Sorting Algorithms

Sorting Algorithms

Song Name	Time	Track #	Artist	Album
Letters from the Wasteland	4:29	1 of 10	The Wallflowers	Breach
When You're On Top	3:54	1 of 13	The Wallflowers	Red Letter Days
Hand Me Down	3:35	2 of 10	The Wallflowers	Breach
How Good It Can Get	4:11	2 of 13	The Wallflowers	Red Letter Days
Sleepwalker	3:31	3 of 10	The Wallflowers	Breach
Closer To You	3:17	3 of 13	The Wallflowers	Red Letter Days
I've Been Delivered	5:01	4 of 10	The Wallflowers	Breach
Everybody Out Of The Water	3:42	4 of 13	The Wallflowers	Red Letter Days
Witness	3:34	5 of 10	The Wallflowers	Breach
Three Ways	4:19	5 of 13	The Wallflowers	Red Letter Days
Some Flowers Bloom Dead	4:43	6 of 10	The Wallflowers	Breach
Too Late to Quit	3:54	6 of 13	The Wallflowers	Red Letter Days
Mourning Train	4:04	7 of 10	The Wallflowers	Breach
If You Never Got Sick	3:44	7 of 13	The Wallflowers	Red Letter Days
Up from Under	3:38	8 of 10	The Wallflowers	Breach
Health and Happiness	4:03	8 of 13	The Wallflowers	Red Letter Days
Murder 101	2:31	9 of 10	The Wallflowers	Breach
See You When I Get There	3:09	9 of 13	The Wallflowers	Red Letter Days
Birdcage	7:42	10 of 10	The Wallflowers	Breach
Feels Like Summer Again	3:48	10 of 13	The Wallflowers	Red Letter Days
Everything I Need	3:37	11 of 13	The Wallflowers	Red Letter Days
Here in Pleasantville	3:40	12 of 13	The Wallflowers	Red Letter Days
Empire in My Mind (Bonus Track)	3:31	13 of 13	The Wallflowers	Red Letter Days

Bubble Sort

Bubble Sort

Function : BUBBLE_SORT (K,N)

```
1.  [Initialize]
        LAST ← N   (entire list assumed unsorted at this point)
2.  [Loop on pass index]
        Repeat thru step 5 for PASS = 1, 2, ..., N - 1
3.  [Initialize exchanges counter for this pass]
        EXCHS ← 0
4.  [Perform pairwise comparisons on unsorted elements]
        Repeat for I = 1, 2, ..., LAST - 1
            If K[I] > K[I + 1]
            then   K[I] ⟷ K[I + 1]
                   EXCHS ← EXCHS + 1
5.  [Were any exchanges made on this pass ?]
        If EXCHS = 0
        then   Return   (mission accomplished; return early)
        else   LAST ← LAST - 1   (reduce size of unsorted list)
6.  [Finished]
        Return   (maximum number of passes required)
```

Bubble Sort

Function : BUBBLESORT (A)

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

Bubble Sort

Function : BUBBLESORT (A)

	Unsorted	*Pass Number (i)*					*Sorted*
j	K_j	1	2	3	4	5	6
1	42	23	23	11	11	11	11
2	23	42	11	23	23	23	23
3	74	11	42	42	42	36	36
4	11	65	58	58	36	42	42
5	65	58	65	36	58	58	58
6	58	74	36	65	65	65	65
7	94	36	74	74	74	74	74
8	36	94	87	87	87	87	87
9	99	87	94	94	94	94	94
10	87	99	99	99	99	99	99

FIGURE Trace of a bubble sort.

Insertion Sort

Insertion Sort

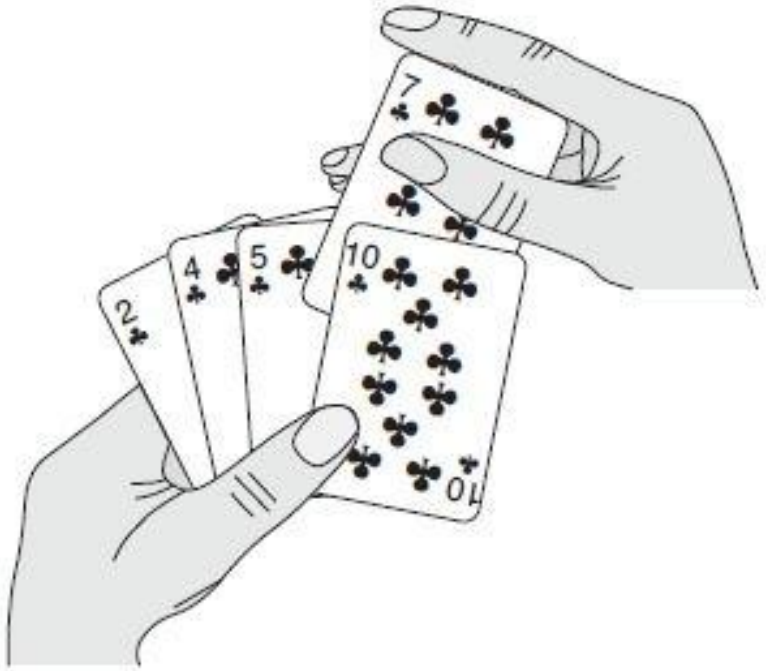


Figure Sorting a hand of cards using insertion sort.

Insertion Sort

Function : INSERTION-SORT (A)

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Selection Sort

Selection Sort

- Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in A[1]. Then find the second smallest element of A, and exchange it with A[2]. Continue in this manner for the first n-1 elements of A. Write pseudo-code for this algorithm, which is known as selection sort.

Merge Sort

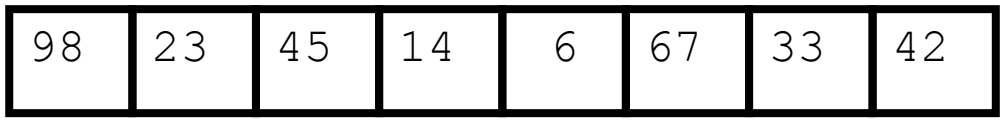
Merge Sort

```
MergeSort(low,high)
{
   if(low<high)
   {
         mid=(low+high)/2
         MergeSort(low,mid)
         MergeSort(mid+1,high)
         Merge(low,mid,high)
   }
}
```

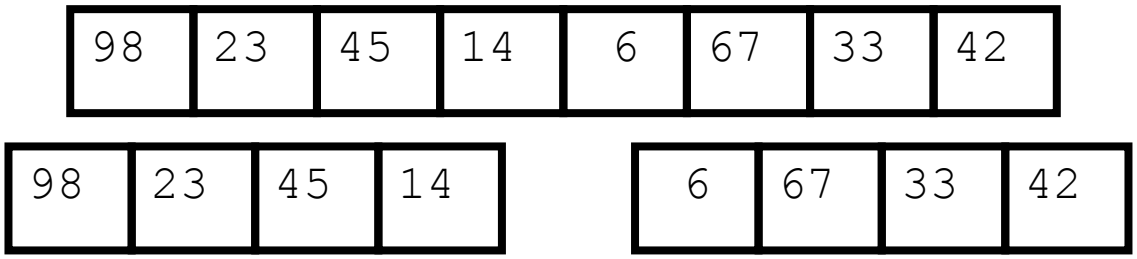
Merge Sort

```
Merge(low,mid,high){
    h=low; i=low; j=mid+1;
    while ( (h<=mid) and (j<=high) )
    {       if(a[h]<=a[j])
            {           b[i]=a[h]
                        h=h+1
            }
            else
            {           b[i]=a[j]
                        j=j+1
            }
            i=i+1
    }//End of while
    If(h>mid)
    {       for k=j to high
            {           b[i]=a[k]
                        i=i+1
            }
    }
    else
    {       for k=h to mid
            {           b[i]=a[k]
                        i=i+1
            }
    }
    for k=low to high
            a[k]=b[k]
}//End of Merge
```

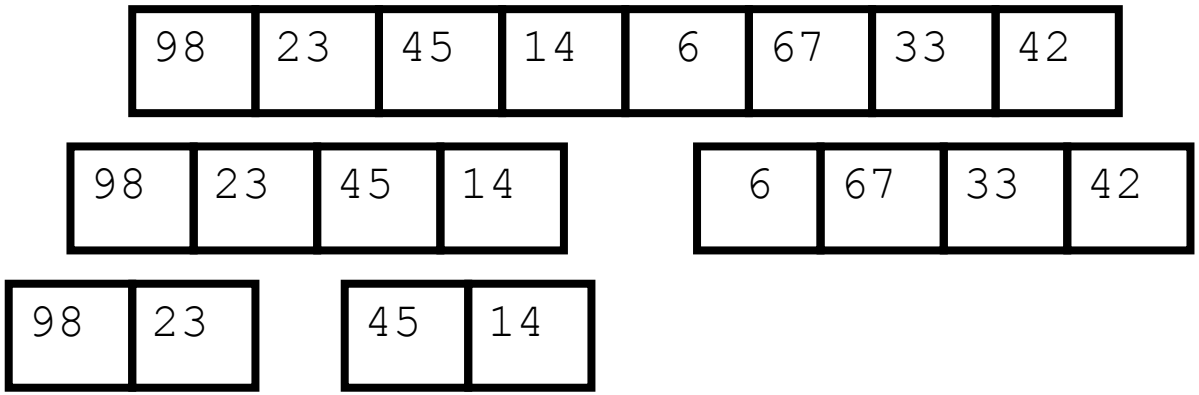
Merge Sort (Example)



