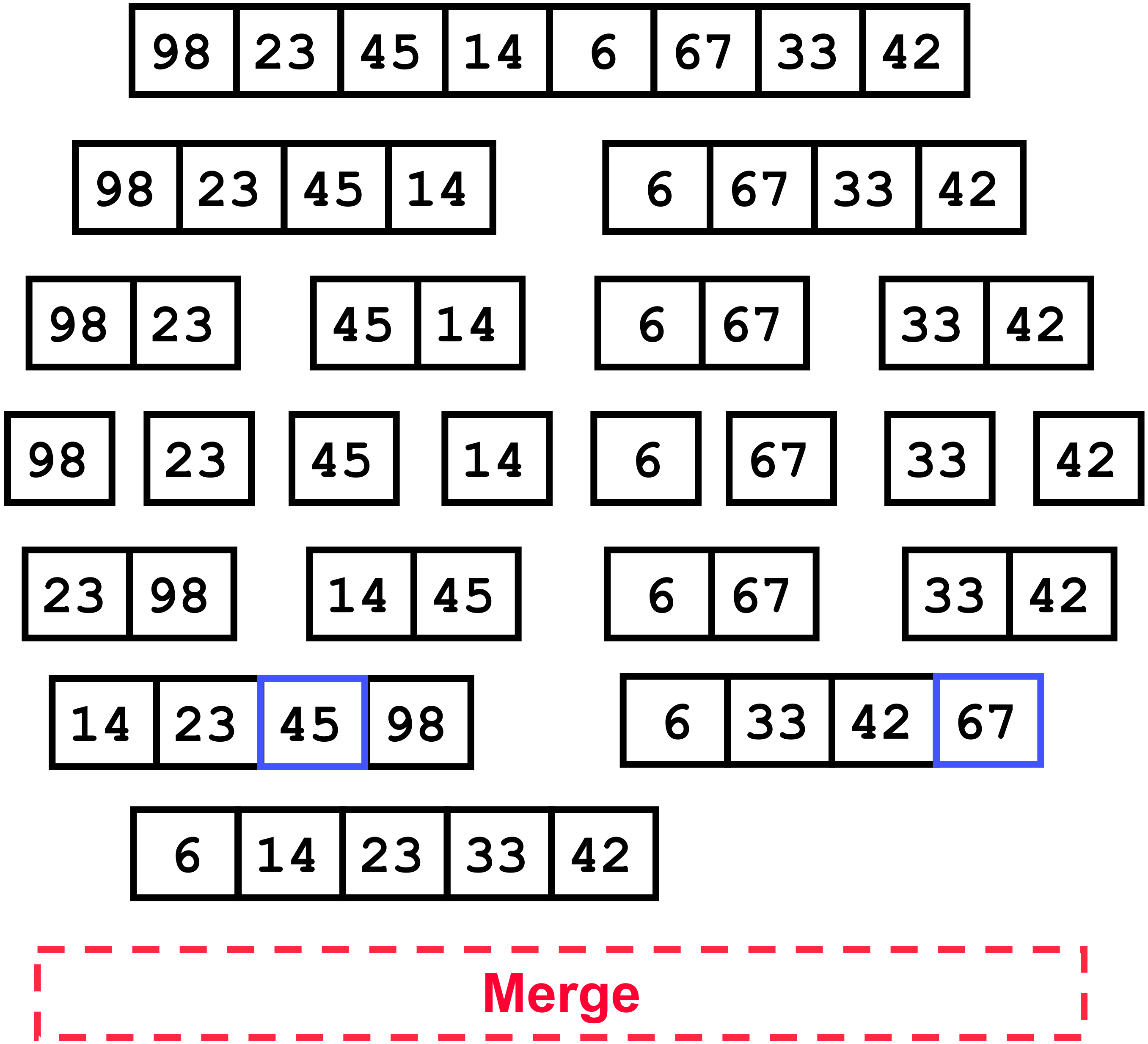

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

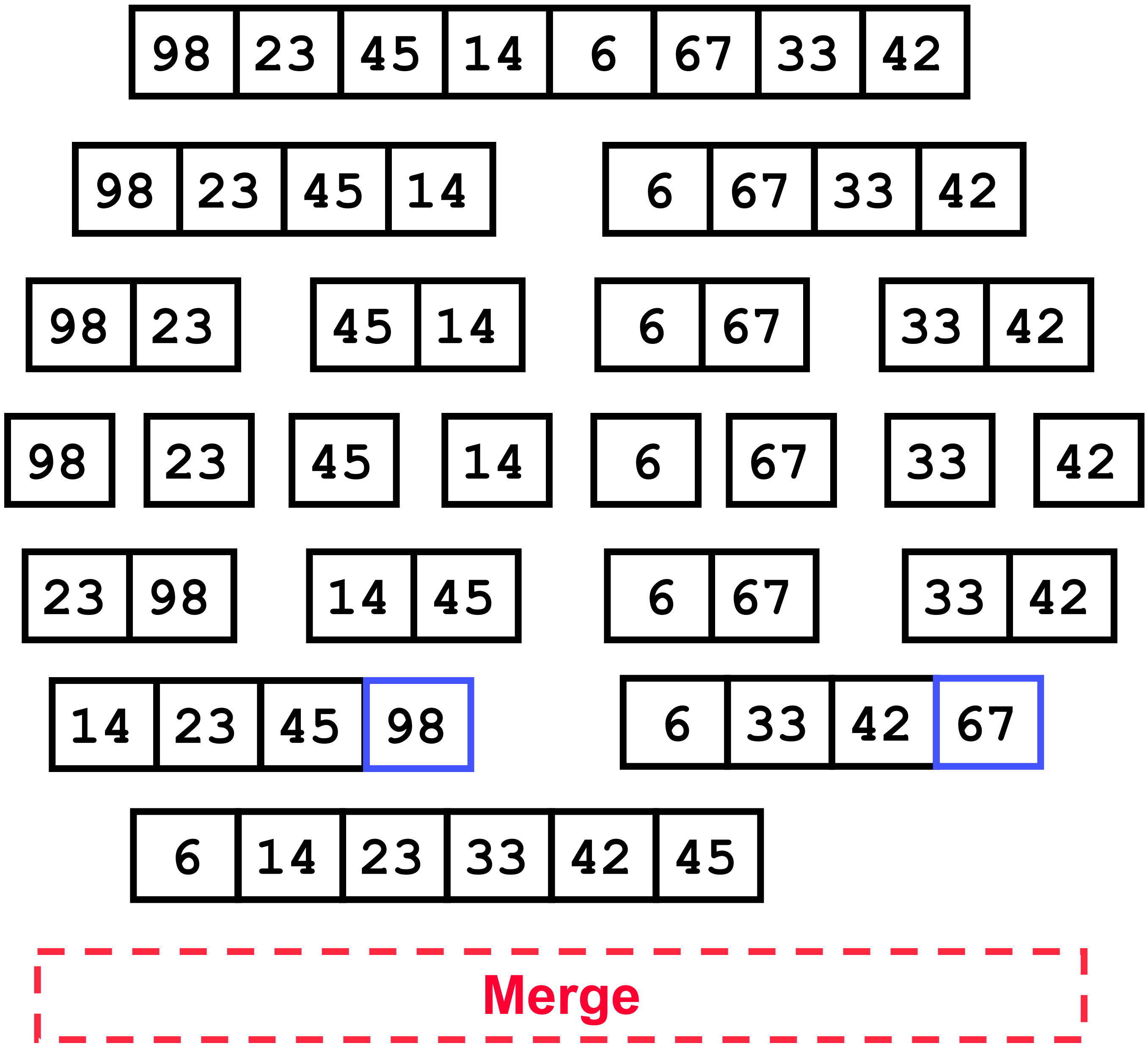

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6
Merge

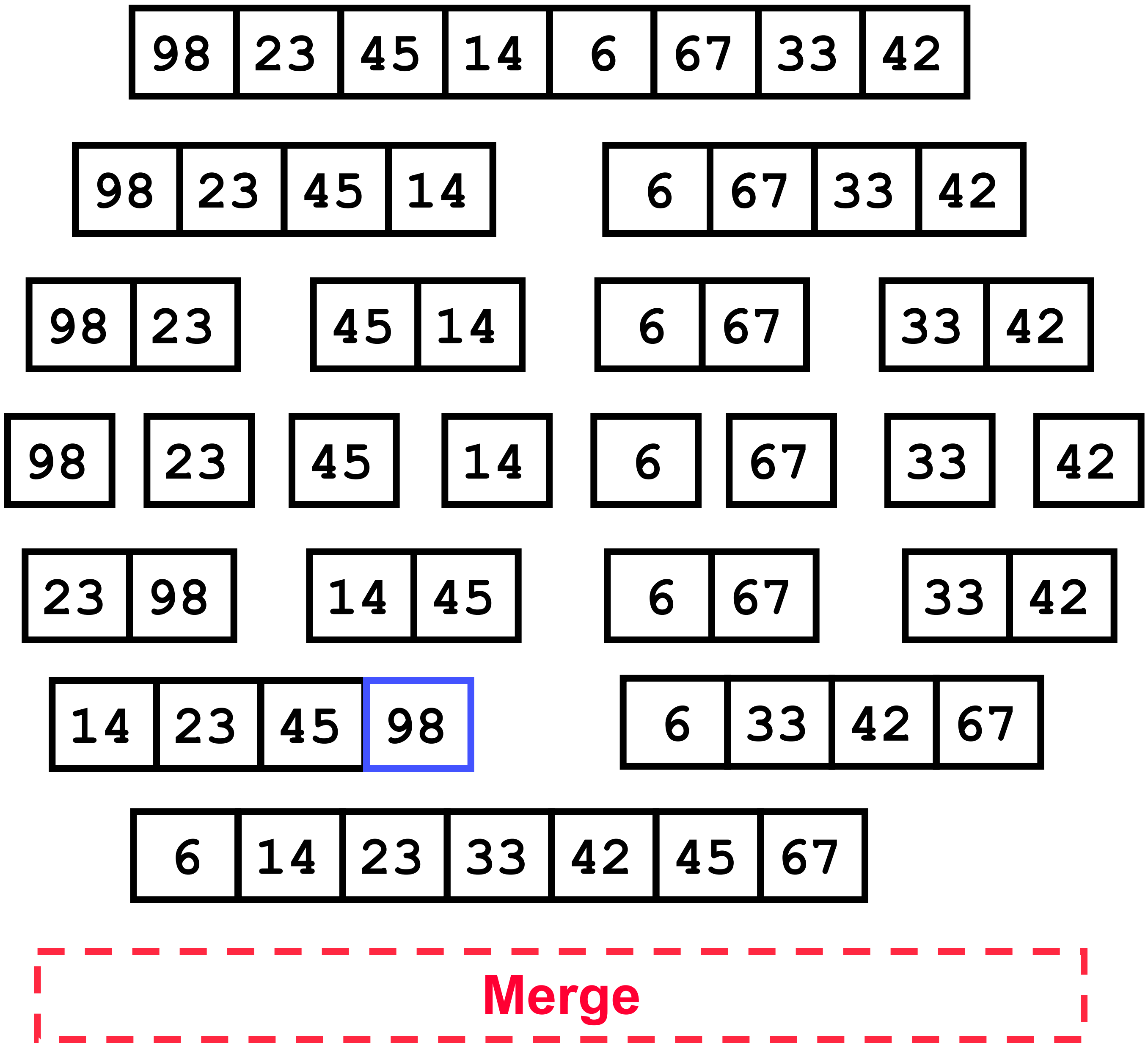

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14
Merge

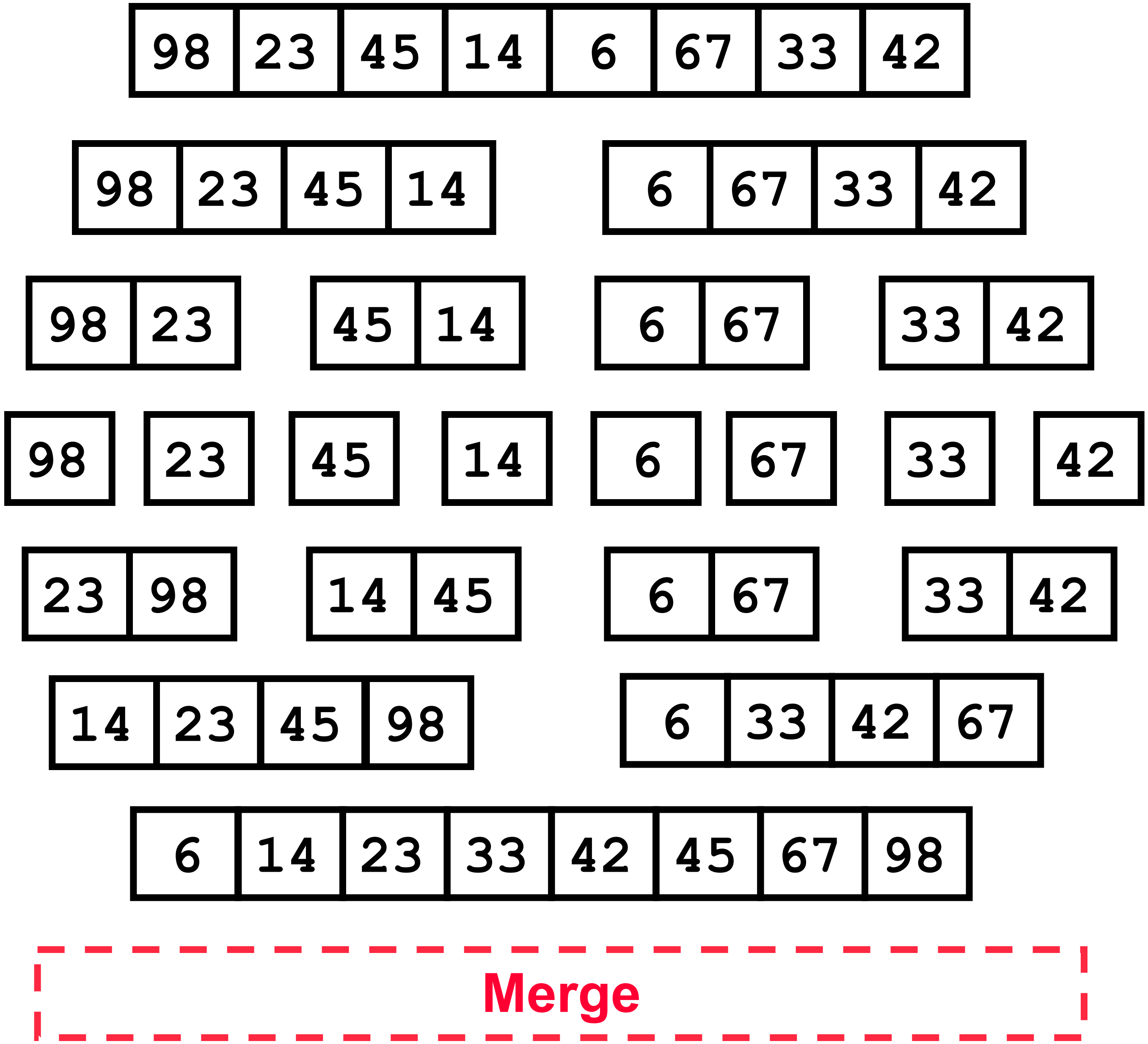

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23
Merge

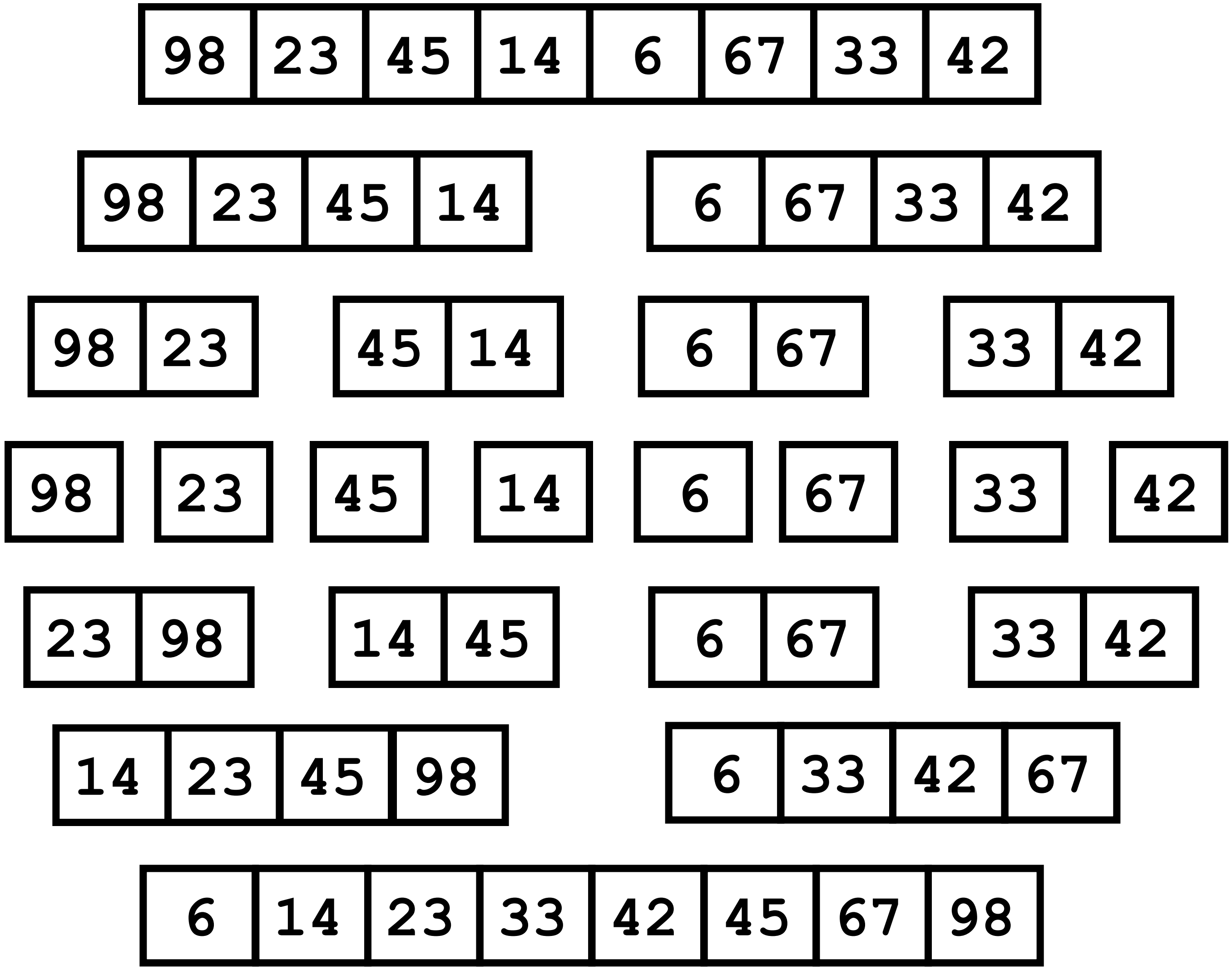

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67
Merge

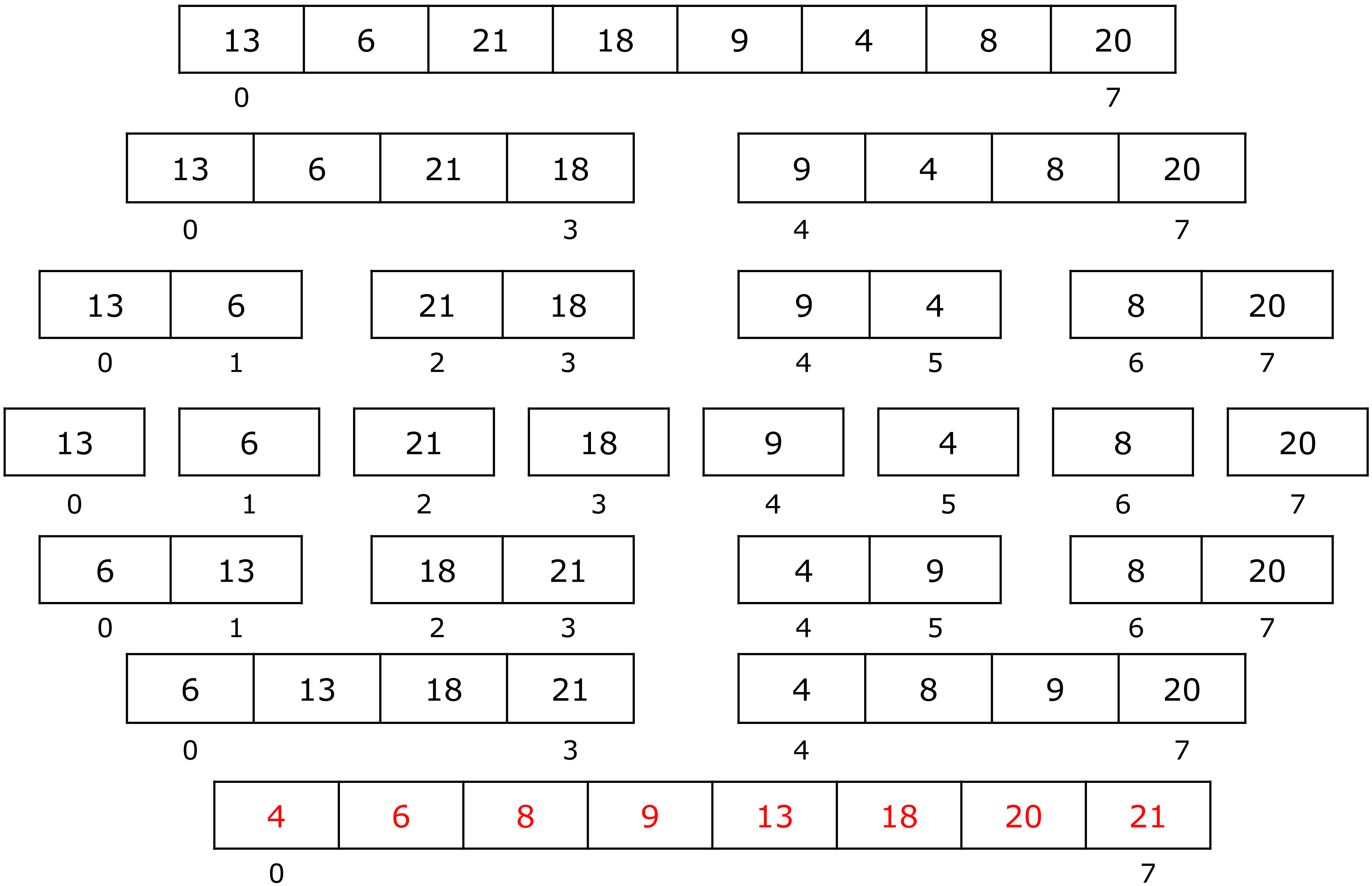

