CSE 373: Data Structures and Algorithms

Lecture 8: Sorting II

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 disk 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?

O(n log n) Comparison Sorting

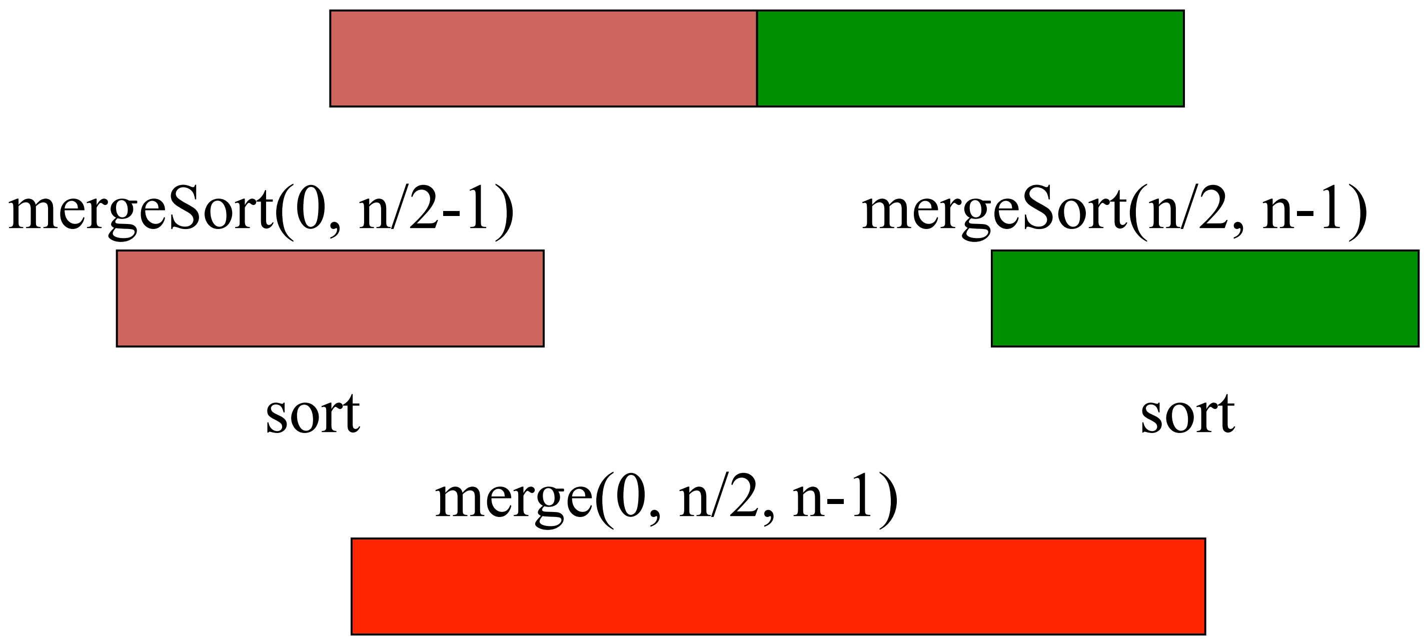
Merge sort

- **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

- Another "divide and conquer" algorithm
  - *divide* the list into two roughly equal parts
  - *conquer* by sorting the two parts
    - recursively divide each part in half, continuing until a part contains only one element (one element is sorted)
  - *combine* the two parts into one sorted list