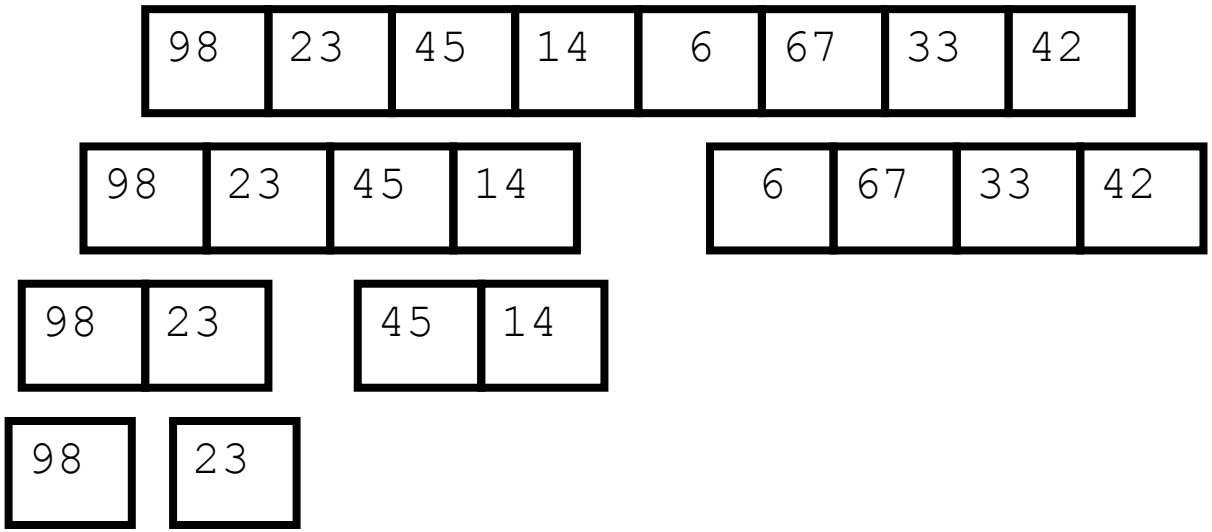
Merge Sort (Example)



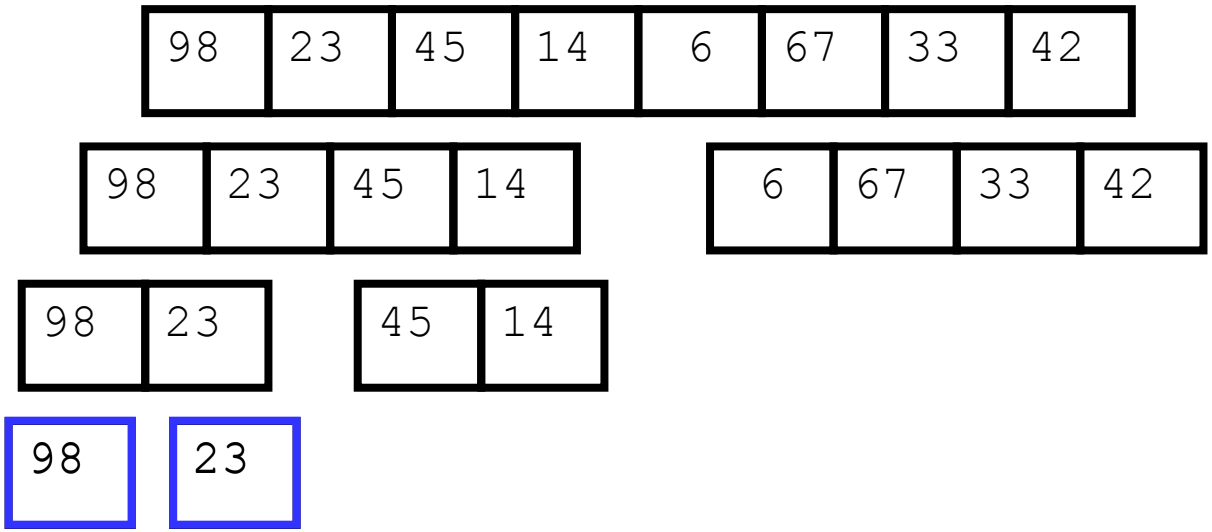
Merge Sort (Example)



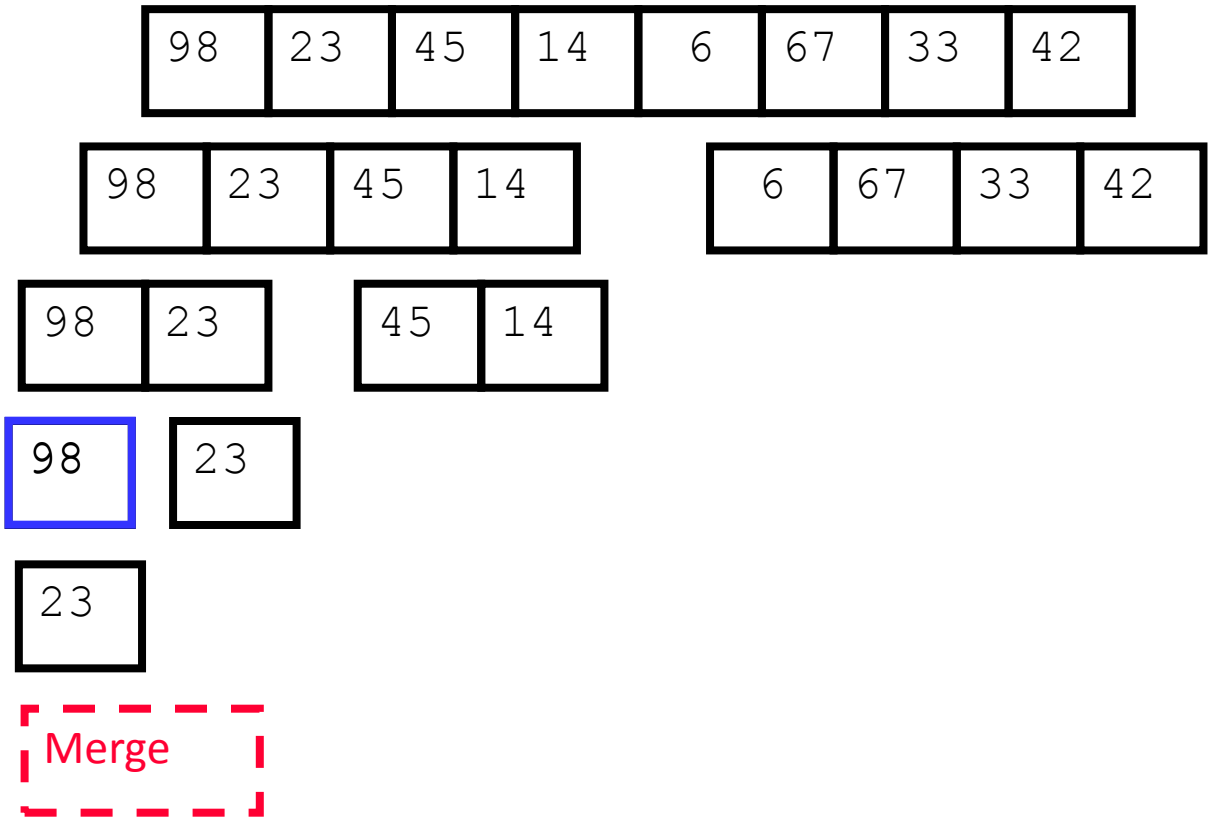
Merge Sort (Example)