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67 98
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67 98

98	23	45	14	6	67	33	42

6	14	23	33	42	45	67	98

Merge sort example 2



Merging two sorted arrays

- *merge* operation:
  - Given two sorted arrays, *merge* operation produces a sorted array with all the elements of the two arrays

A

6	13	18	21

B

4	8	9	20

C

4	6	8	9	13	18	20	21

Running time of *merge*: O(*n*), where *n* is the number of elements in the merged array.

when merging two sorted parts of the same array, we'll need a *temporary array* to store the merged whole

Merge sort code

```
public static void mergeSort(int[] a) {
    int[] temp = new int[a.length];
    mergeSort(a, temp, 0, a.length - 1);
}

private static void mergeSort(int[] a, int[] temp,
                              int left, int right) {
    if (left >= right) {  // base case
        return;
    }

    // sort the two halves
    int mid = (left + right) / 2;
    mergeSort(a, temp, left, mid);
    mergeSort(a, temp, mid + 1, right);

    // merge the sorted halves into a sorted whole
    merge(a, temp, left, right);
}
```

Merge code

```
private static void merge(int[] a, int[] temp,
                          int left, int right) {
    int mid = (left + right) / 2;
    int count = right - left + 1;

    int l = left;                        // counter indexes for L, R
    int r = mid + 1;

    // main loop to copy the halves into the temp array
    for (int i = 0; i < count; i++)
        if (r > right) {                 // finished right; use left
            temp[i] = a[l++];
        } else if (l > mid) {            // finished left; use right
            temp[i] = a[r++];
        } else if (a[l] < a[r]) {        // left is smaller (better)
            temp[i] = a[l++];