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

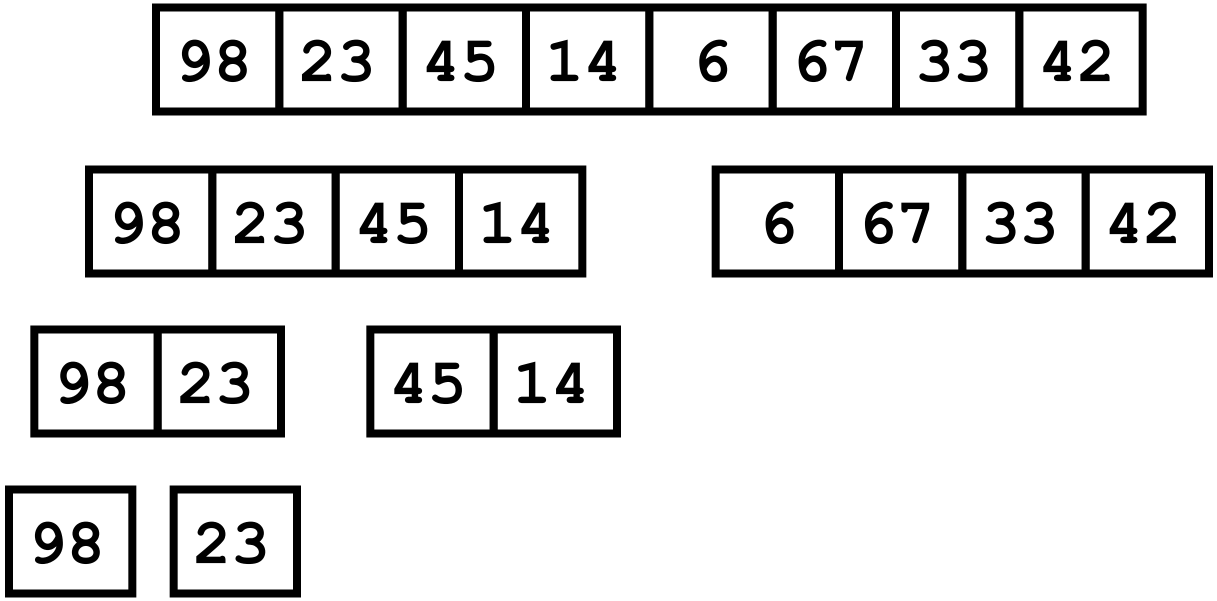
98	23	45	14	6	67	33	42

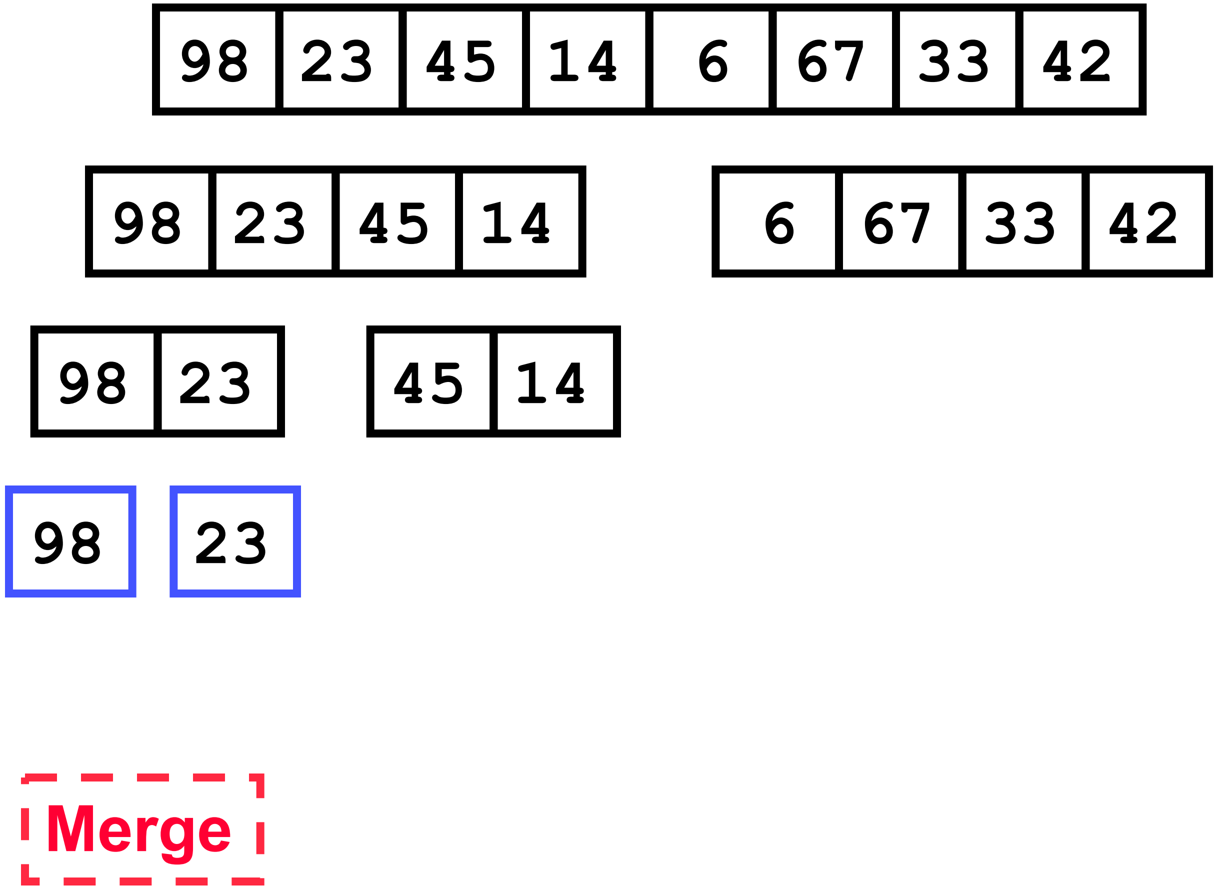
98	23	45	14

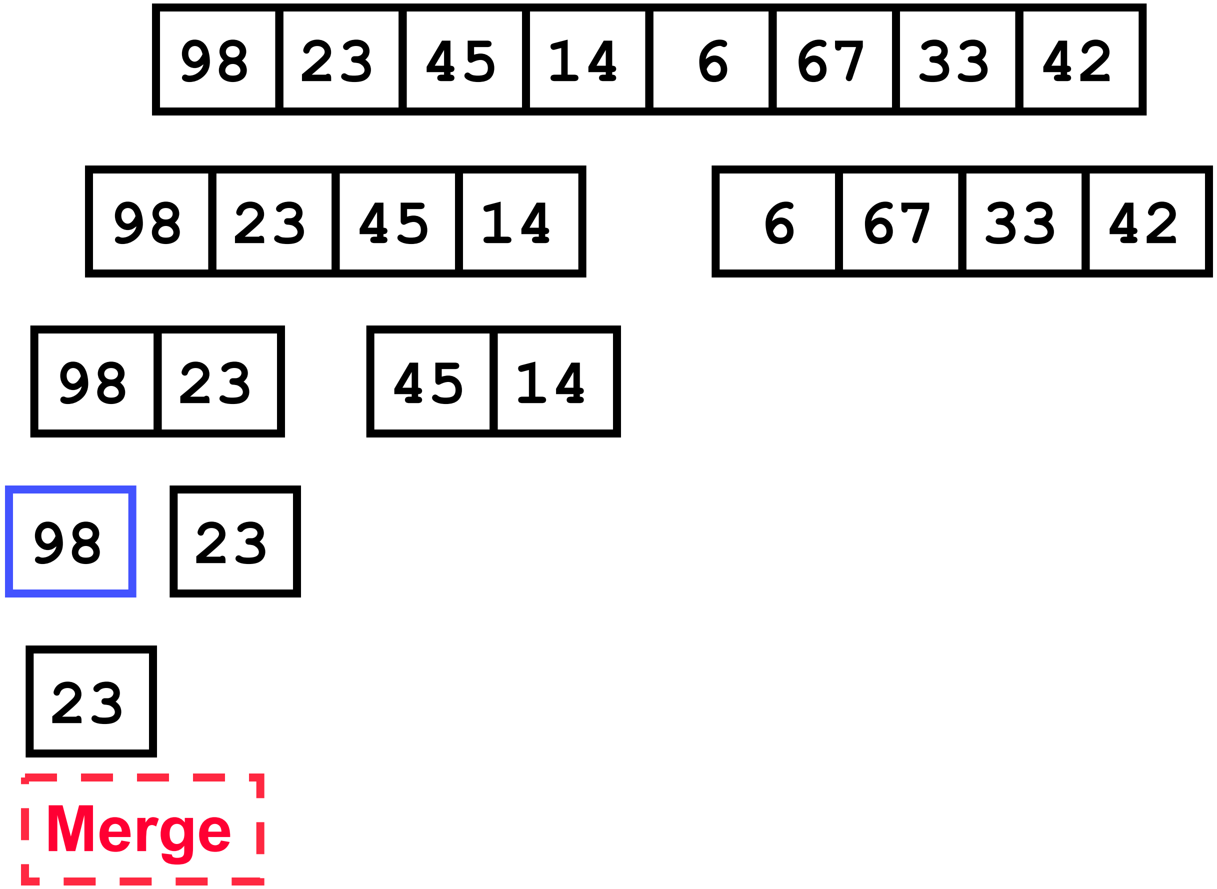
6	67	33	42

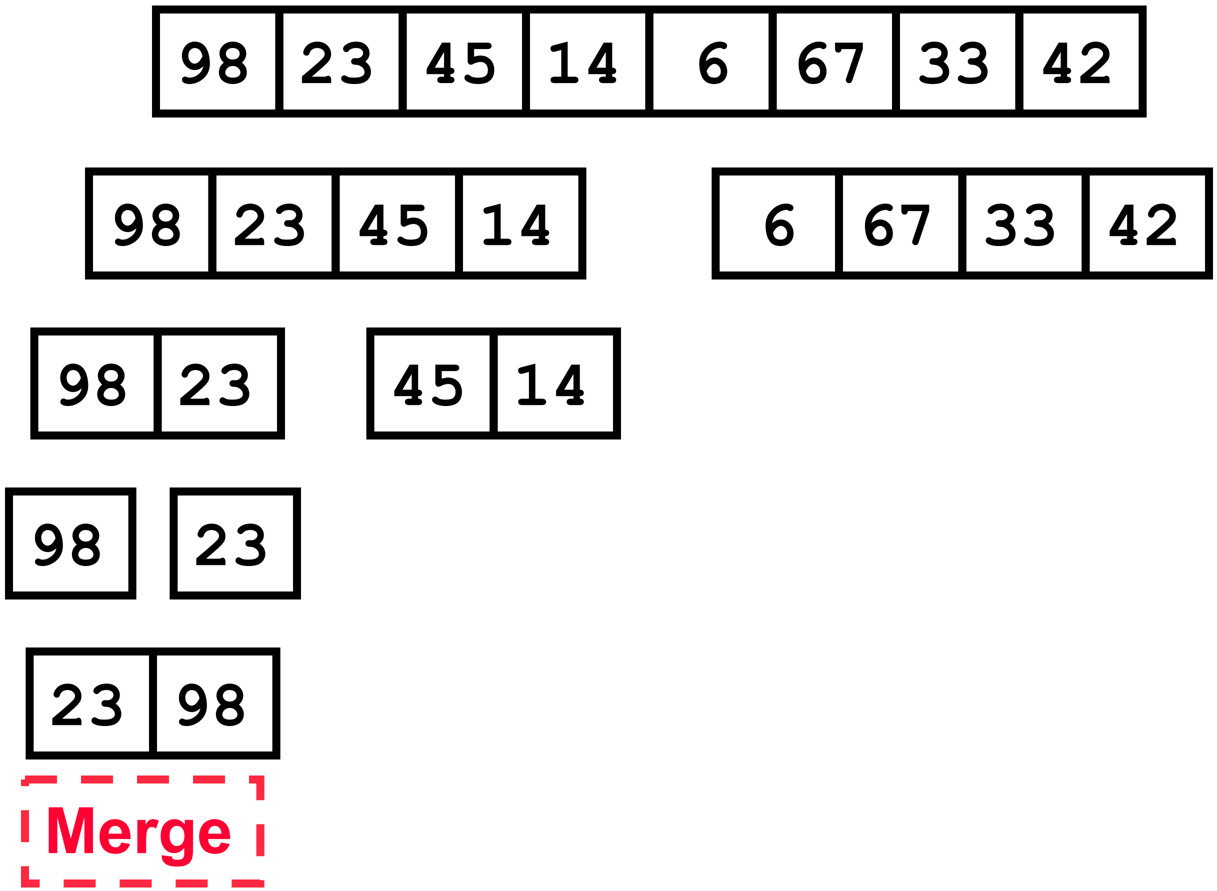
98	23

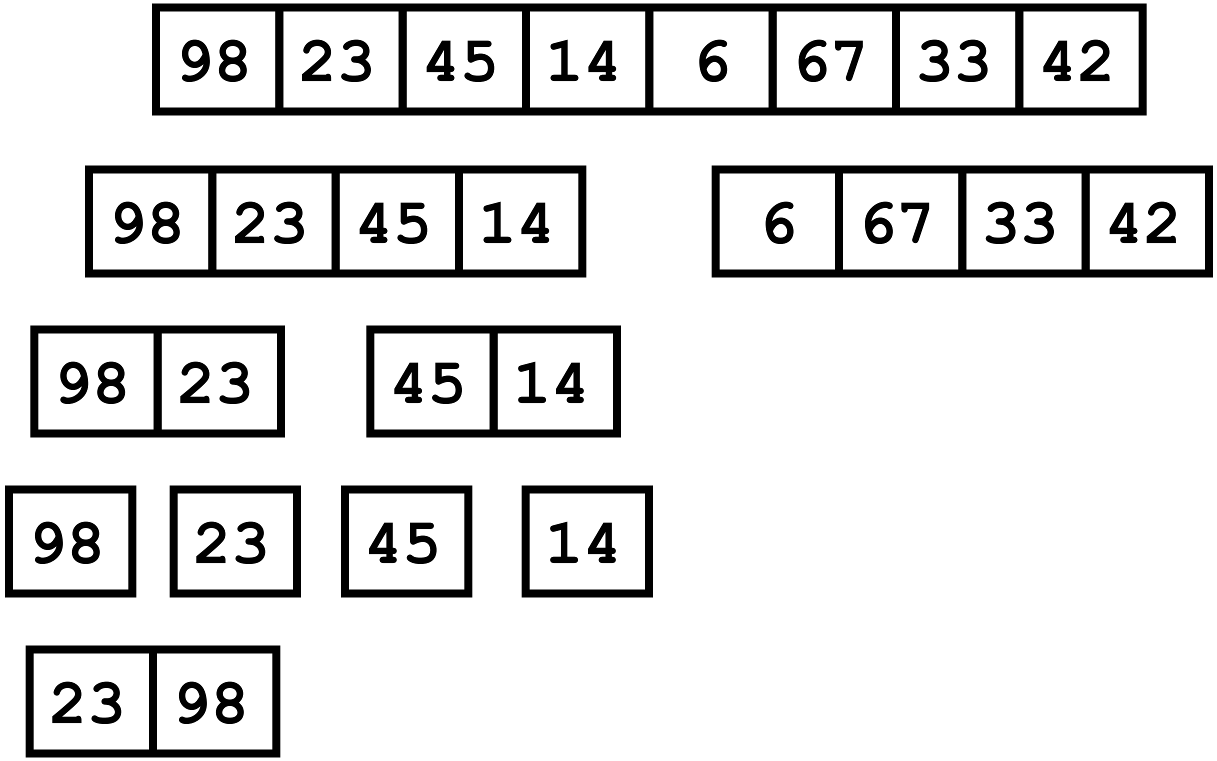
45	14

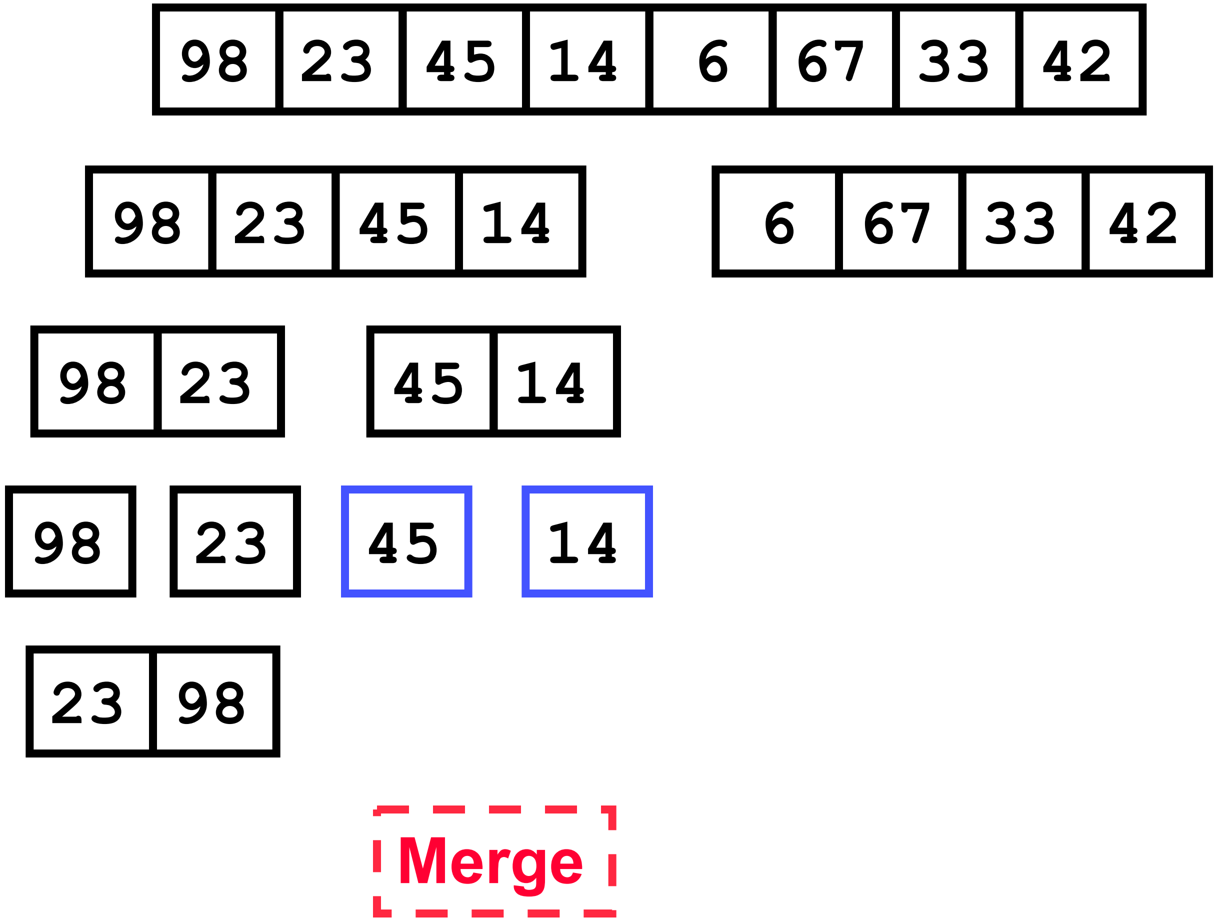

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

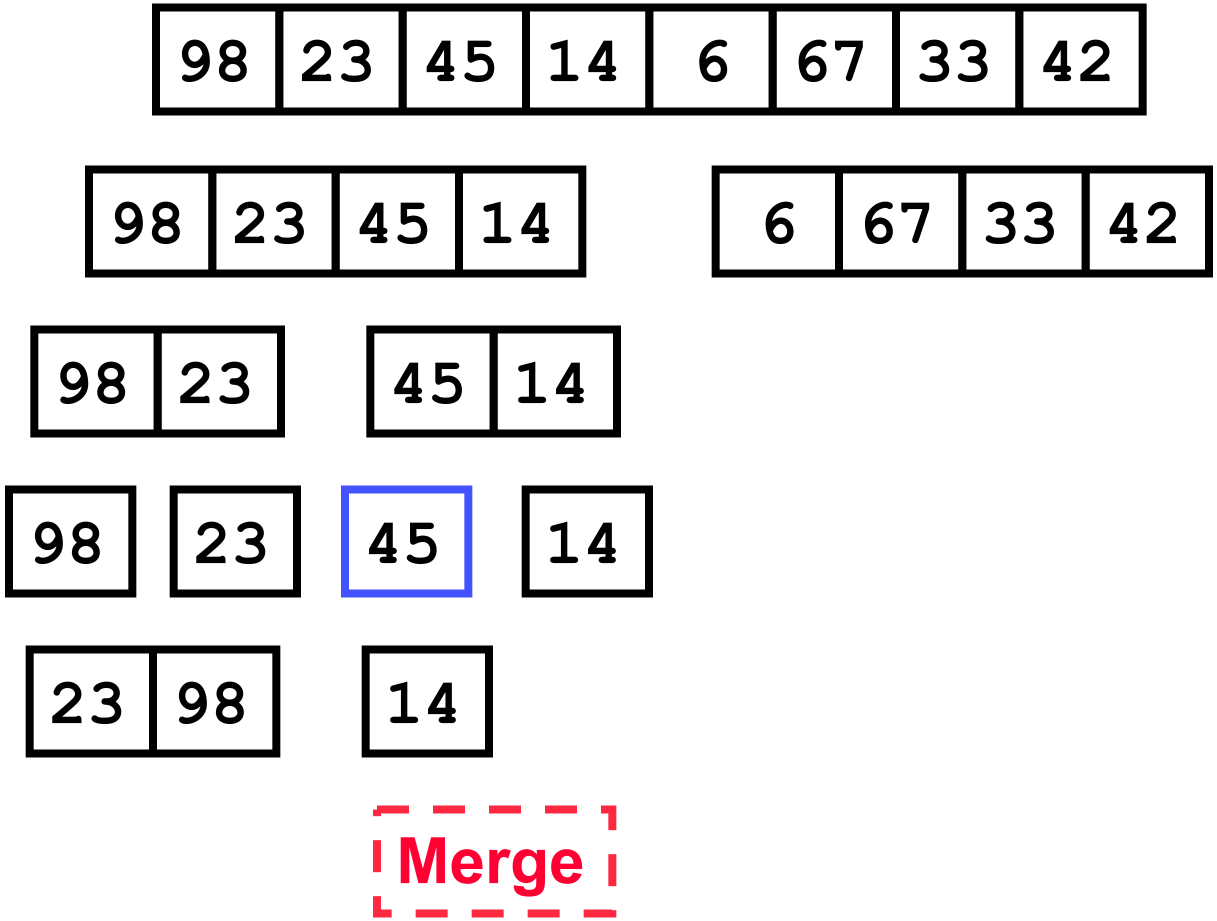

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