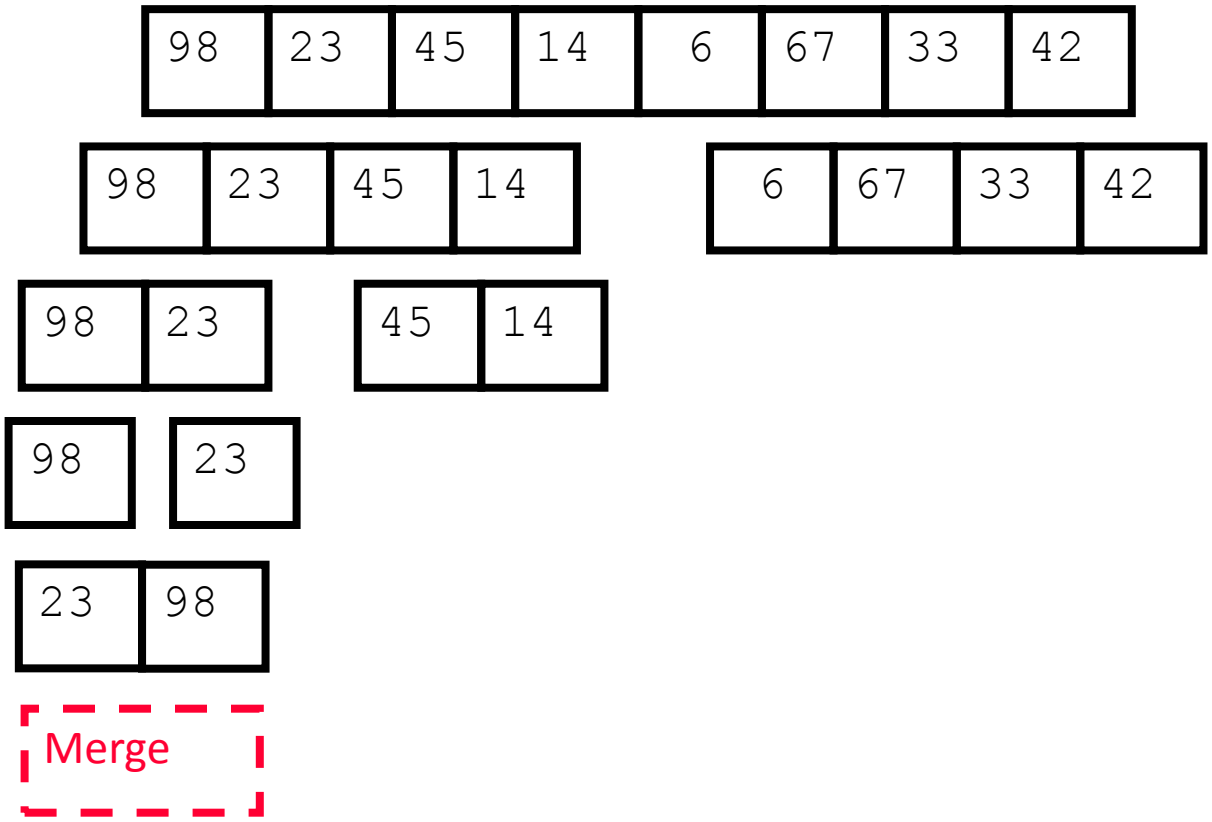
Merge Sort (Example)



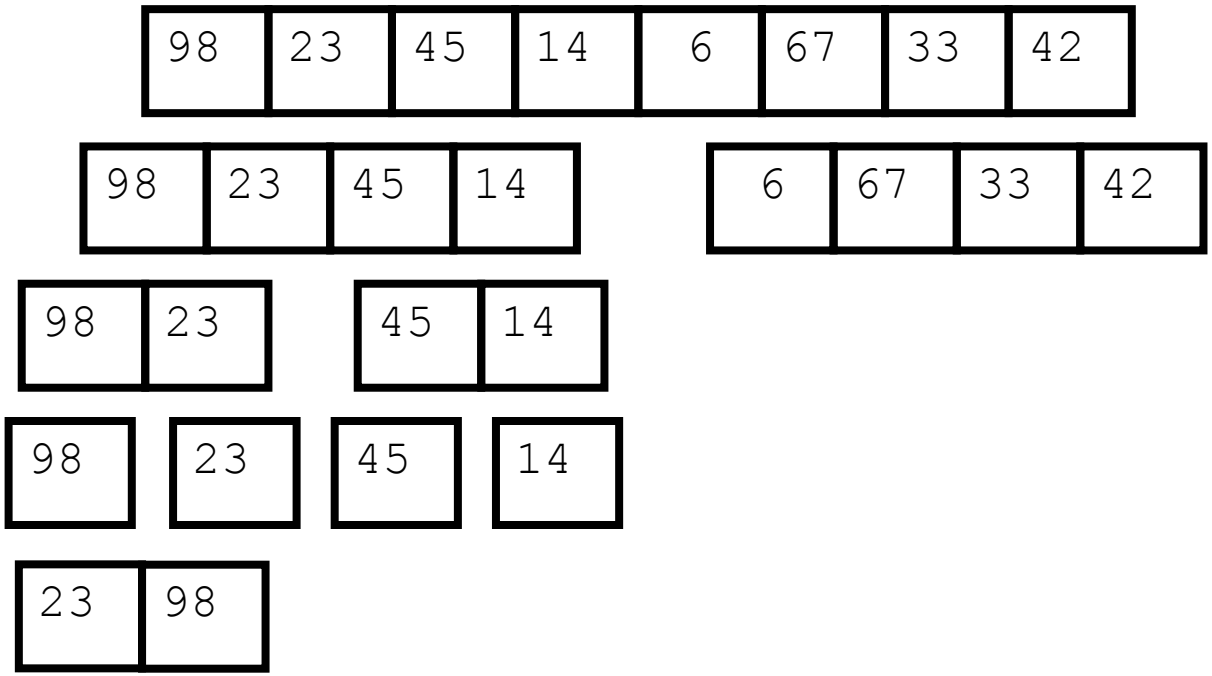
Merge Sort (Example)



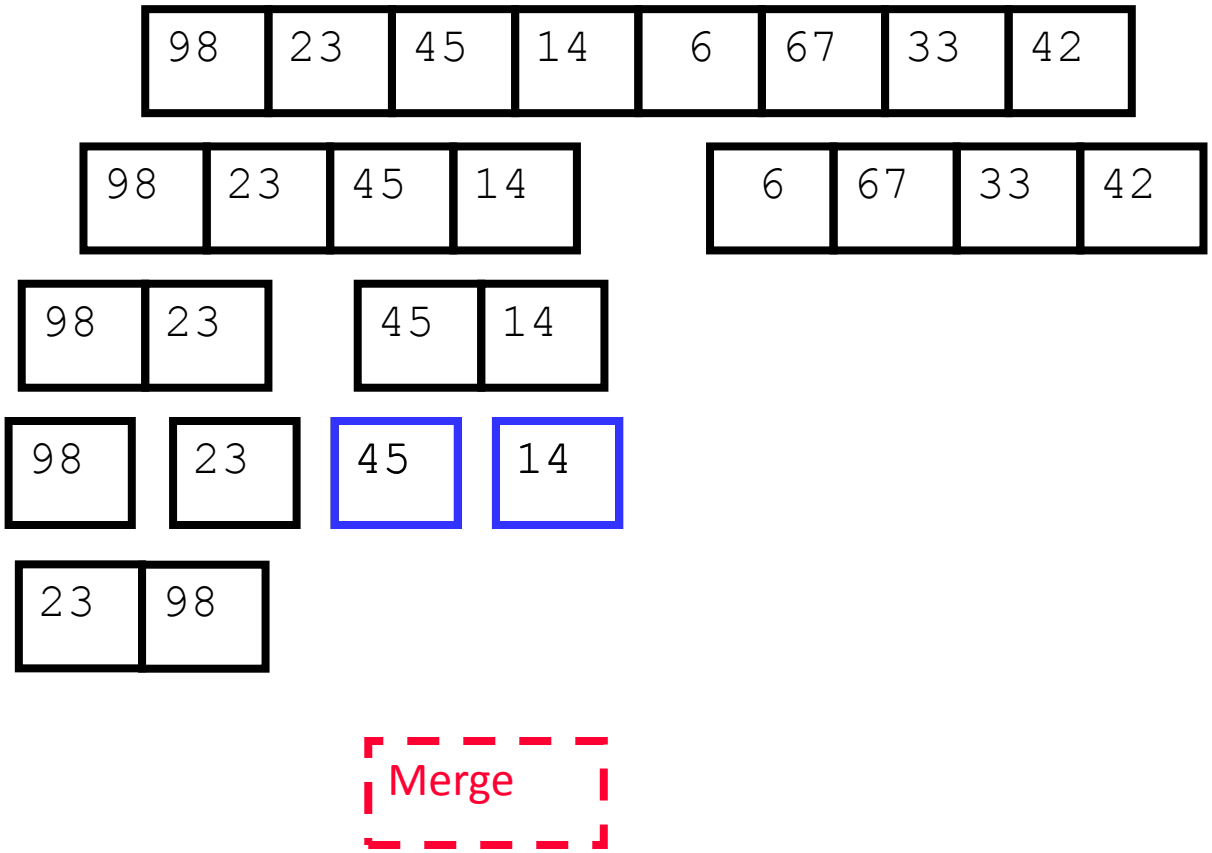
Merge Sort (Example)