        } else {                         // right is smaller (better)
            temp[i] = a[r++];
        }

    // copy sorted temp array back into main array
    for (int i = 0; i < count; i++) {
        a[left + i] = temp[i];
    }
}
```

Merge sort runtime

- let T(n) be runtime of merge sort on n items
  - T(0) = 1
  - T(1) = 2*T(0) + 1
  - T(2) = 2*T(1) + 2
  - T(4) = 2*T(2) + 4
  - T(8) = 2*T(4) + 8
  - ...
  - T(n/2) = 2*T(n/4) + n/2
  - T(n) = 2*T(n/2) + n

Merge sort runtime

- $T(n) = 2*T(n/2) + n$
- $T(n/2) = 2*T(n/4) + n/2$

- $T(n) = 2*(2*T(n/4) + n/2) + n$
- $T(n) = 4*T(n/4) + 2n$
- $T(n) = 8*T(n/8) + 3n$
- ...
- $T(n) = 2^k\, T(n/2^k) + kn$

  To get to a more simplified case, let's set $k = \log_2 n$.

- $T(n) = 2^{\log n}\, T(n/2^{\log n}) + (\log n)\, n$
- $T(n) = n * T(n/n) + n \log n$
- $T(n) = n * T(1) + n \log n$
- $T(n) = n * 1 + n \log n$
- $T(n) = n + n \log n$
- $T(n) = O(n \log n)$

Sorting practice problem

- Consider the following array of int values.

```
[22, 11, 34, -5,  3, 40,  9, 16,
 6]
```

(e) Write the contents of the array after all the recursive calls of merge sort have finished (before merging).