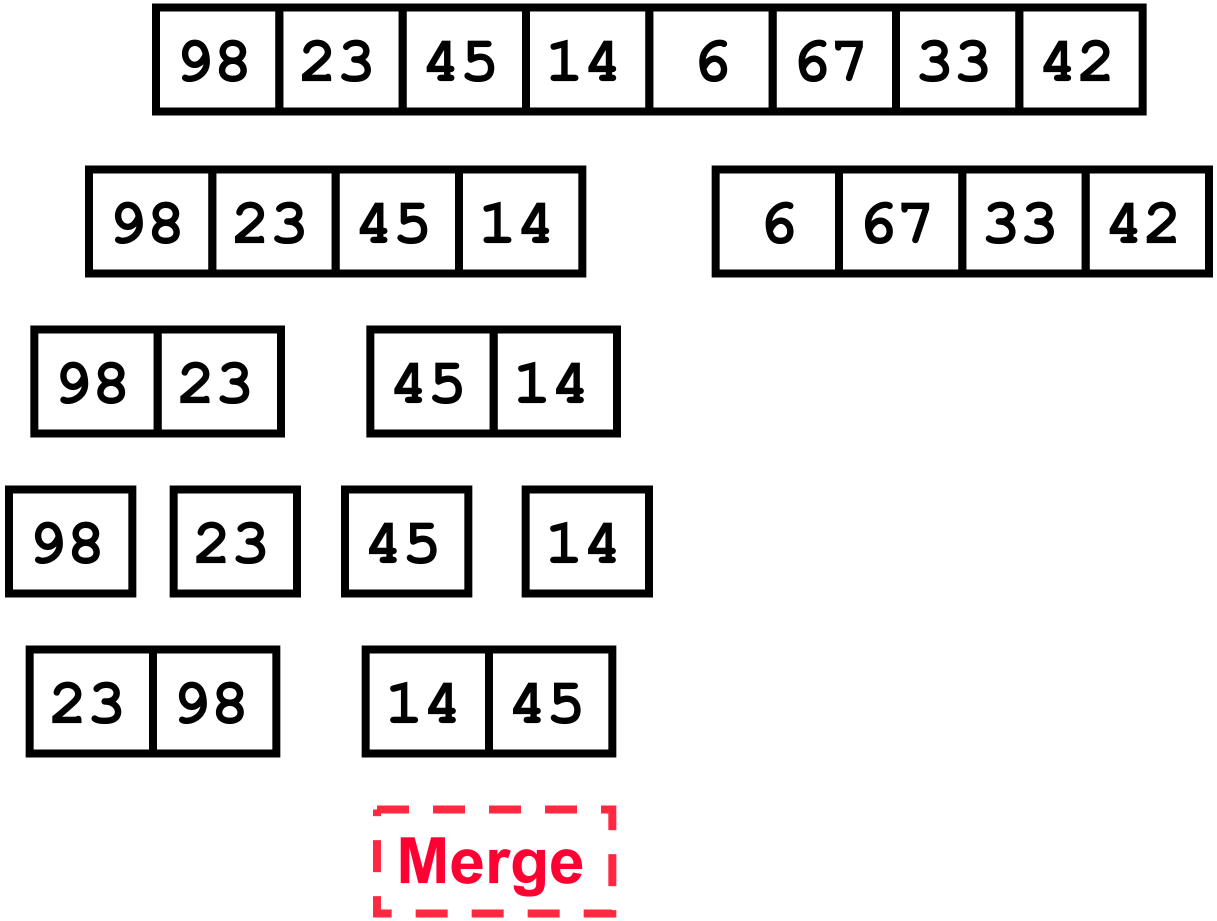

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge


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

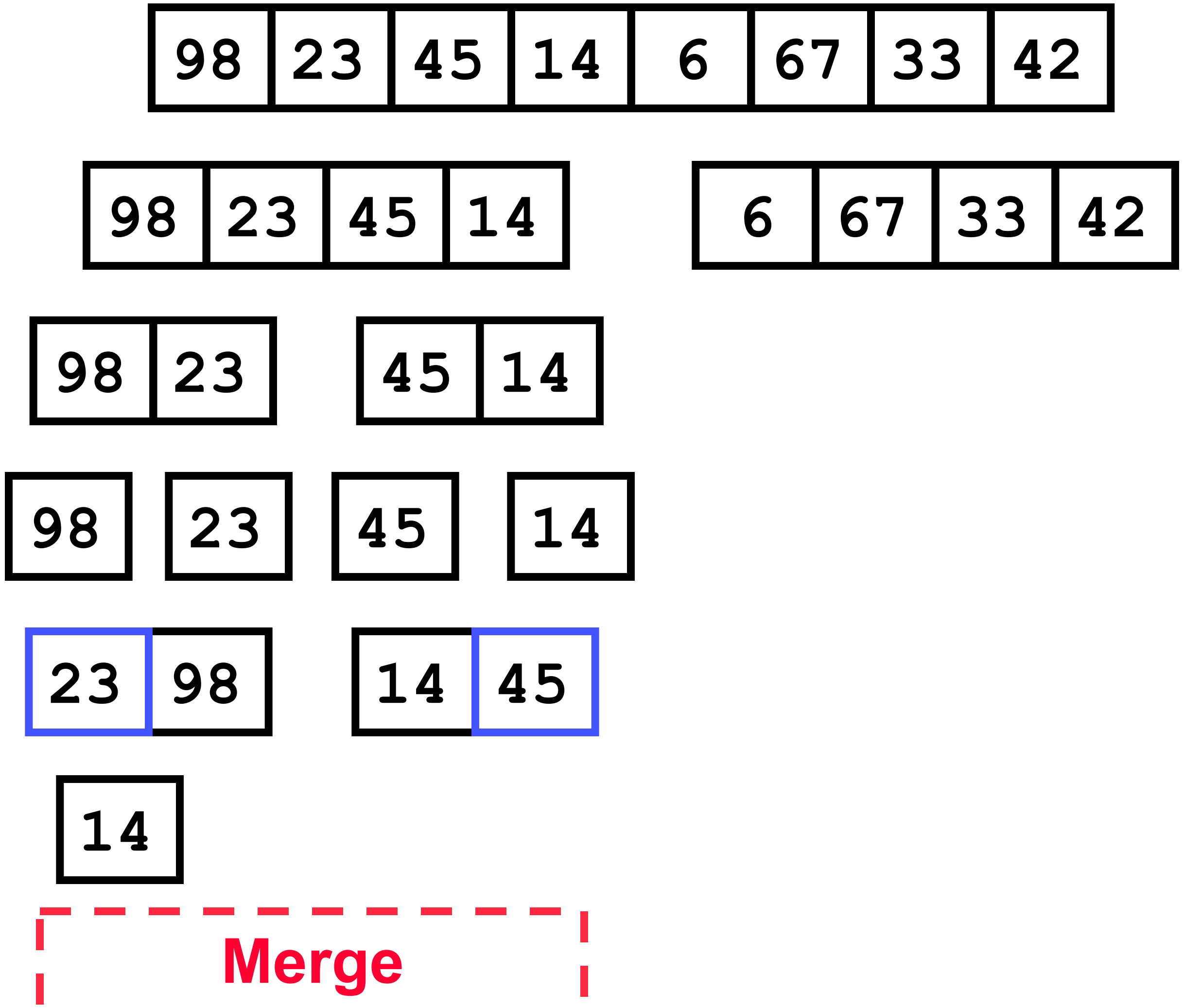
45	14

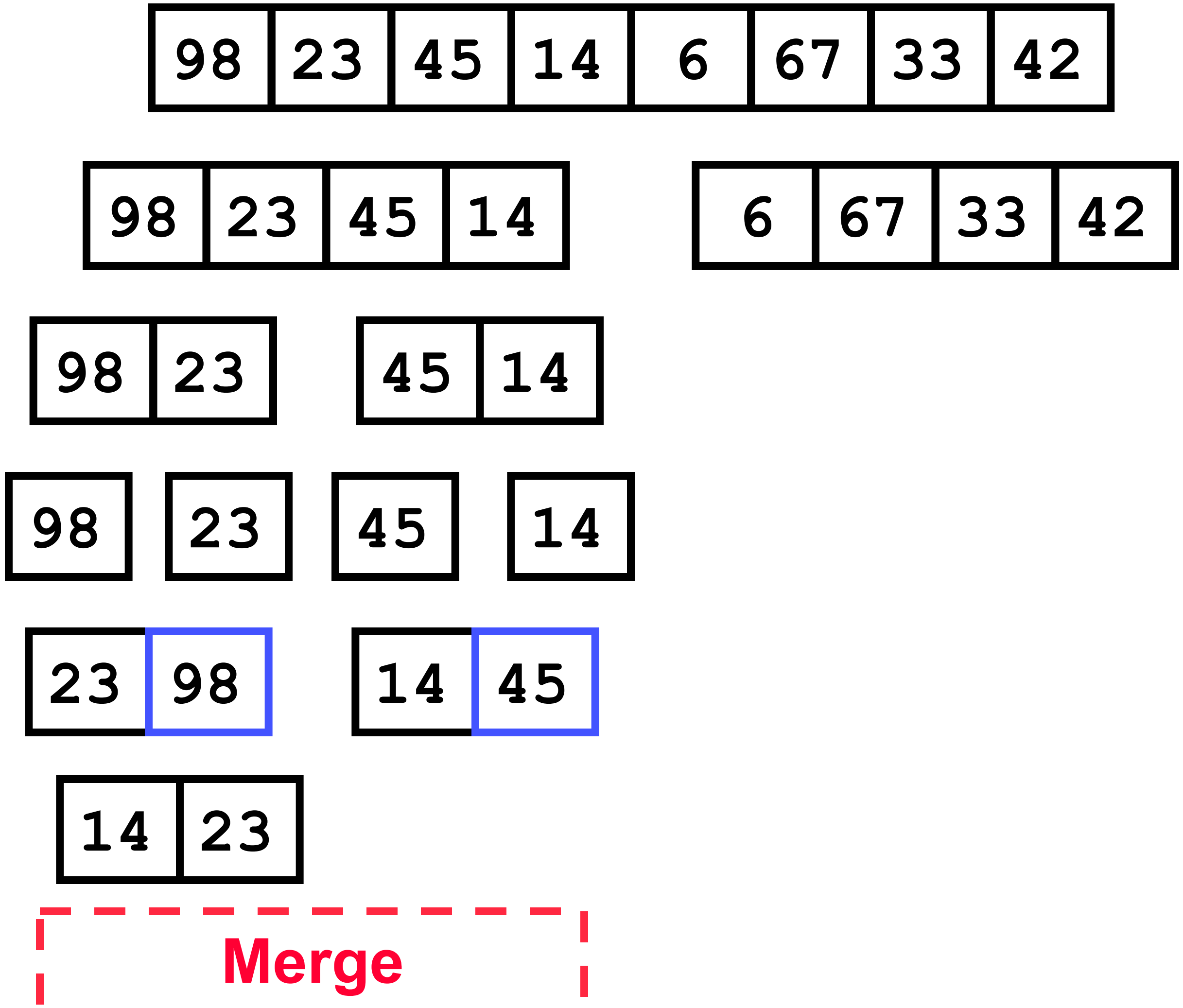
98	23	45	14

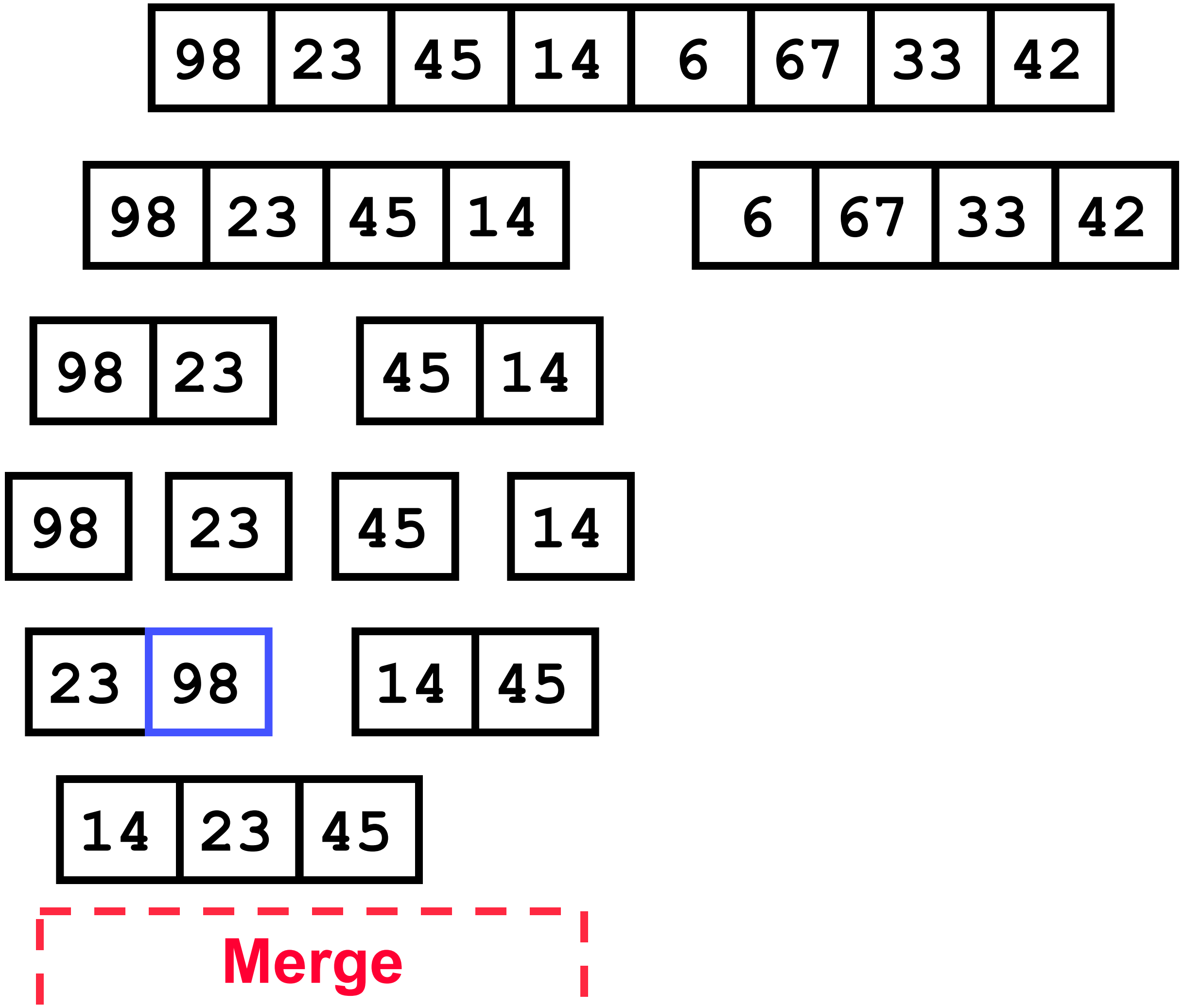
23	98

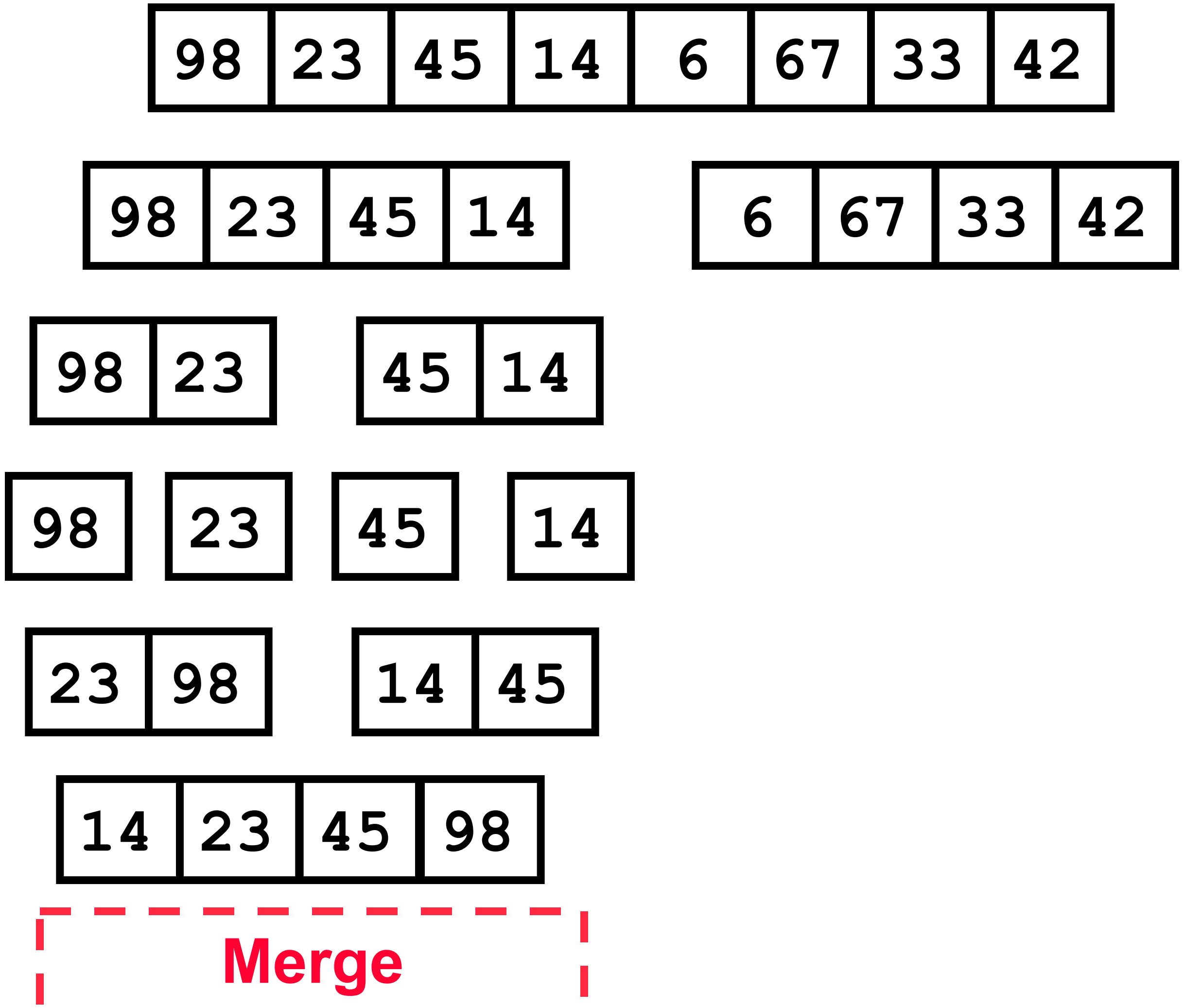
14	45

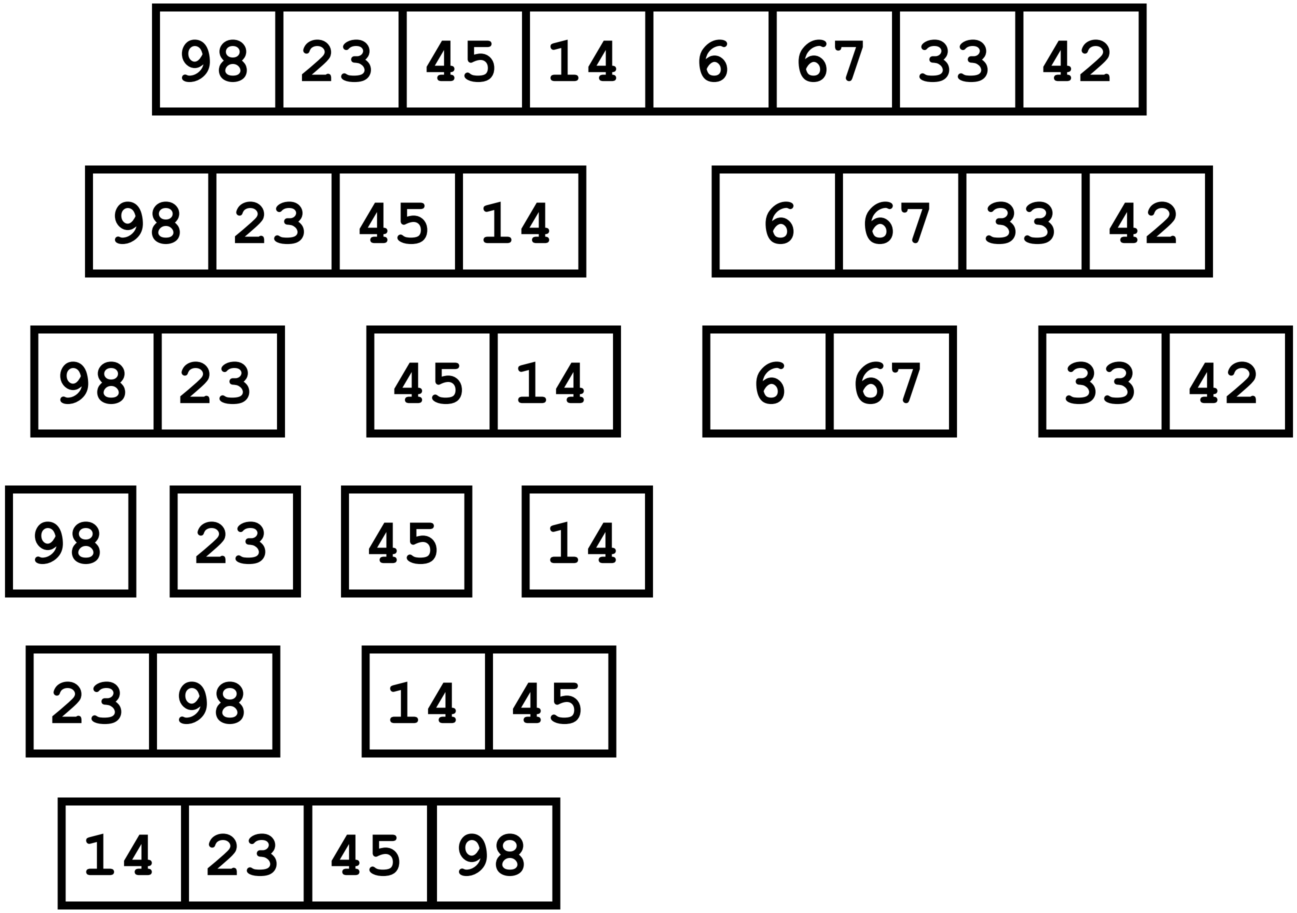
Merge

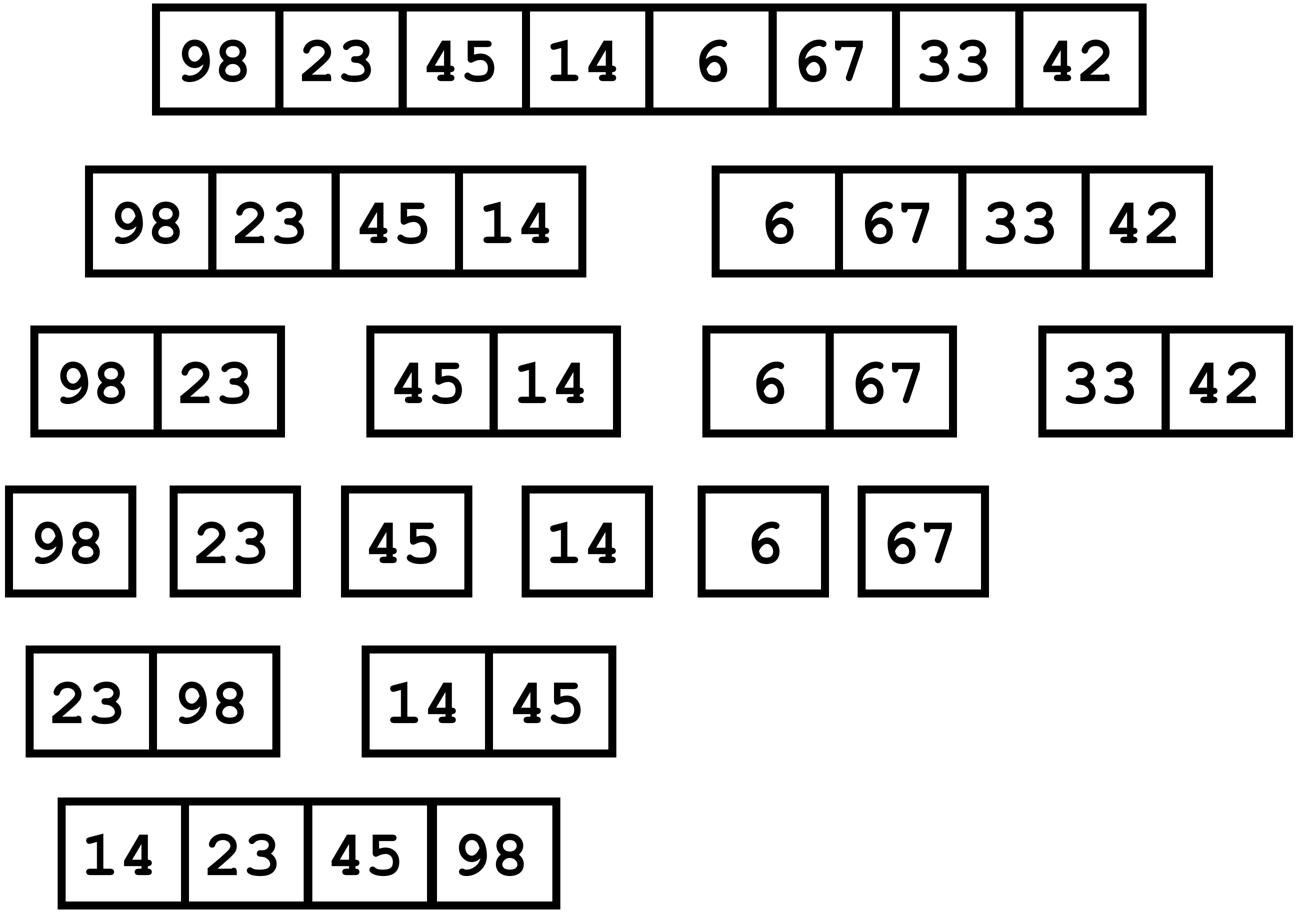

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

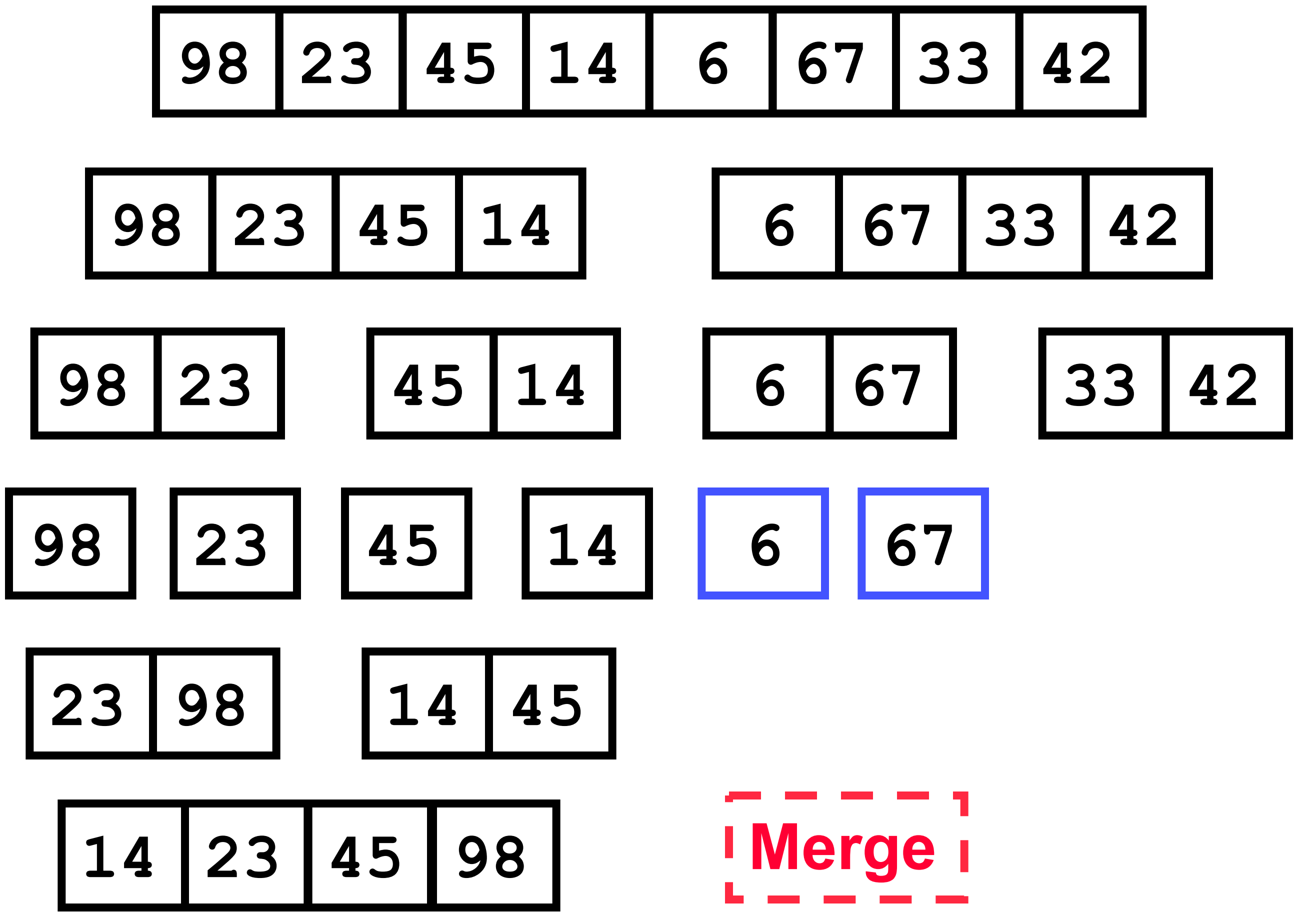