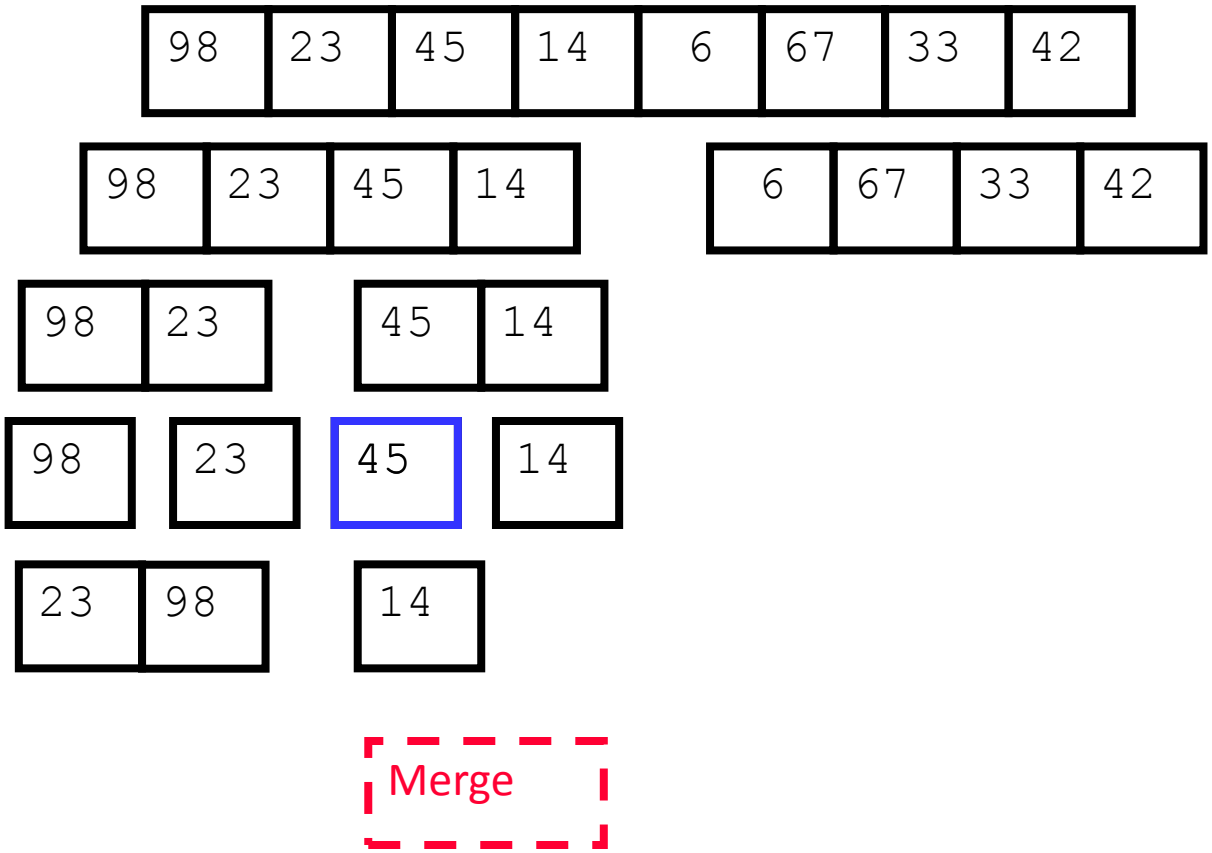
Merge Sort (Example)



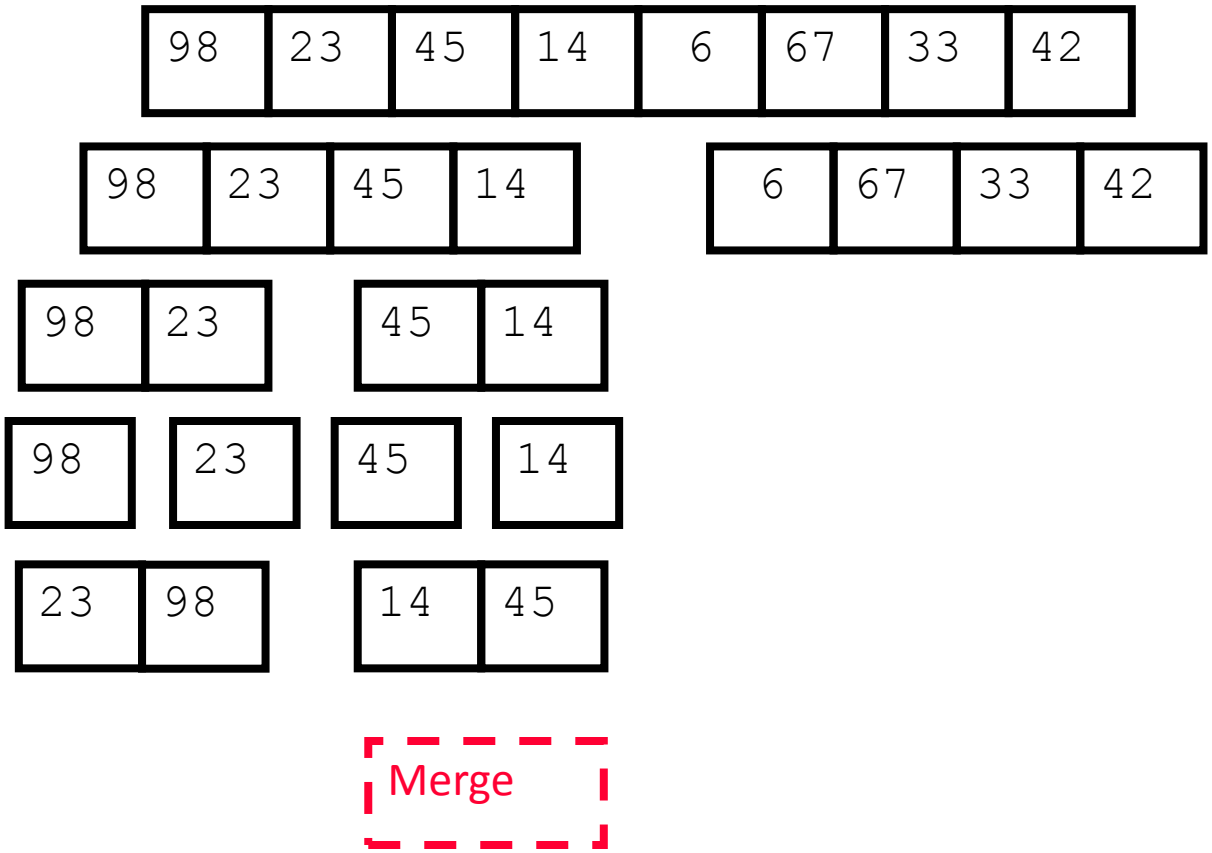
Merge Sort (Example)



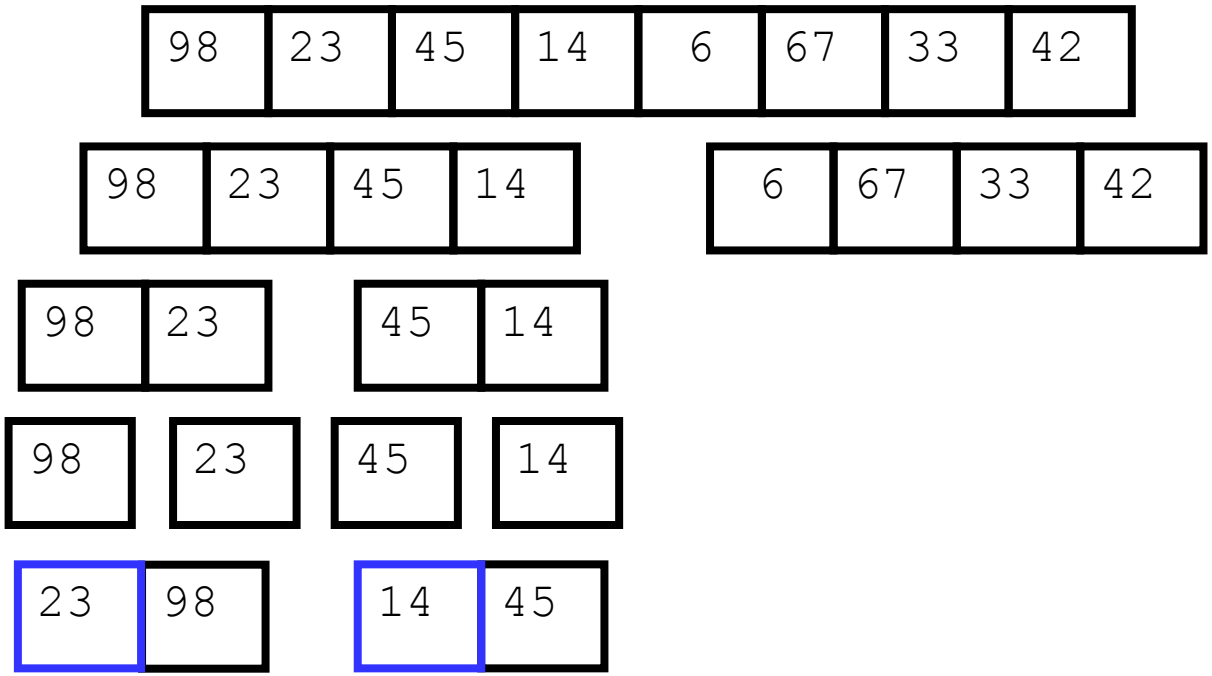
Merge Sort (Example)



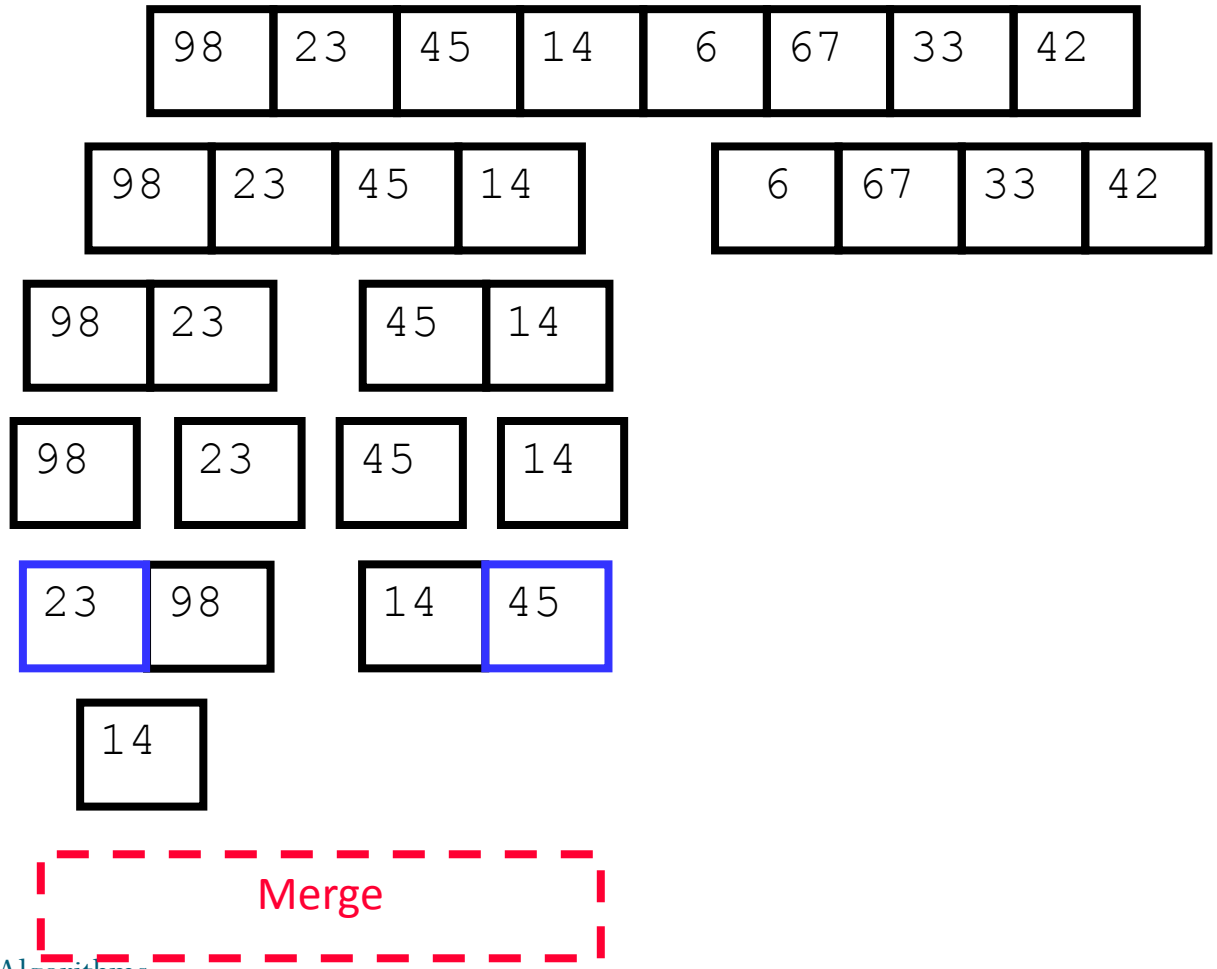
Merge Sort (Example)



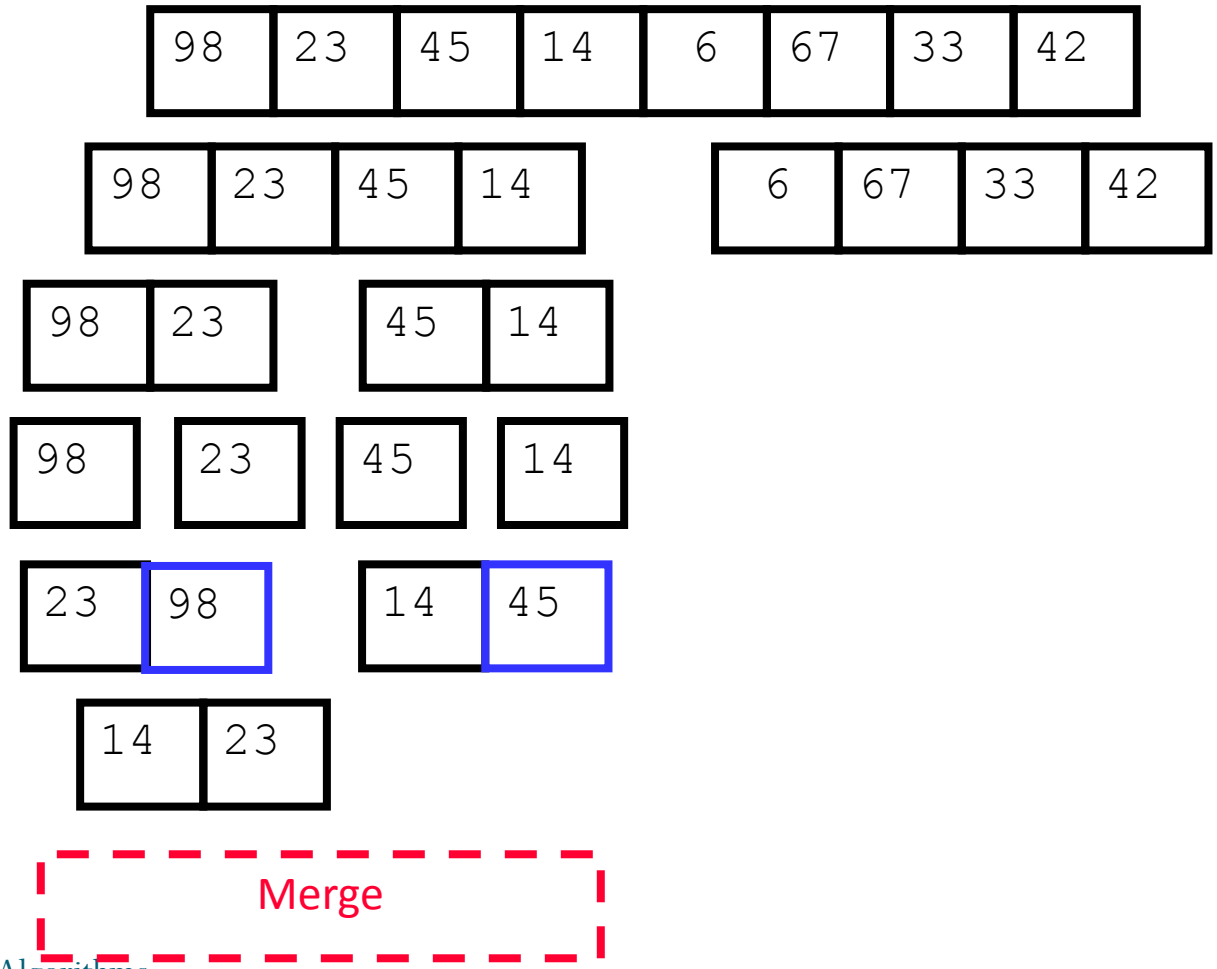
Merge Sort (Example)