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

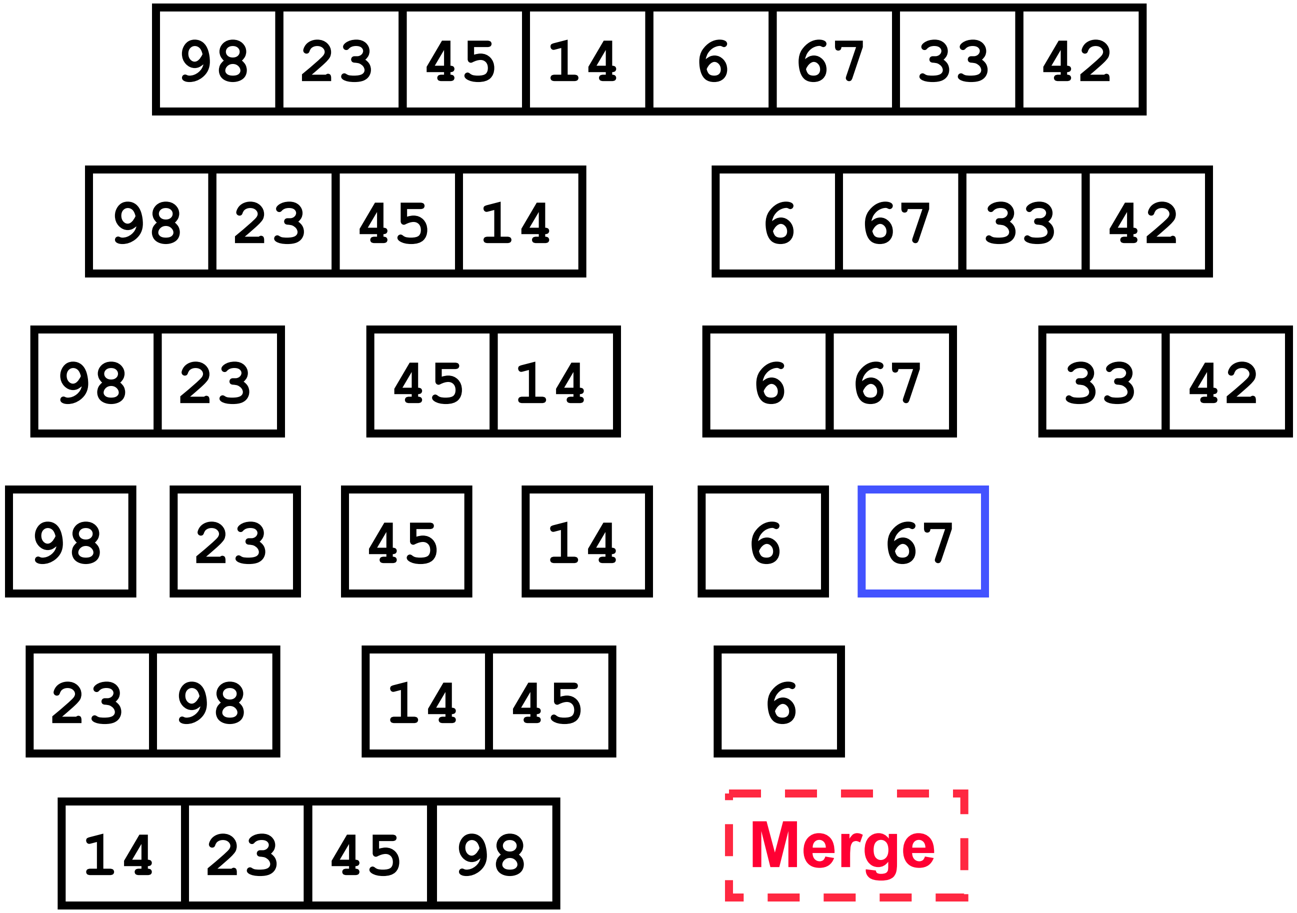

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

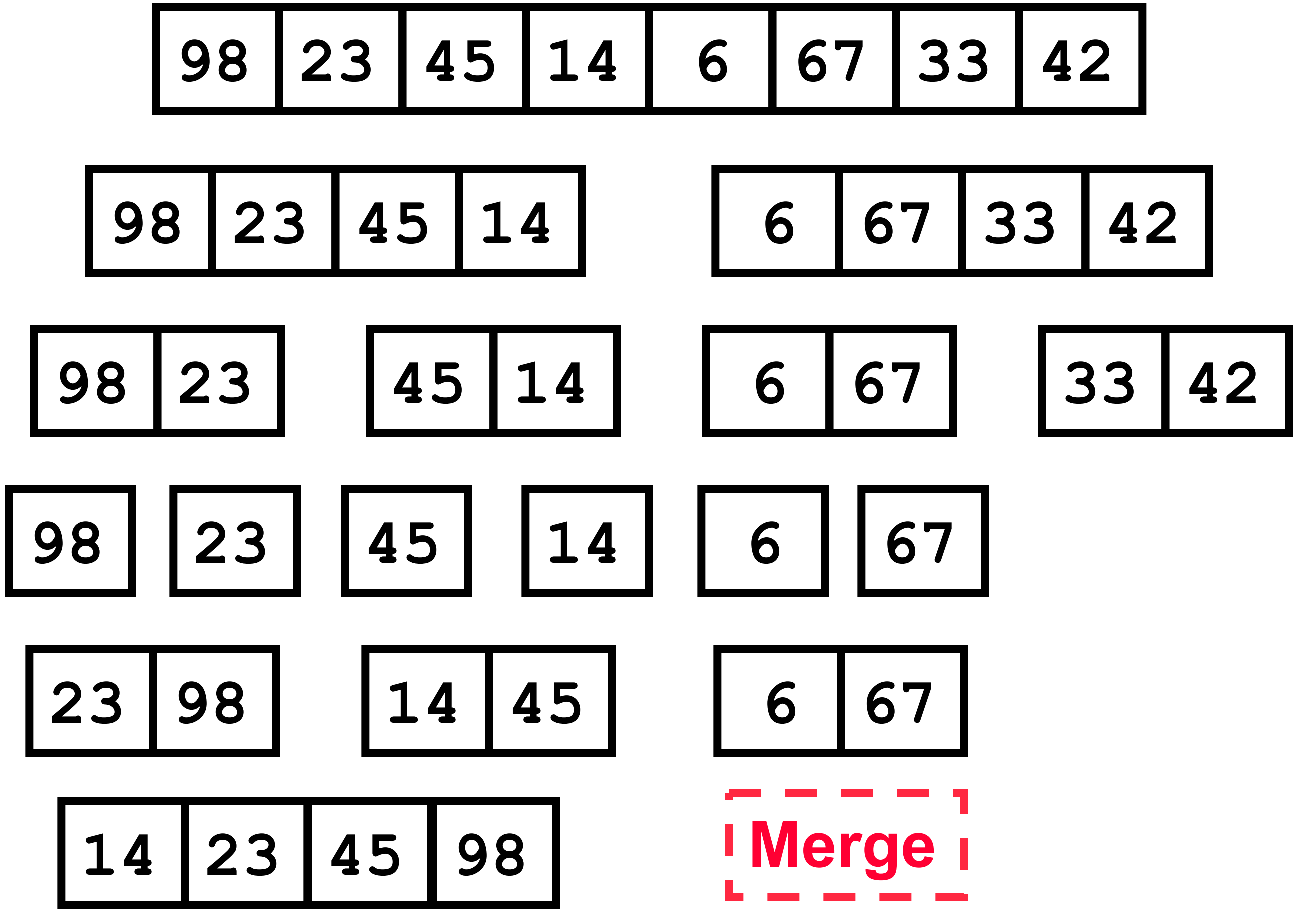

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

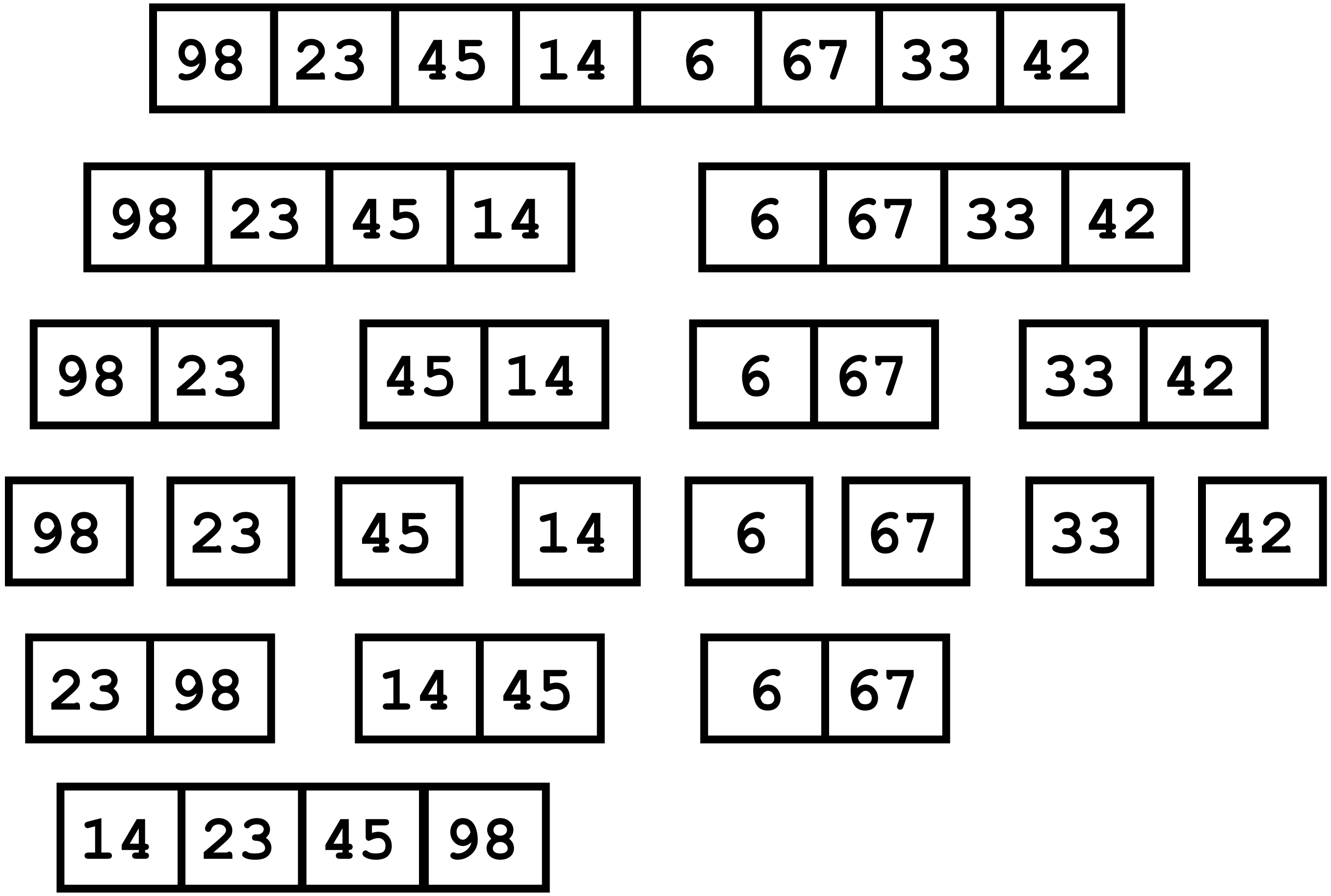

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

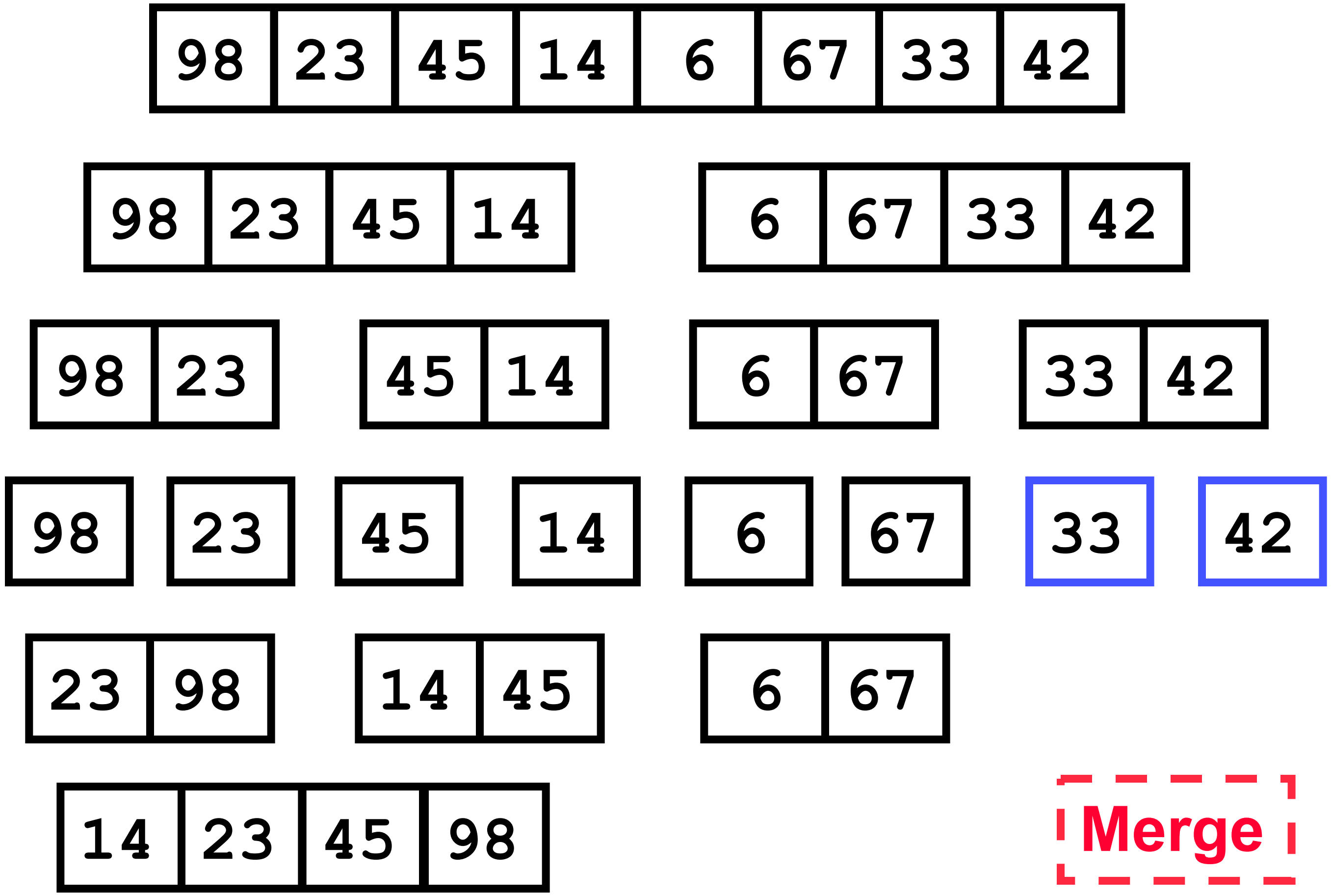

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

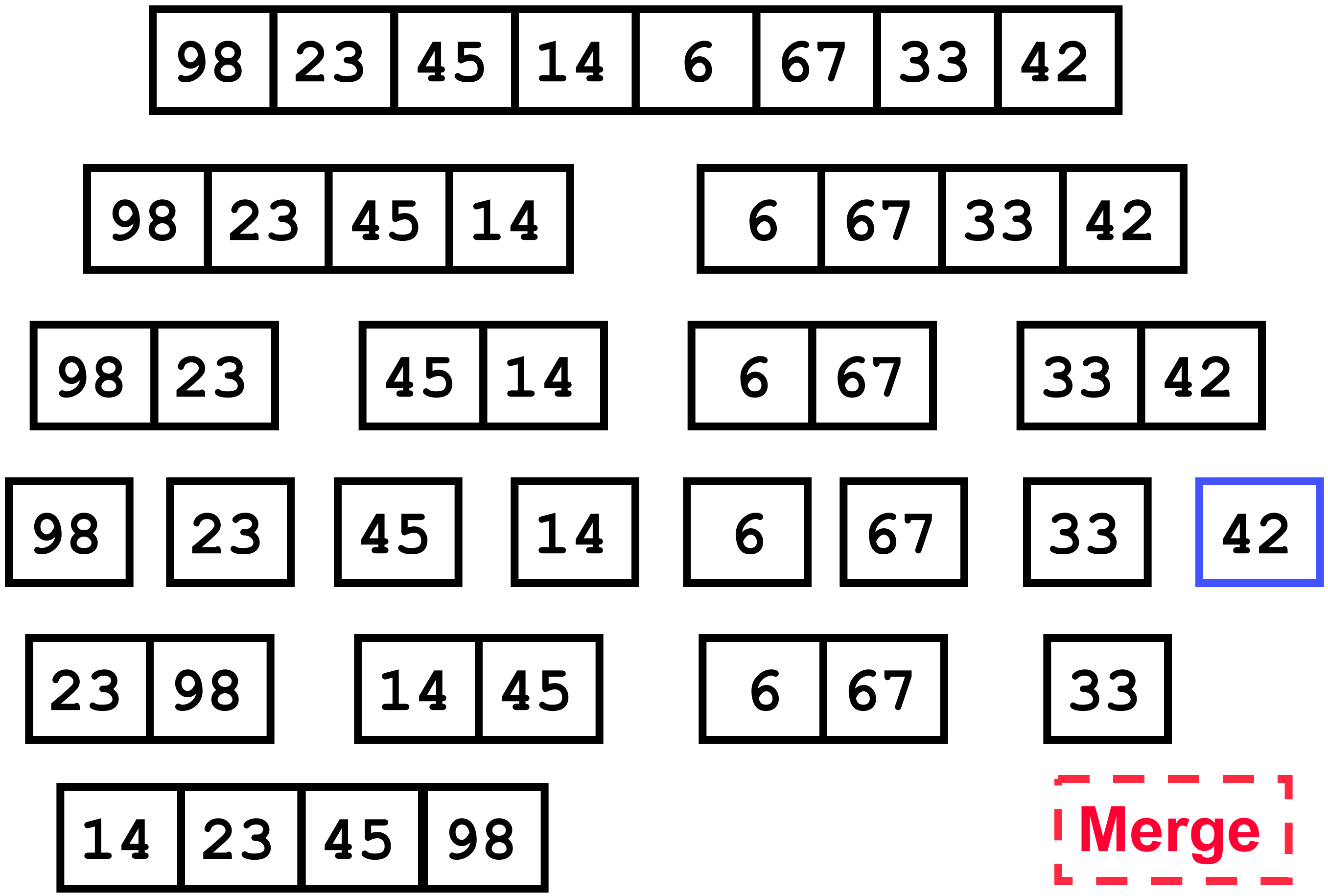