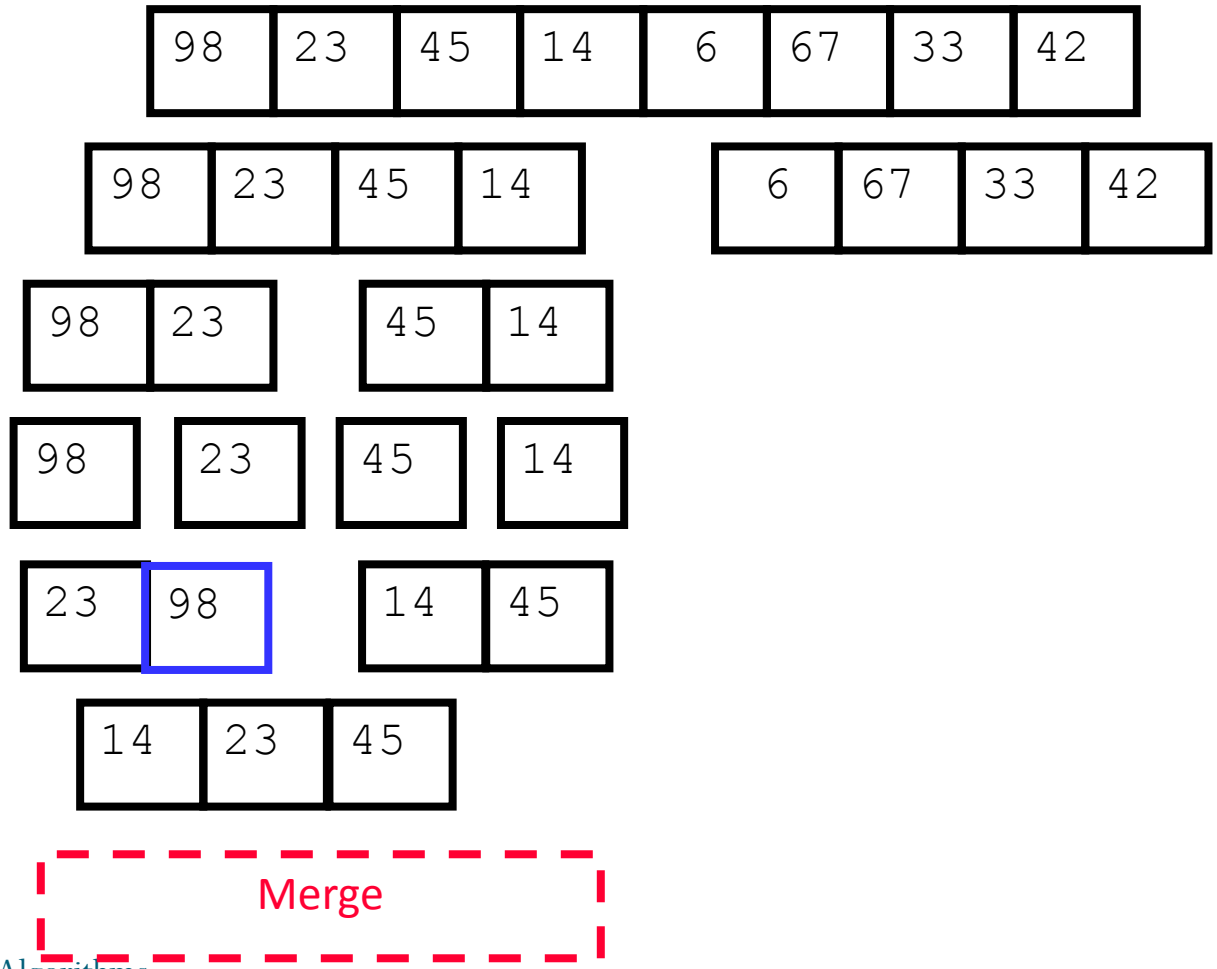
Merge Sort (Example)



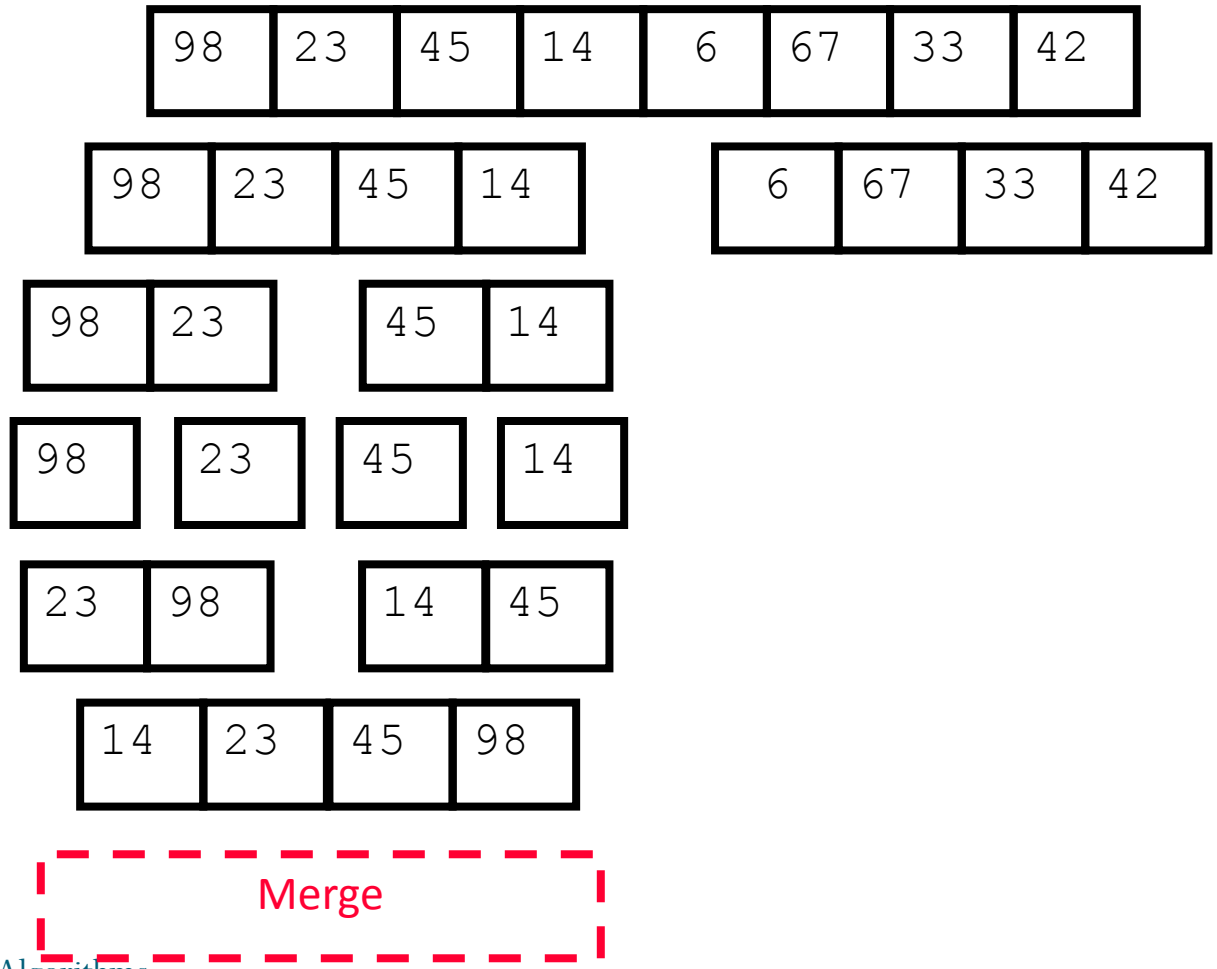
Merge Sort (Example)



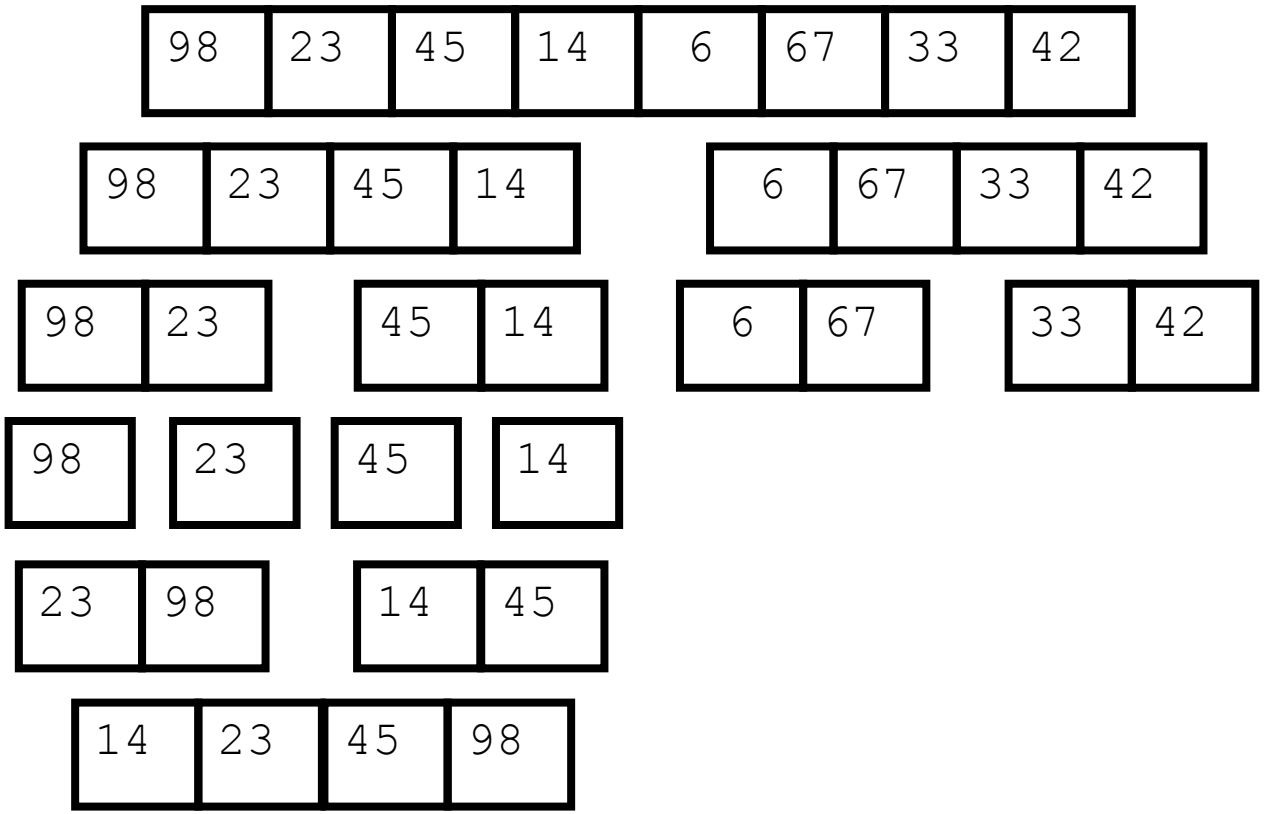
Merge Sort (Example)