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

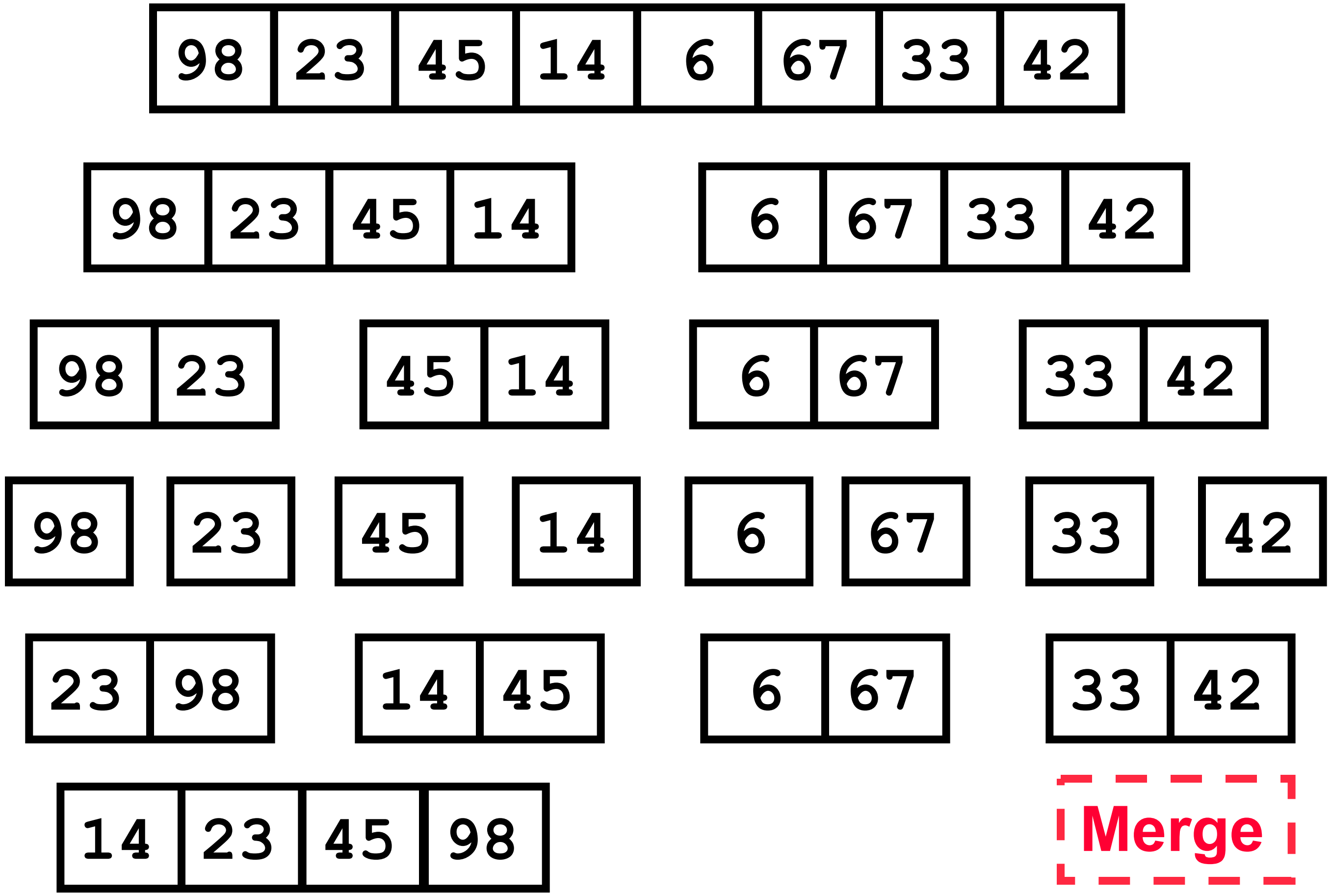

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge

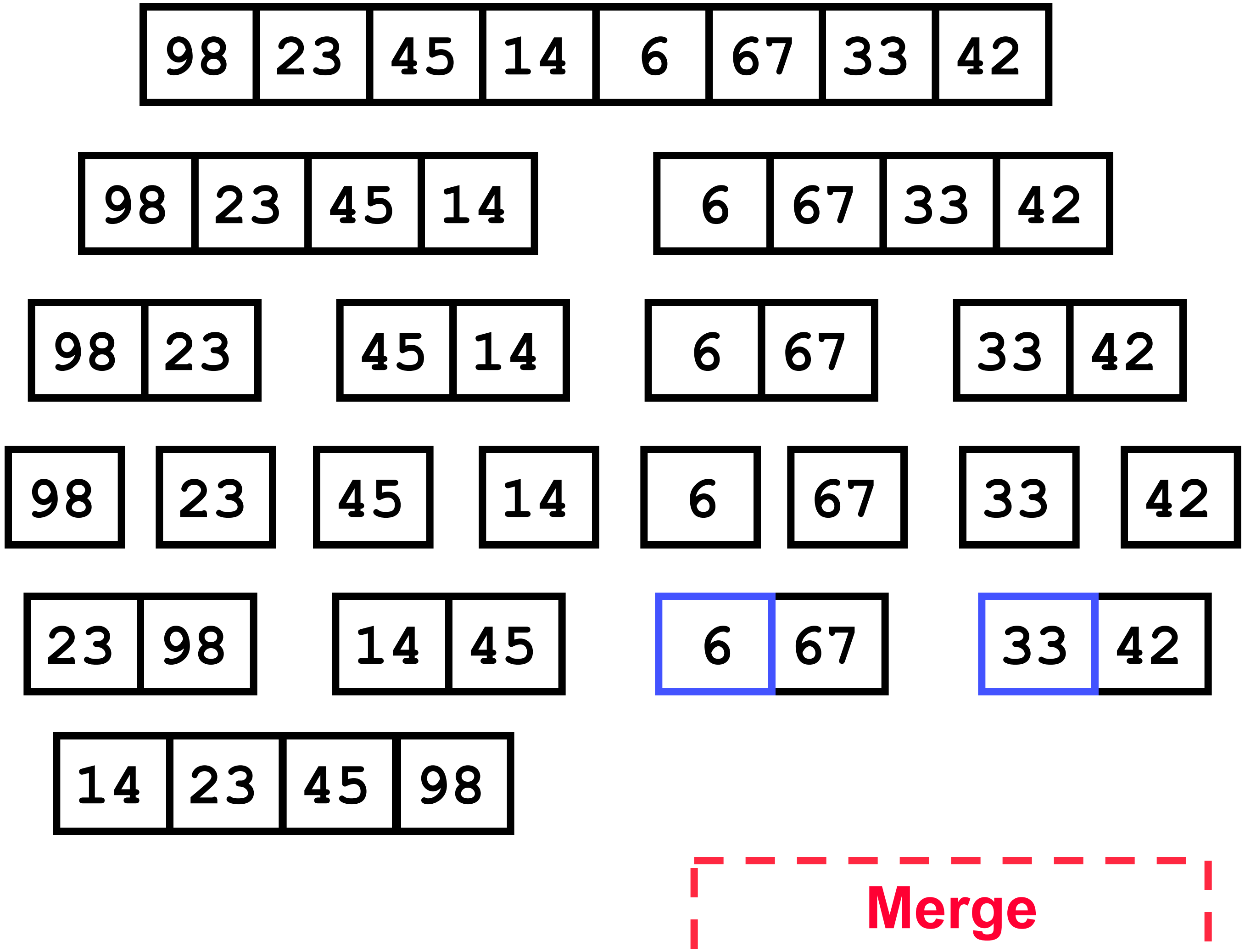

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6 67
14 23 45 98
Merge

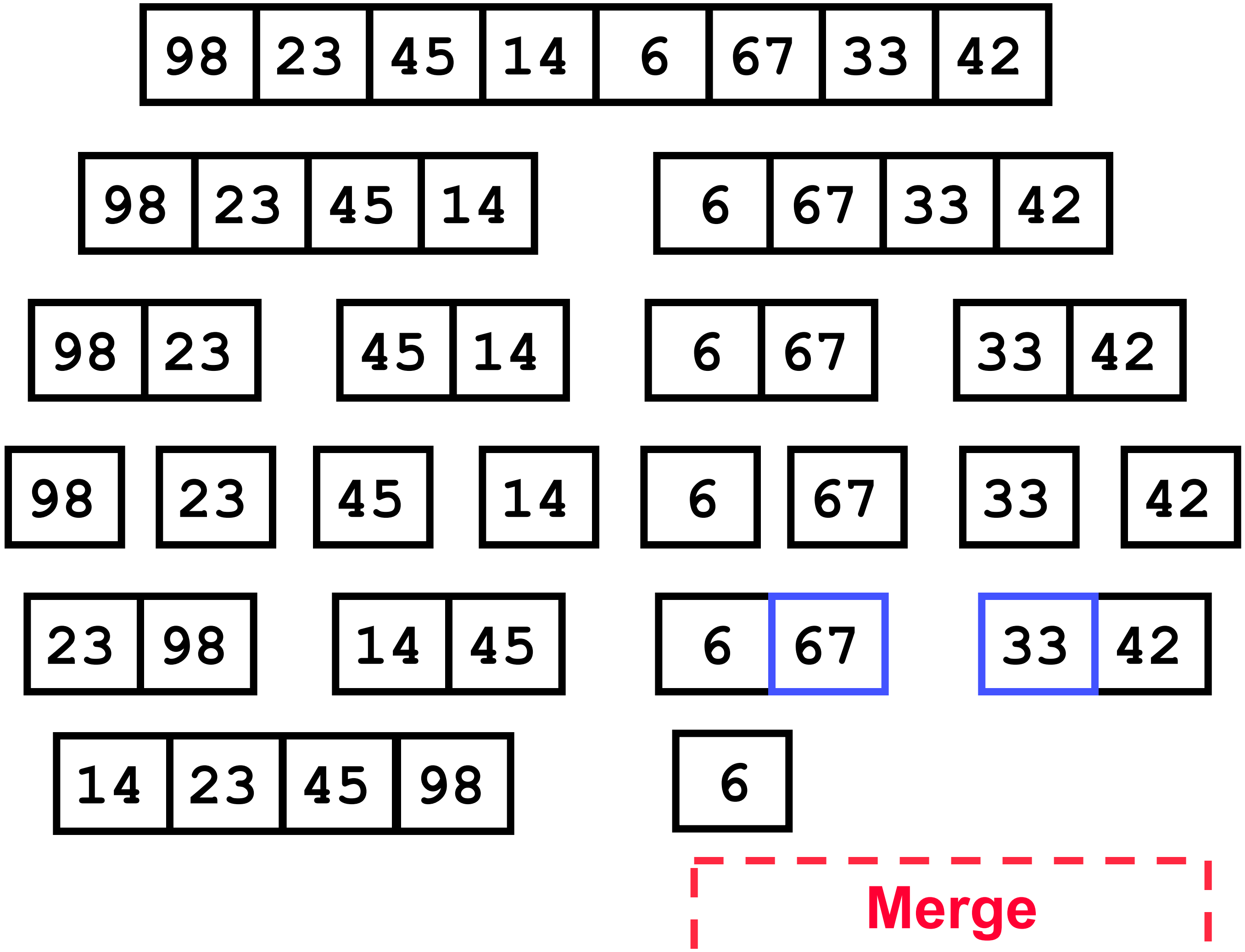

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98

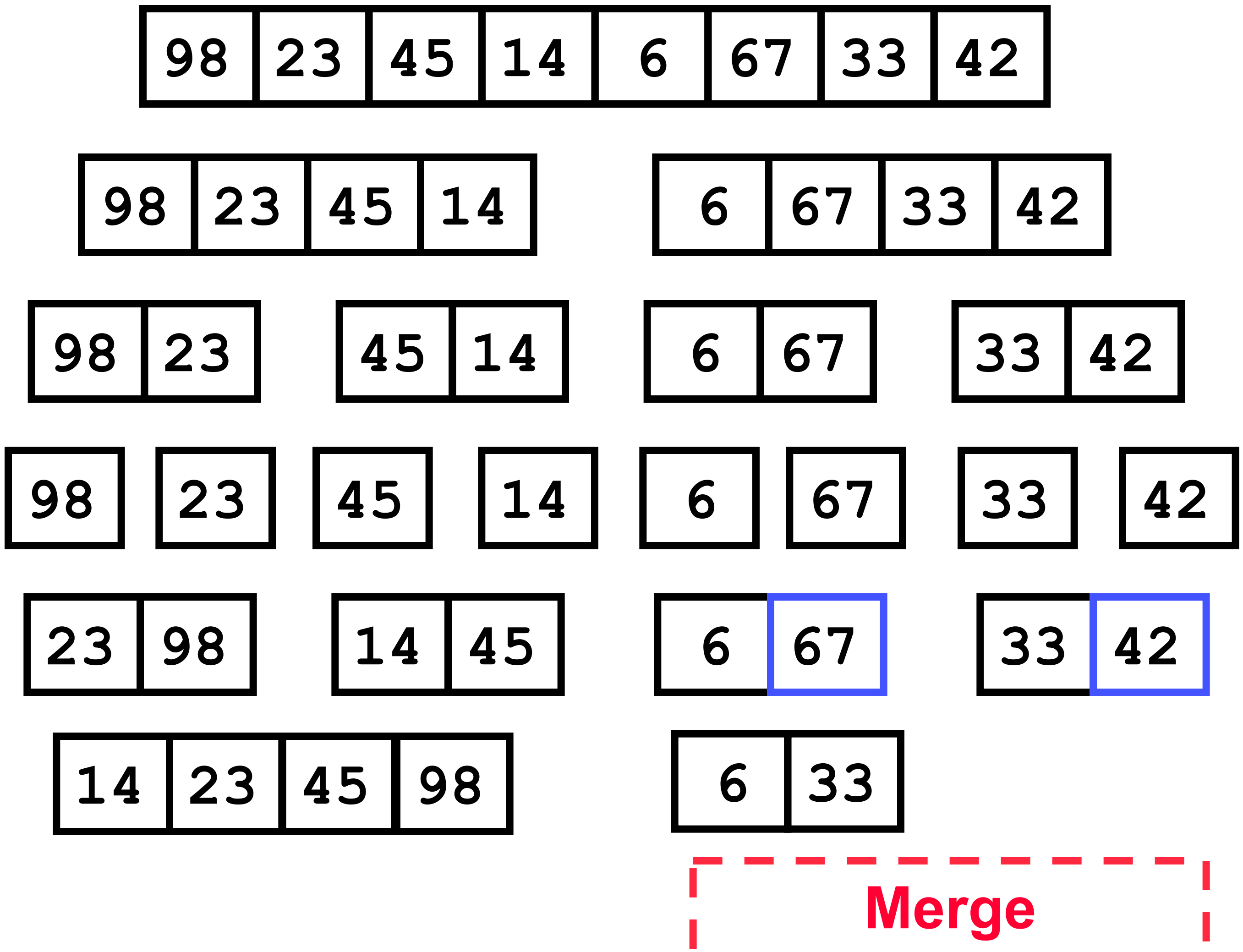

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98
Merge

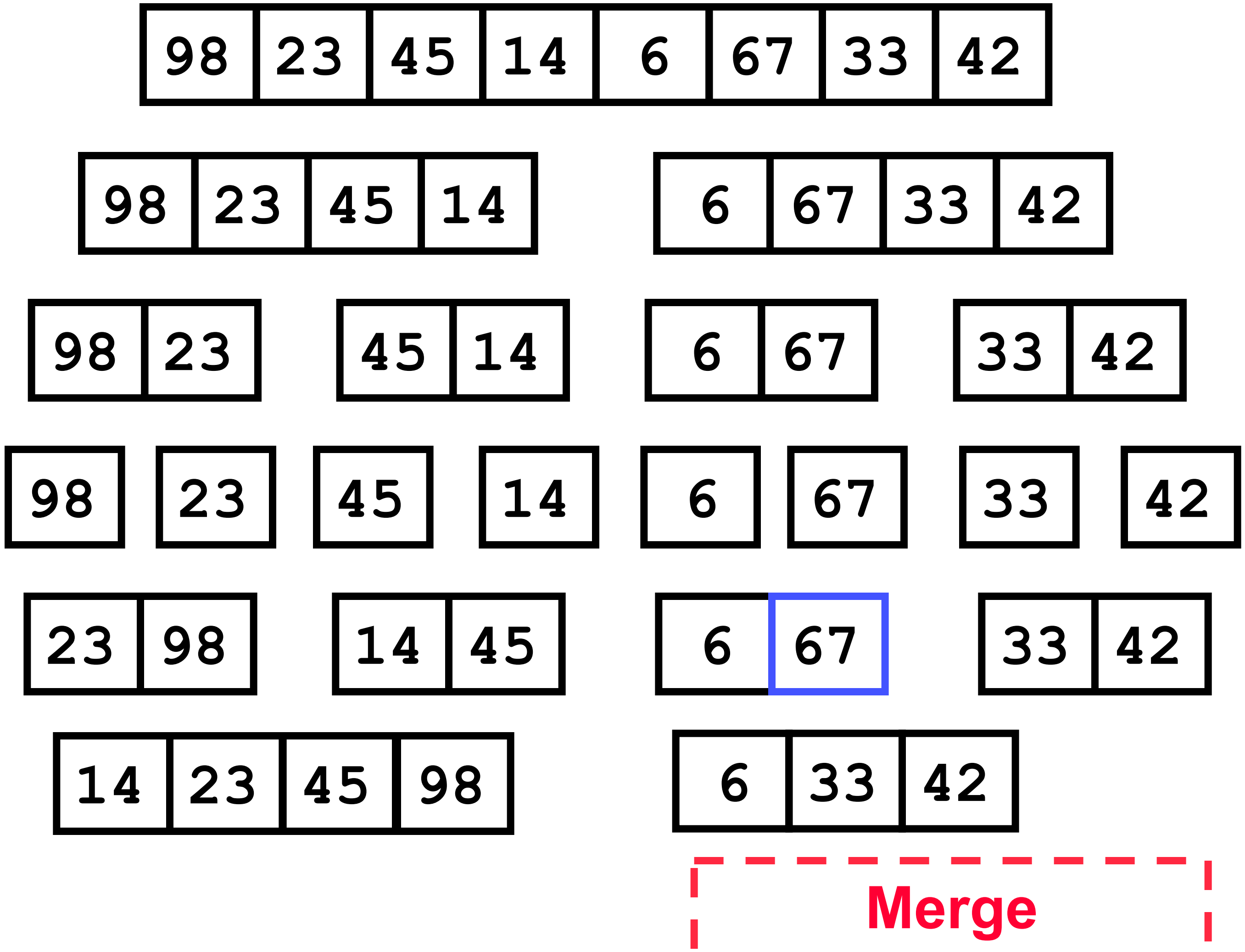

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge

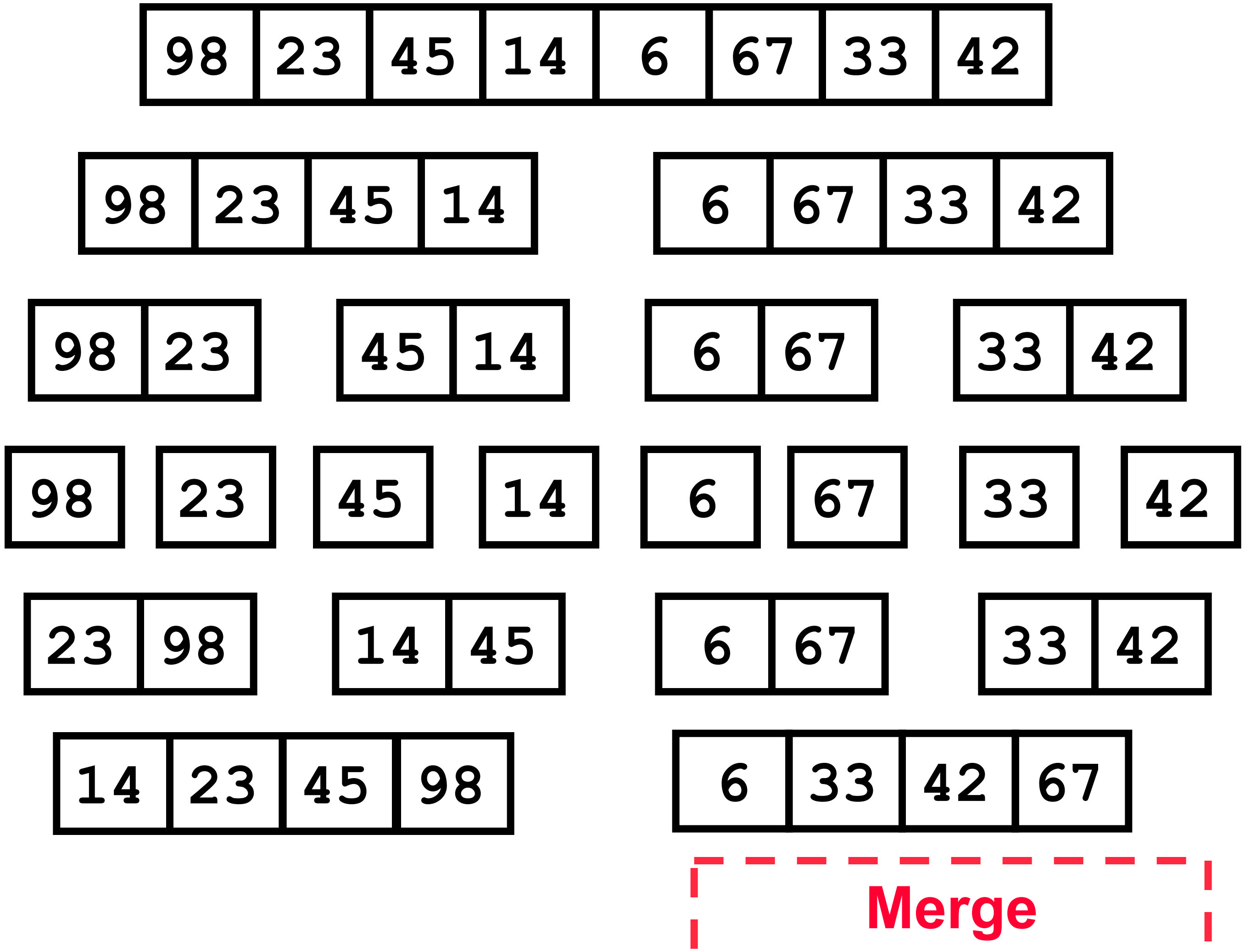

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

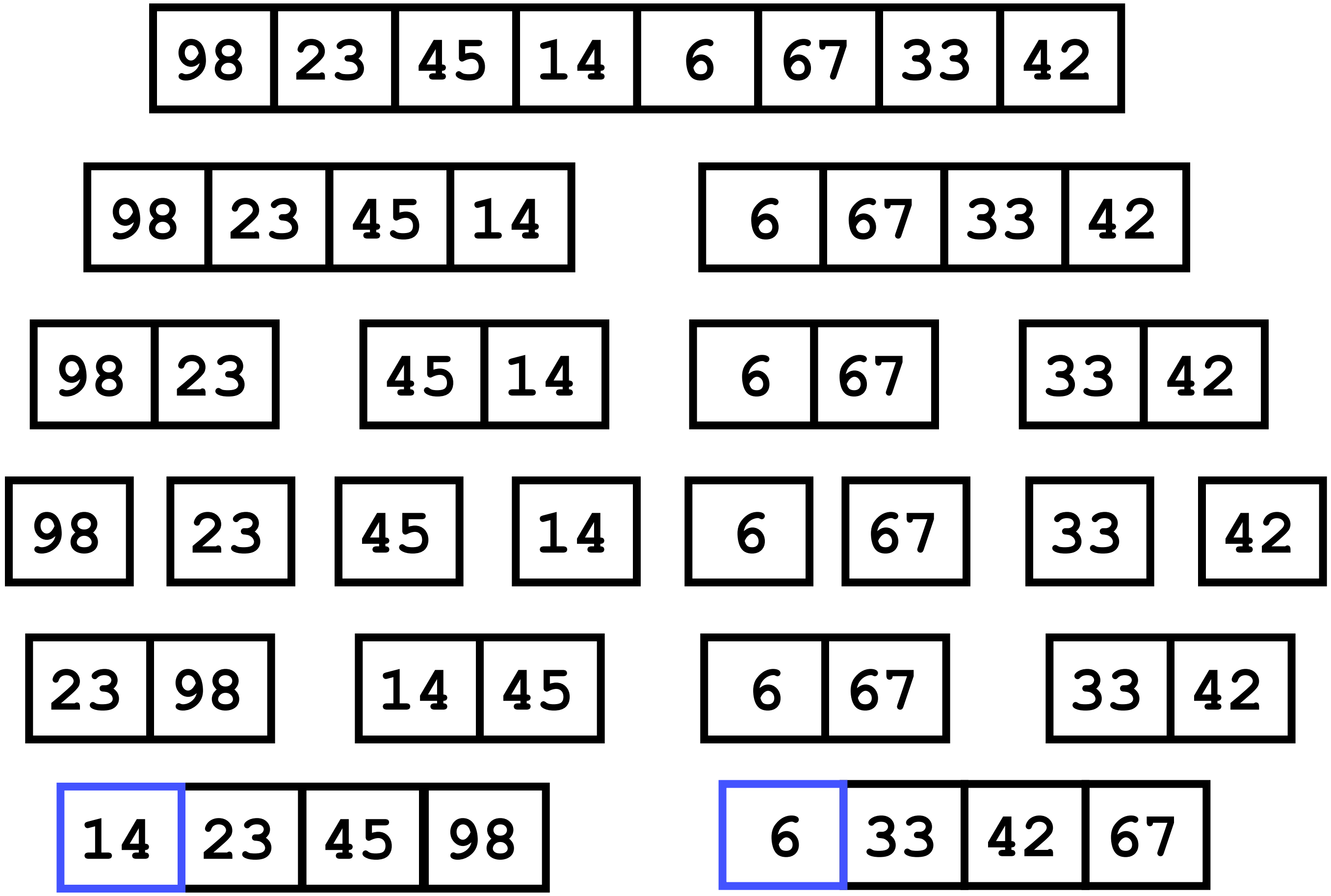

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge


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

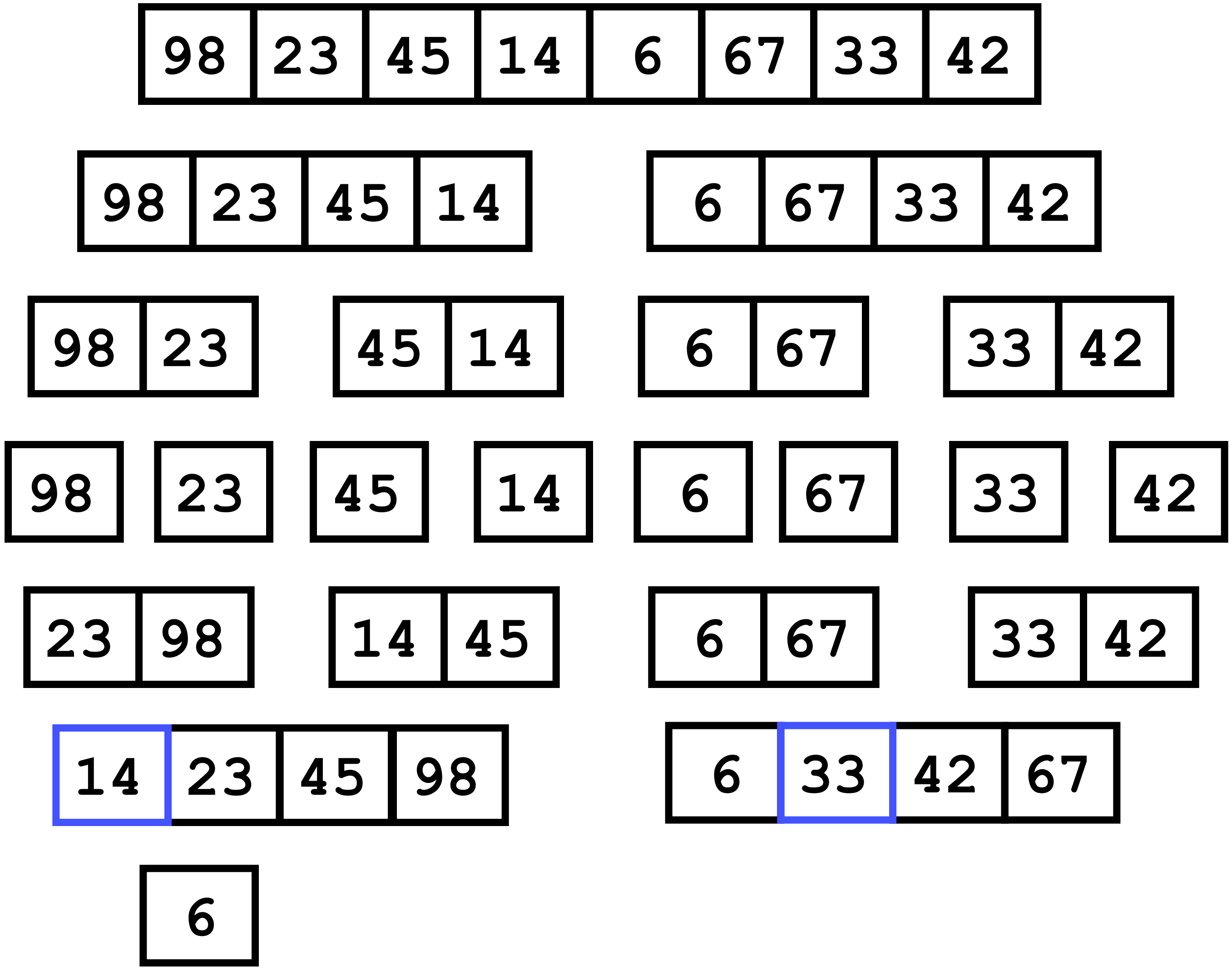

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33
Merge


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

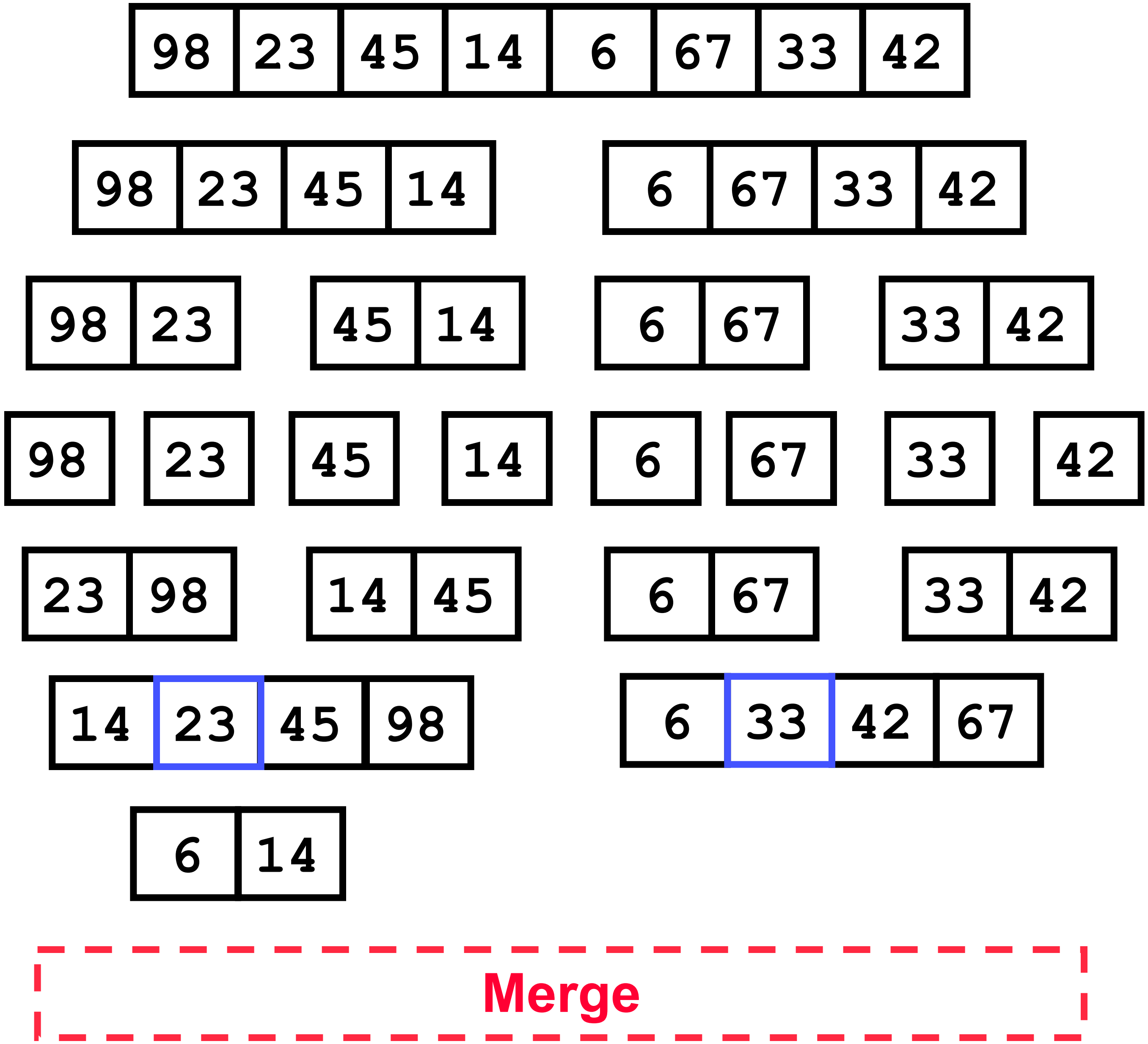