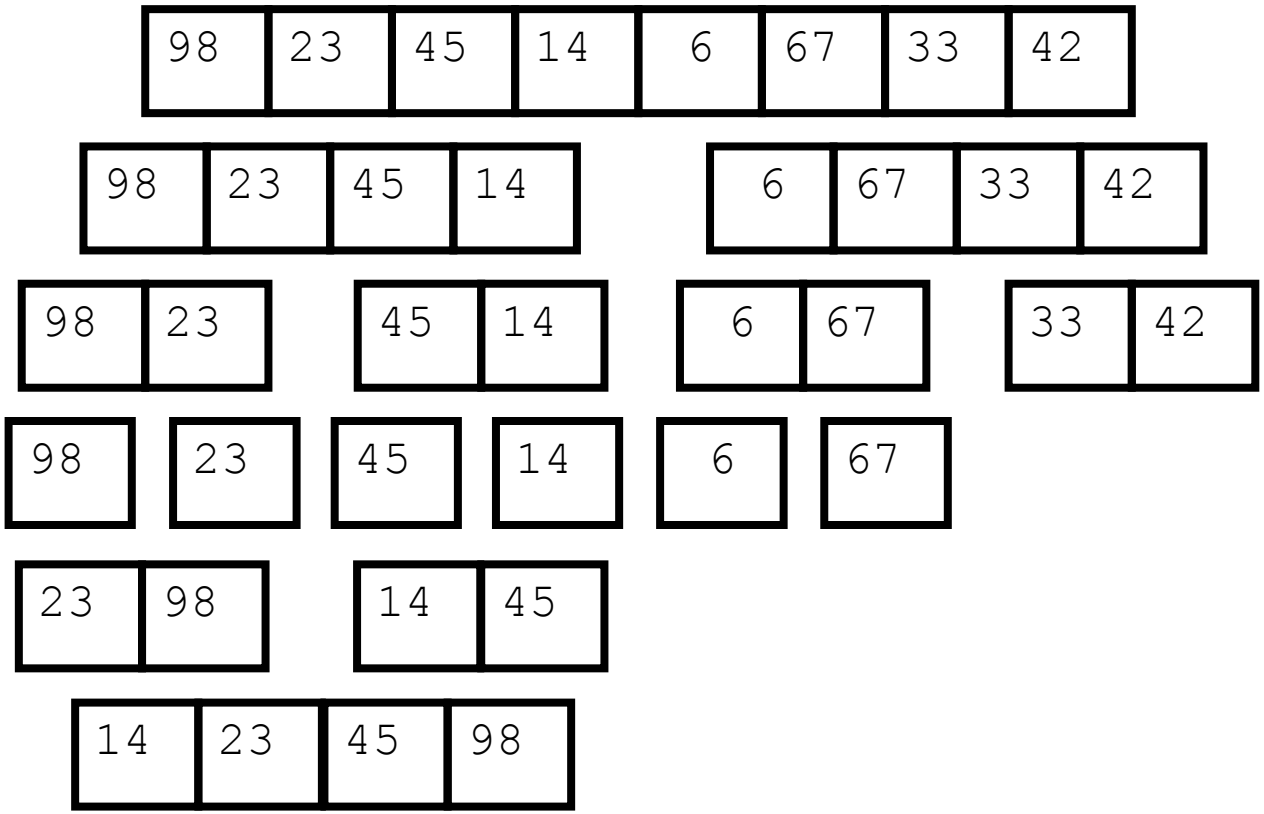
Merge Sort (Example)



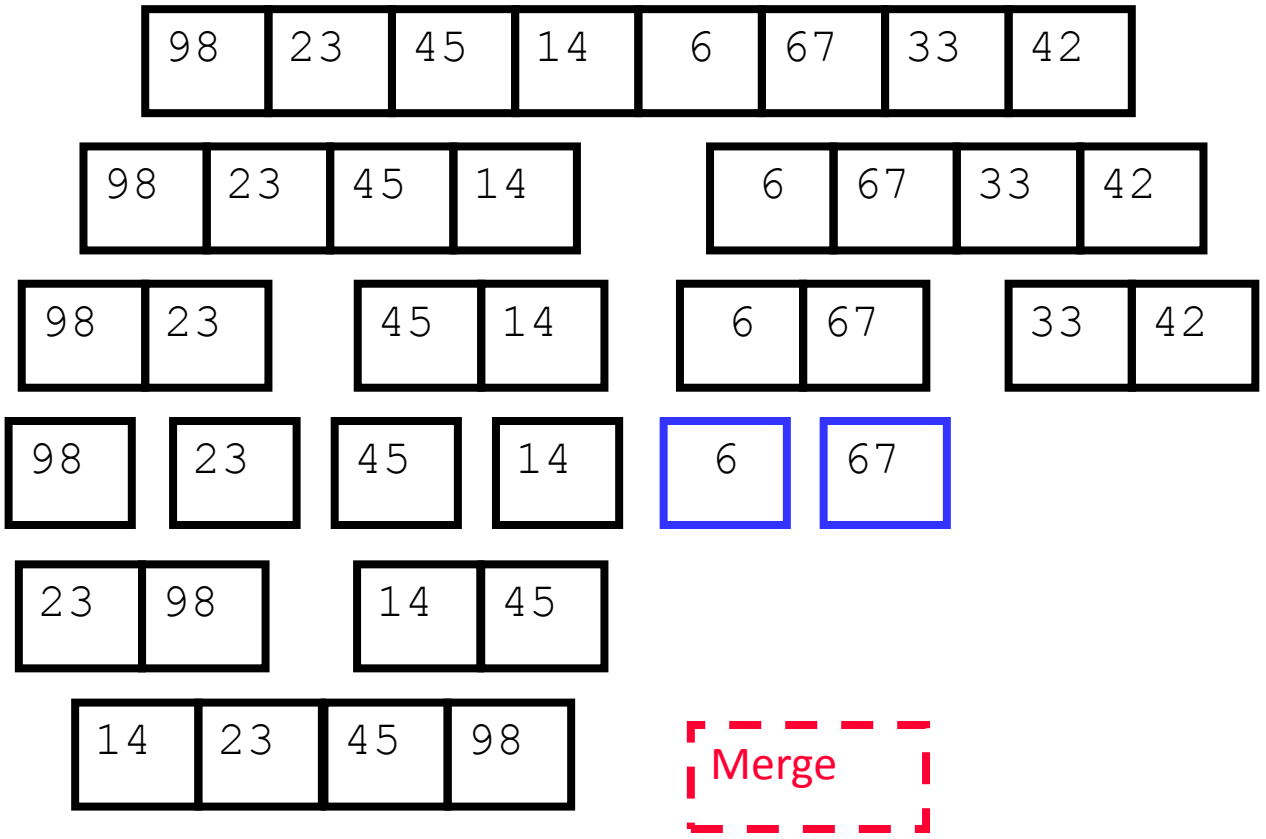
Merge Sort (Example)