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

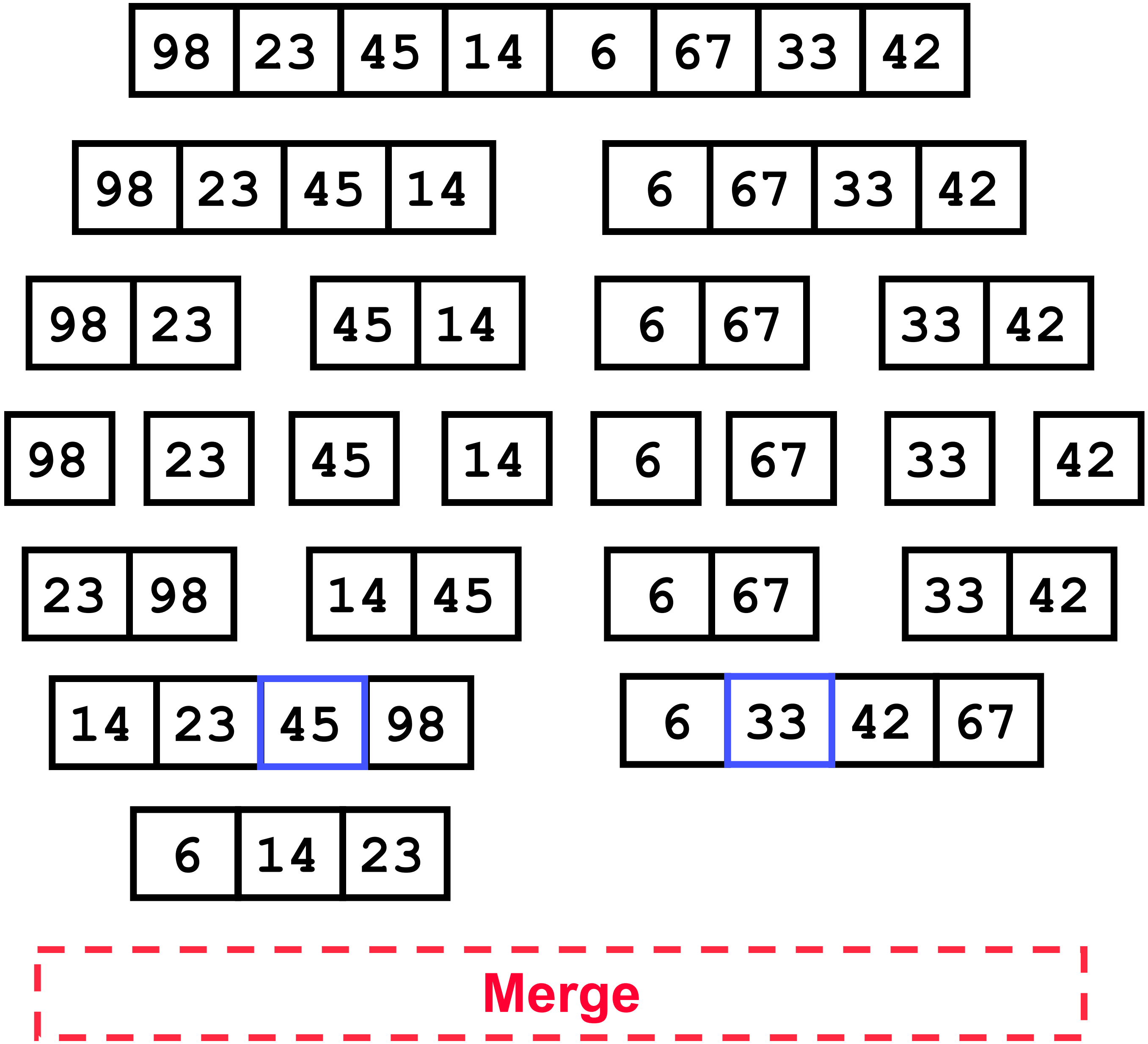

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67

Merge

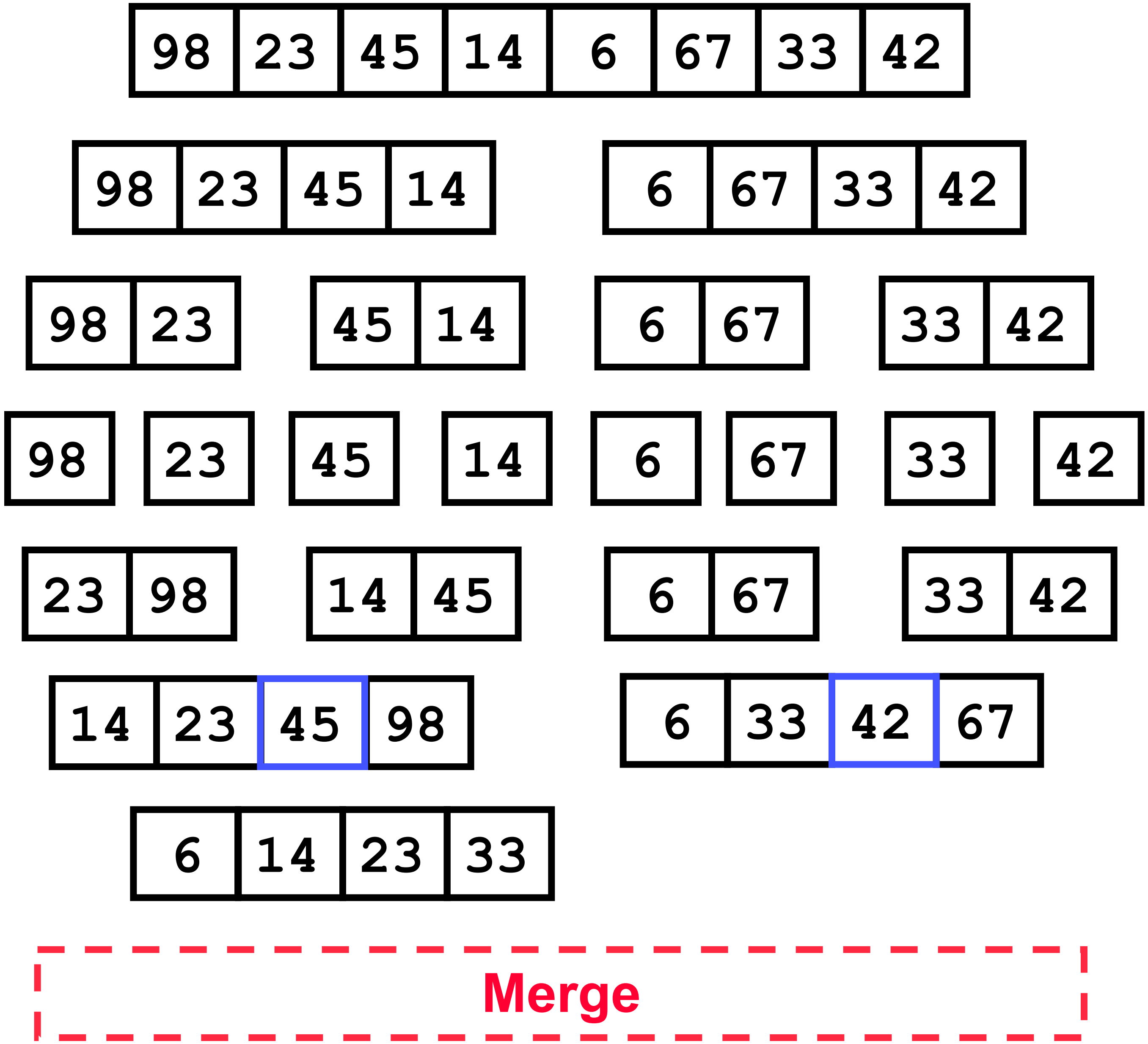

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6

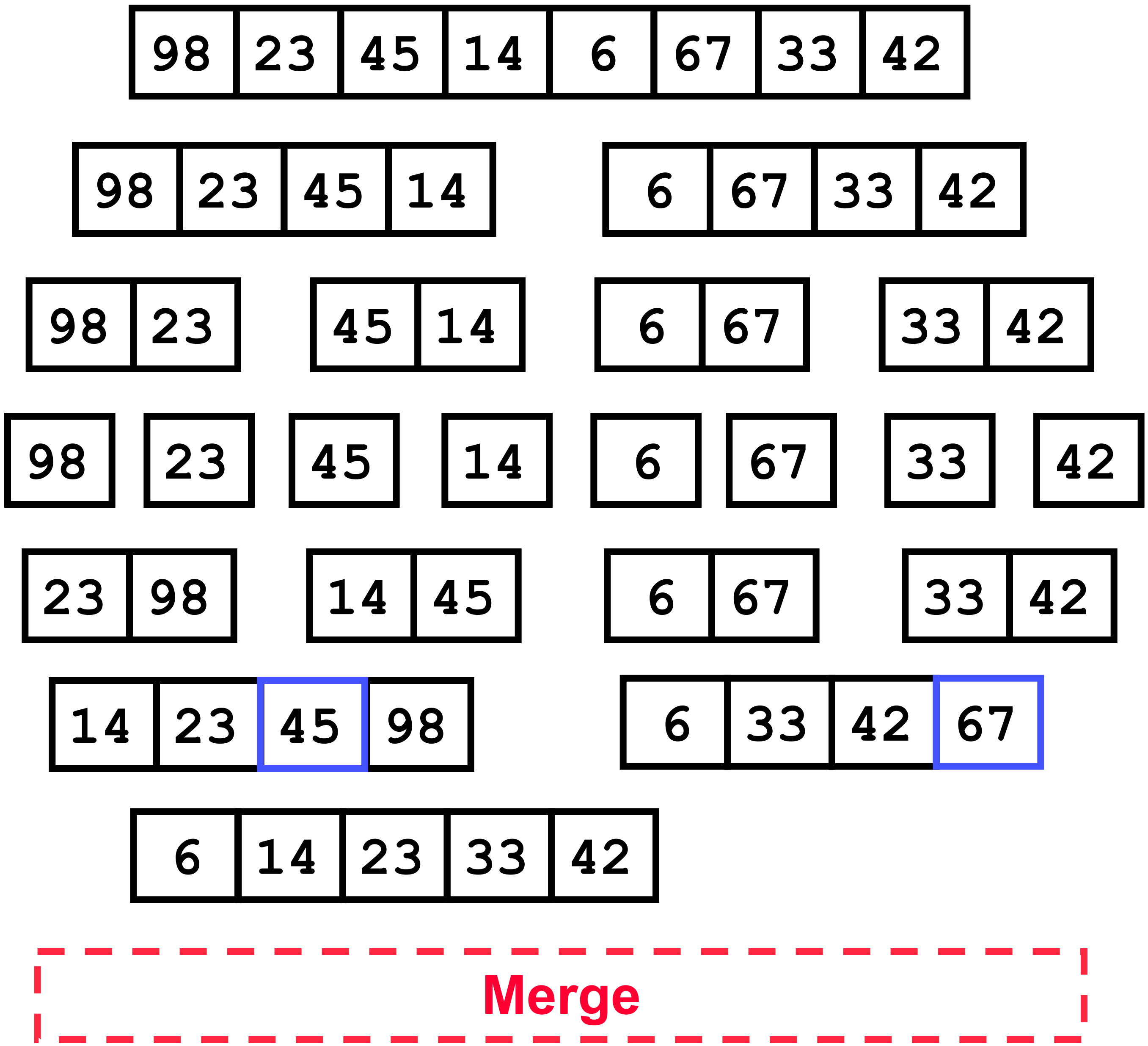
Merge


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]

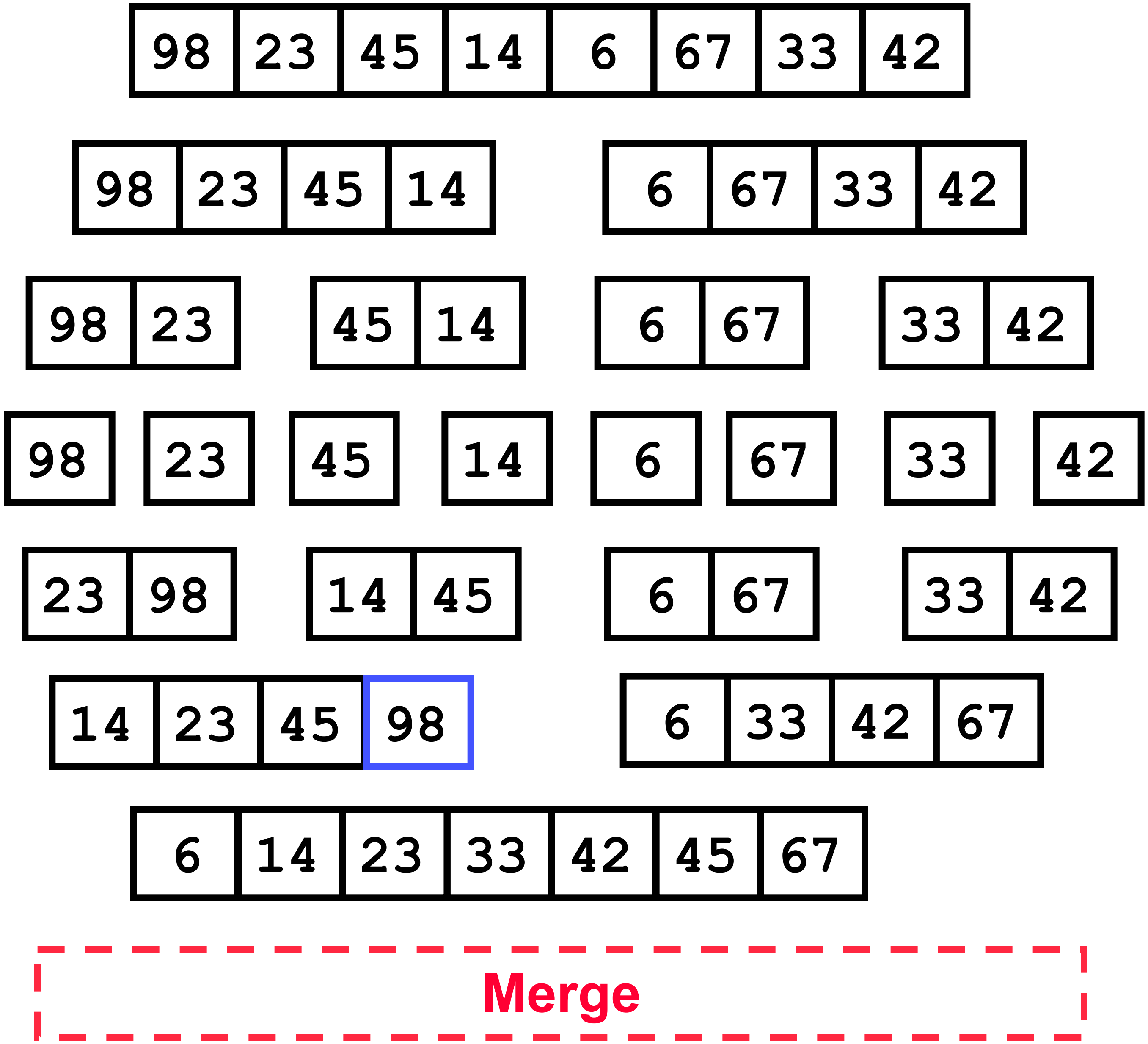
[98]   [23]   [45]   [14]   [6]   [67]   [33]   [42]

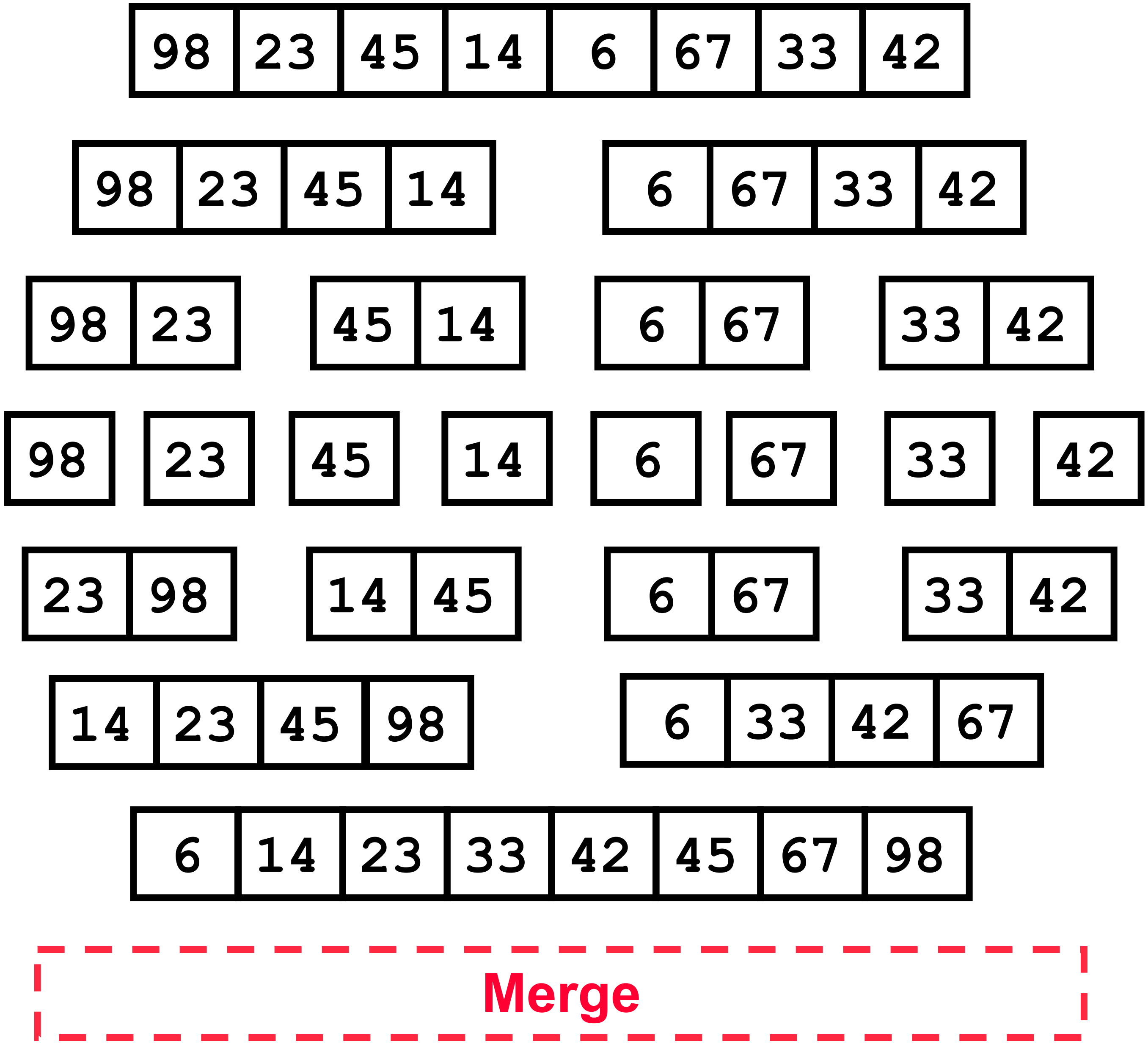
[23 | 98]   [14 | 45]   [6 | 67]   [33 | 42]

[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

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 example 2

13	6	21	18	9	4	8	20
0							7

13	6	21	18	9	4	8	20
0			3	4			7

13	6	21	18	9	4	8	20
0	1	2	3	4	5	6	7

13	6	21	18	9	4	8	20
0	1	2	3	4	5	6	7

6	13	18	21	4	9	8	20
0	1	2	3	4	5	6	7

6	13	18	21	4	8	9	20
0			3	4			7

4	6	8	9	13	18	20	21
0							7

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

Recurrence for Merge Sort

- Recall: Running time of recursive algorithms are described using recurrences
  - A recurrence is an equation that describes a function in terms of its value on smaller inputs

- Merge Sort:

$$T(n) = \Theta(1), n = 1$$

$$T(n) = 2T(n) + n, n > 1$$

Solving Recurrences

- Methods:
  - Repeated substitution method
  - Substitution method
  - Recursion trees
  - Telescoping
  - Master method

Recall Master Theorem

- **Master Theorem**:
  A recurrence written in the form
  T(n) = a * T(n / b) + f(n)

  (where f(n) is a function that is O(n^k) for some power k)
  has a solution such that

$$T(n) = \begin{cases} O(n^{\log_b a}), & a > b^k \\ O(n^k \log n), & a = b^k \\ O(n^k), & a < b^k \end{cases}$$

- This form of recurrence is very common for divide-and-conquer algorithms

Master Theorem

- Binary search is of the correct format:
  $T(n) = a * T(n / b) + f(n)$
  - $T(n) = 2T(n/2) + n$
  - $T(1) = n$

  - $f(n) = n = O(n) = O(n^1)$ ... therefore $k = 1$
  - $a = 2$, $b = 2$

- $2 = 2^1$, therefore:
  $T(n) = O(n^1 \log n) =$ **$O(n \log n)$**

Repeated Substitution Method

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

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

  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).

Quick sort

- **quick sort**: orders a list of values by partitioning the list around one element called a *pivot*, then sorting each partition
  - invented by British computer scientist C.A.R. Hoare in 1960

- another divide and conquer algorithm:
  - choose one element in the list to be the pivot (partition element)
  - *divide* the elements so that all elements less than the pivot are to its left and all greater are to its right
  - *conquer* by applying the quick sort algorithm (recursively) to both partitions

Quick sort, continued

- For correctness, it's okay to choose any pivot.

- For efficiency, one of following is best case, the other worst case:
  - pivot partitions the list roughly in half
  - pivot is greatest or least element in list

- Which case above is best?

- We will divide the work into two methods:
  - `quickSort` – performs the recursive algorithm
  - `partition` – rearranges the elements into two partitions

Quick sort pseudo-code

- Let S be the input set.

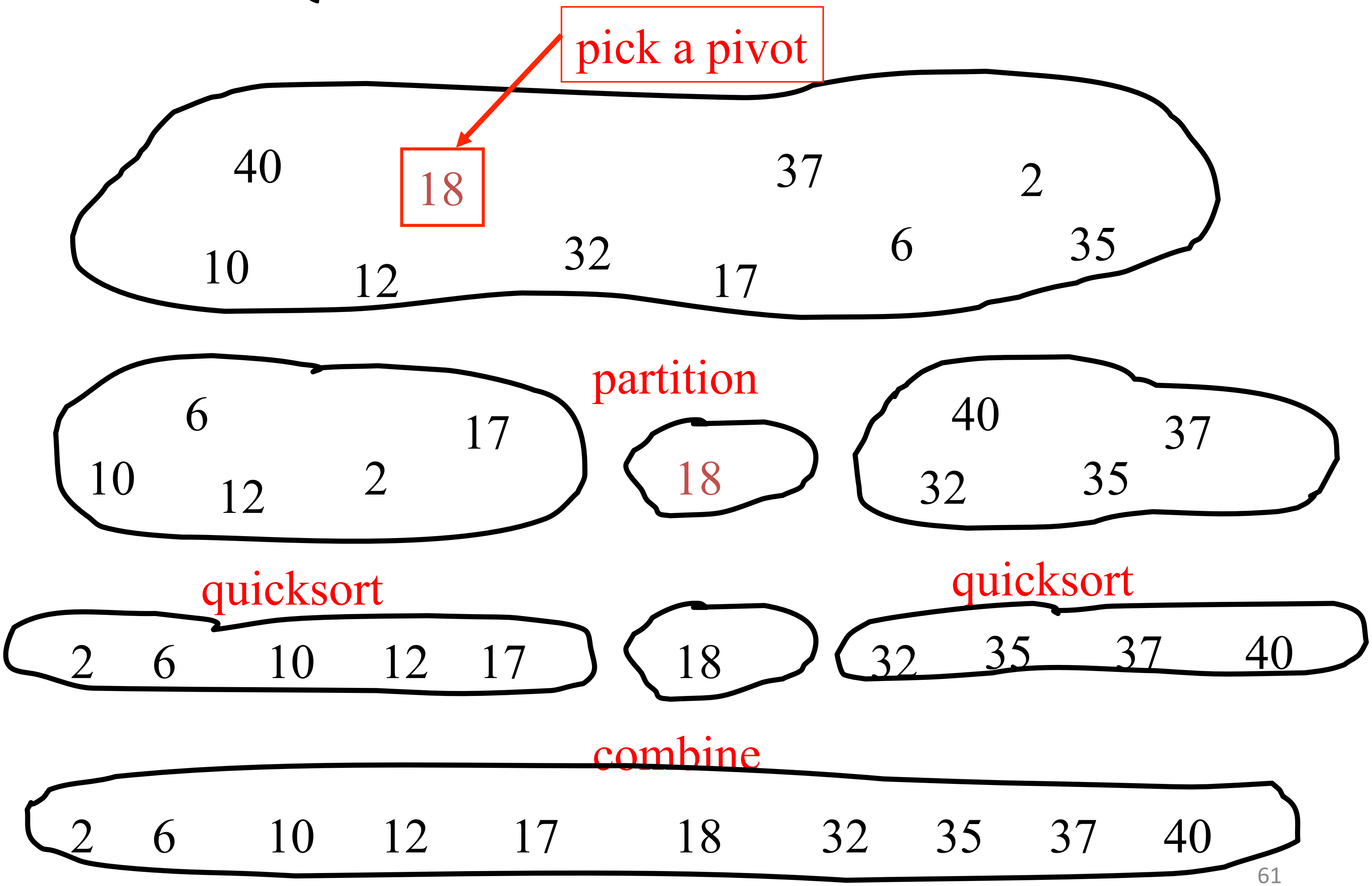
1. If $|S| = 0$ or $|S| = 1$, then return.

2. Pick an element v in S. Call v the pivot.

3. Partition $S - \{v\}$ into two disjoint groups:
   - $S_1 = \{x \in S - \{v\} \mid x \leq v\}$
   - $S_2 = \{x \in S - \{v\} \mid x \geq v\}$

4. Return { quicksort(S_1), v, quicksort(S_2) }

Quick sort illustrated



How to choose a pivot

- first element
  - bad if input is sorted or in reverse sorted order
  - bad if input is nearly sorted
  - variation: particular element (e.g. middle element)
- random element
  - even a malicious agent cannot arrange a bad input
- median of three elements
  - choose the median of the left, right, and center elements

Partitioning algorithm

The basic idea:

1. Move the pivot to the rightmost position.
2. Starting from the left, find an element ≥ pivot. Call the position i.
3. Starting from the right, find an element ≤ pivot. Call the position j.
4. Swap S[i] and S[j].

8	1	4	9	0	3	5	2	7	6

0 9

Partitioning example

8	1	4	9	0	3	5	2	7	6

0 9

"Median of three" pivot

9	17	3	12	8	7	21	1

0 7

pick pivot

1	17	3	12	8	7	21	9

0 i j 7

1	7	3	12	8	17	21	9

0 i j 7

1	7	3	8	12	17	21	9

0 j i 7

swap S[i]
with S[right]

1	7	3	8	9	17	21	12

0 7

Quick sort code

```
public static void quickSort(int[] a) {
    quickSort(a, 0, a.length - 1);
}

private static void quickSort(int[] a, int min, int max) {
    if (min >= max) {  // base case; no need to sort
        return;
    }

    // choose pivot -- we'll use the first element (might be bad!)
    int pivot = a[min];
    swap(a, min, max);  // move pivot to end

    // partition the two sides of the array
    int middle = partition(a, min, max - 1, pivot);

    // restore the pivot to its proper location
    swap(a, middle, max);

    // recursively sort the left and right partitions
    quickSort(a, min, middle - 1);
    quickSort(a, middle + 1, max);
}
```

Quick sort code, cont'd.

```
// partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    i--;  j++;  // kludge because the loops pre-increment
    while (true) {
        // move index markers i,j toward center
        // until we find a pair of mis-partitioned elements
        do { i++; } while (i < j && a[i] < pivot);
        do { j--; } while (i < j && a[j] > pivot);

        if (i >= j) {
            break;
        } else {
            swap(a, i, j);
        }
    }

    return i;
}
```

Quick sort code, cont'd.

```
// partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    i--;  j++;  // kludge because the loops pre-increment
    while (true) {
        // move index markers i,j toward center
        // until we find a pair of mis-partitioned elements
        do { i++; } while (i < j && a[i] < pivot);
        do { j--; } while (i < j && a[j] > pivot);

        if (i >= j) {
            break;
        } else {
            swap(a, i, j);
        }
    }

    return i;
}
```

Quick sort code, cont'd.

```
// partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    i--;  j++;  // kludge because the loops pre-increment
    while (true) {
        // move index markers i,j toward center
        // until we find a pair of mis-partitioned elements
        do { i++; } while (i < j && a[i] < pivot);
        do { j--; } while (i < j && a[j] > pivot);

        if (i < j) {
            swap(a, i, j);
        } else {
            break;
        }
    }

    return i;
}
```

Quick sort code, cont'd.

```
// partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    i--;  j++;  // kludge because the loops pre-increment
    while (true) {
        // move index markers i,j toward center
        // until we find a pair of mis-partitioned elements
        do { i++; } while (i < j && a[i] < pivot);
        do { j--; } while (i < j && a[j] > pivot);

        if (i < j) {
            swap(a, i, j);
        } else {
            break;
        }
    }

    return i;
}
```

Quick sort runtime

- Worst case: pivot is the smallest (or largest) element all the time (recurrence solution technique: telescoping)

  $T(n) = T(n\text{-}1) + cn$

  $T(n\text{-}1) = T(n\text{-}2) + c(n\text{-}1)$

  $T(n\text{-}2) = T(n\text{-}3) + c(n\text{-}2)$

  …

  $T(2) = T(1) + 2c$

$$T(N) = T(1) + c \sum_{i=2}^{N} i = O(N^2)$$

- Best case: pivot is the median (recurrence solution technique: Master's Theorem)

  $T(n) = 2\,T(n/2) + cn$

  $T(n) = cn \log n + n = O(n \log n)$

Quick sort runtime summary

- O(n log n) on average.
- O(n^2) worst case.

	comparisons
merge	O(n log n)
quick	average: O(n log n) worst: O(n^2)

Sorting practice problem

- Consider the following array of int values.

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

(f) Write the contents of the array after all the partitioning of quick sort has finished (before any recursive calls).

Assume that the median of three elements (first, middle, and last) is chosen as the pivot.

Sorting practice problem

- Consider the following array:
  `[ 7, 17, 22, -1,  9,  6, 11, 35, -3]`
- Each of the following is a view of a sort-in-progress on the elements. Which sort is which?
  - (If the algorithm is a multiple-loop algorithm, the array is shown after a few of these loops have completed. If the algorithm is recursive, the array is shown after the recursive calls have finished on each sub-part of the array.)
  - Assume that the quick sort algorithm chooses the ***first*** element as its pivot at each pass.

(a) `[-3, -1,  6, 17,  9, 22, 11, 35,  7]`
(b) `[-1,  7, 17, 22, -3,  6,  9, 11, 35]`
(c) `[ 9, 22, 17, -1, -3,  7,  6, 35, 11]`
(d) `[-1,  7,  6,  9, 11, -3, 17, 22, 35]`
(e) `[-3,  6, -1,  7,  9, 17, 11, 35, 22]`
(f) `[-1,  7, 17, 22,  9,  6, 11, 35, -3]`

Sorting practice problem

- For the following questions, indicate which of the six sorting algorithms will <u>successfully</u> sort the elements in the least amount of time.
  - The algorithm chosen should be the one that completes fastest, *without crashing*.
  - Assume that the quick sort algorithm chooses the ***first*** element as its pivot at each pass.
  - Assume stack overflow occurs on 5000+ stacked method calls.
  - (a) array size 2000, random order
  - (b) array size 500000, ascending order
  - (c) array size 100000, descending order
    - special constraint: no extra memory may be allocated! (O(1) storage)
  - (d) array size 1000000, random order
  - (e) array size 10000, ascending order
    - special constraint: no extra memory may be allocated! (O(1) storage)

Lower Bound for Comparison Sorting

- Theorem: Any algorithm that sorts using only comparisons between elements requires $\Omega(n \log n)$ comparisons.
  - Intuition
    - n! permutations that a sorting algorithm can output
    - each new comparison between any elements a and b cuts down the number of possible valid orderings by at most a factor of 2 (either all orderings where a > b or orderings where b > a)
    - to know which output to produce, the algorithm must make at least $\log_2(n!)$ comparisons before
    - $\log_2(n!) = \Omega(n \log n)$

O(n) Specialized Sorting

Bucket sort

- The bucket sort makes assumptions about the data being sorted
  - Namely our data falls in a discrete range
- Allows us to achieve better than $\Theta(n \log n)$ run times

Bucket Sort: Example

- Sort a large number of local phone numbers (e.g., all 2,000,000+ phone numbers in the 206 area code)

- Idea:
  - create an array with 10 000 000 bits (i.e. `BitSet`)
  - set each bit to 0 (indicating false)
  - for each phone number, set the bit indexed by the phone number to 1 (true)
  - once each phone number has been checked, walk through the array and for each bit which is 1, record that number

- The runtime is **O**(*n*) (one pass through the data to set bits; one pass through the array to extract the phone numbers with bits set)

Bucket Sort

- Uses very little memory
- Each entry in the bit array is a *bucket*
- We fill each bucket as appropriate

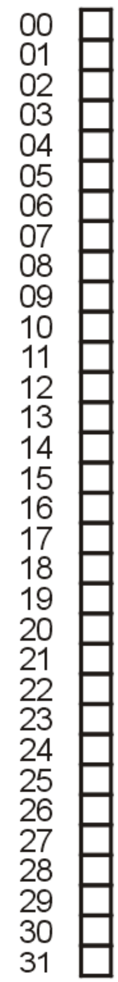
Example

- Consider sorting the following set of unique integers in the range 0, ..., 31:

20	1	31	8	29	28	11	14	6	16	15
27	10	4	23	7	19	18	0	26	12	22

- Create an bit array with 32 buckets

- This requires 4 bytes



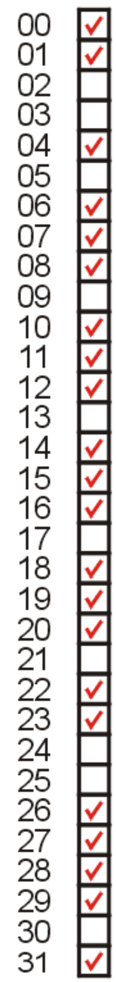
Example (cont.)

- For each number, set the bit of the corresponding bucket to 1

- Now, traverse the list and record those numbers for which the bit is 1 (true):

0 1 4 6 7 8 10 11 12 14 15

16 18 19 20 22 23 26 27 28 29 31



Bucket Sort

- What if there are repetitions in the data?
  - A bit array is now insufficient.

- Two options, each bucket is either:
  - a counter, or
  - a linked list

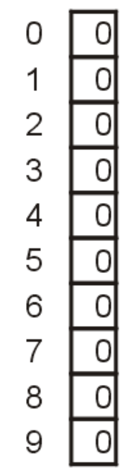
- The first is better if objects in the bin are the same

Example 2

- Sort the following digits:

```
0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2 1 3
```

- Start with an array of 10 counters.
  - Each counter initially set to zero:



Example 2 (cont.)

- Moving through the first 10 digits

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2 1 3

increment the corresponding buckets

0	1
1	0
2	2
3	3
4	0
5	2
6	0
7	1
8	1
9	0

Example 2 (cont.)

- Now read off the number of each occurrence:

0 0 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 5 5 5 5 5 5 5 7 8 8 8 9 9

- For example:
  - there are seven 2s
  - there are two 4s

0	2
1	3
2	7
3	10
4	2
5	7
6	0
7	1
8	3
9	2

Run-time Summary

- The following table summarizes the run-times of bucket sort

Case	Run Time	Comments
Worst	$\Theta(n + m)$	No worst case
Average	$\Theta(n + m)$	
Best	$\Theta(n + m)$	No best case

External Sorting

Simple External Merge Sort

- Divide and conquer: divide the file into smaller, sorted subfiles (called runs) and merge runs

- Initialize:
  - Load chunk of data from file into RAM
  - Sort internally
  - Write sorted data (run) back to disk (in separate files)
- While we still have runs to sort:
  - Merge runs from previous pass into runs of twice the size (think merge() method from mergesort)
  - Repeat until you only have one run