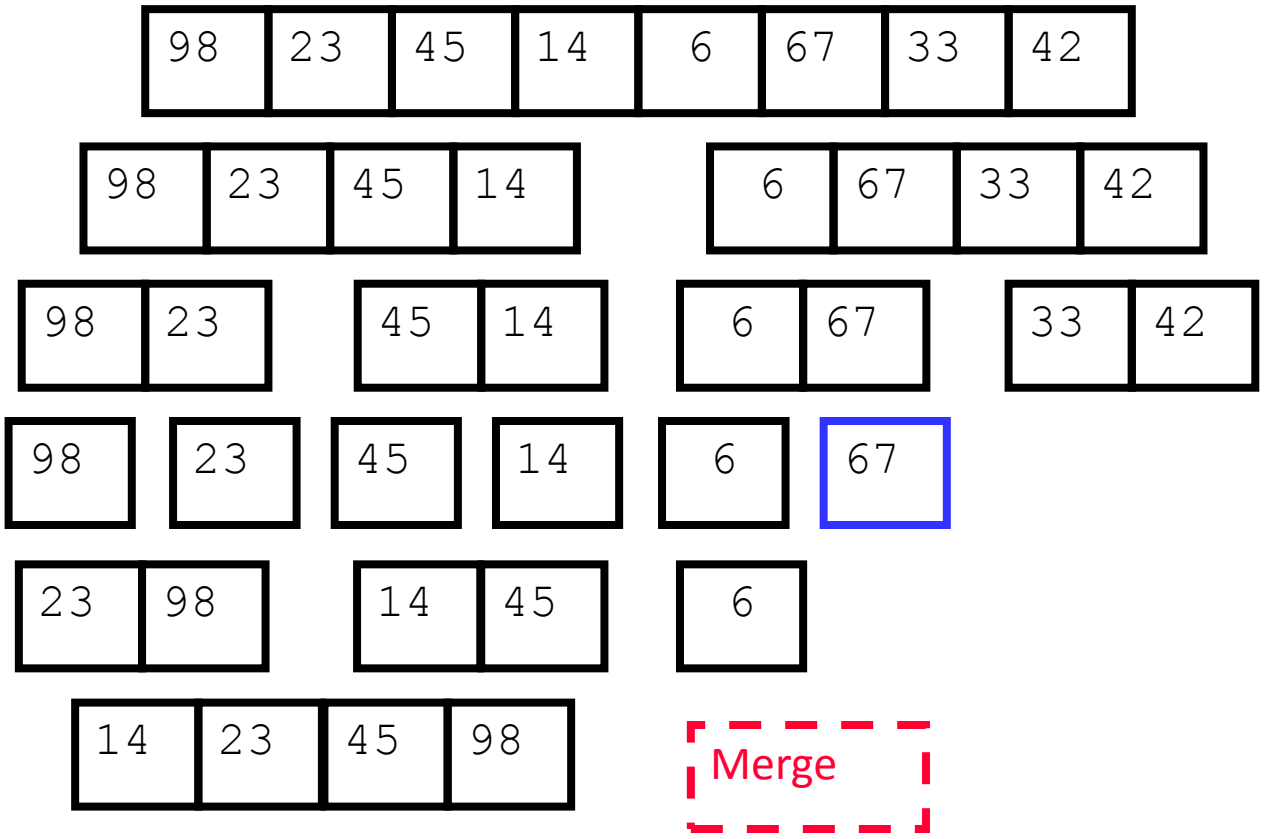
Merge Sort (Example)



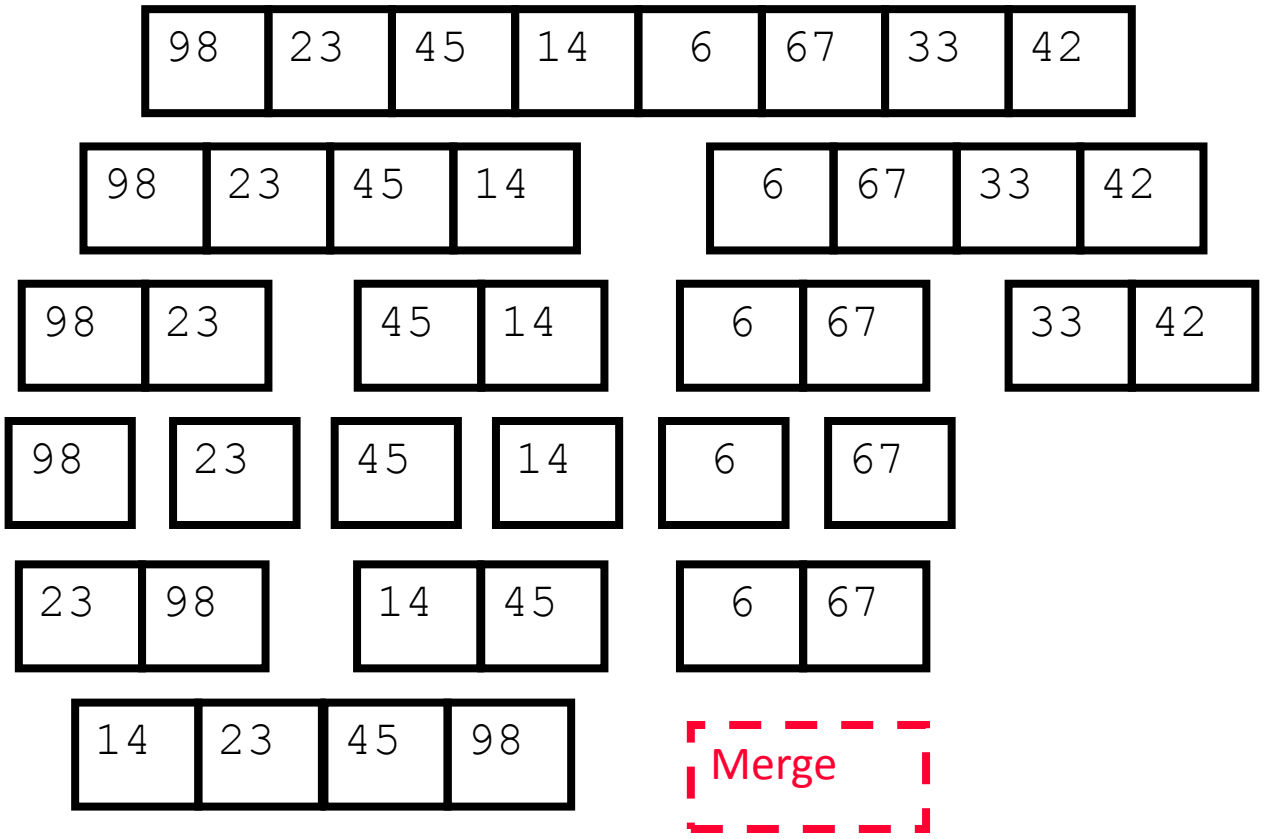
Merge Sort (Example)



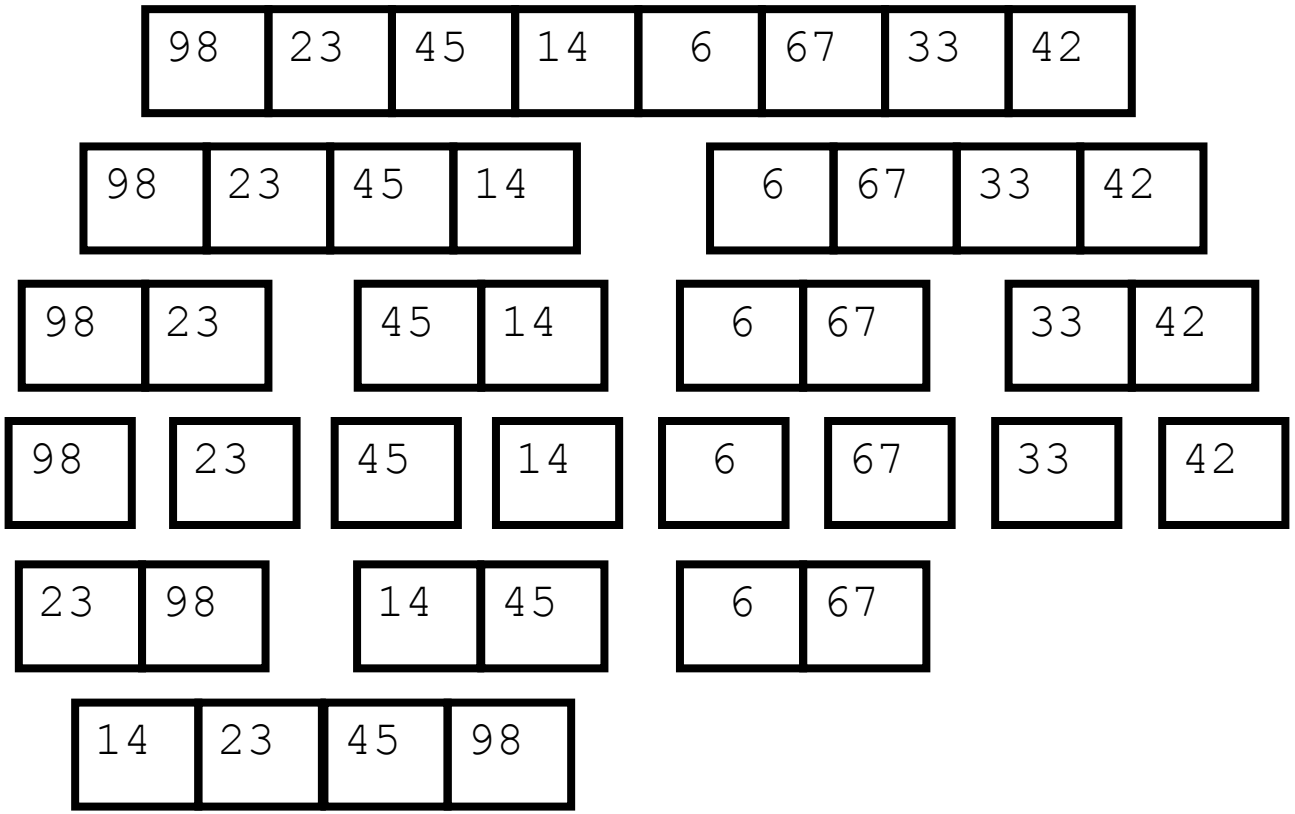
Merge Sort (Example)



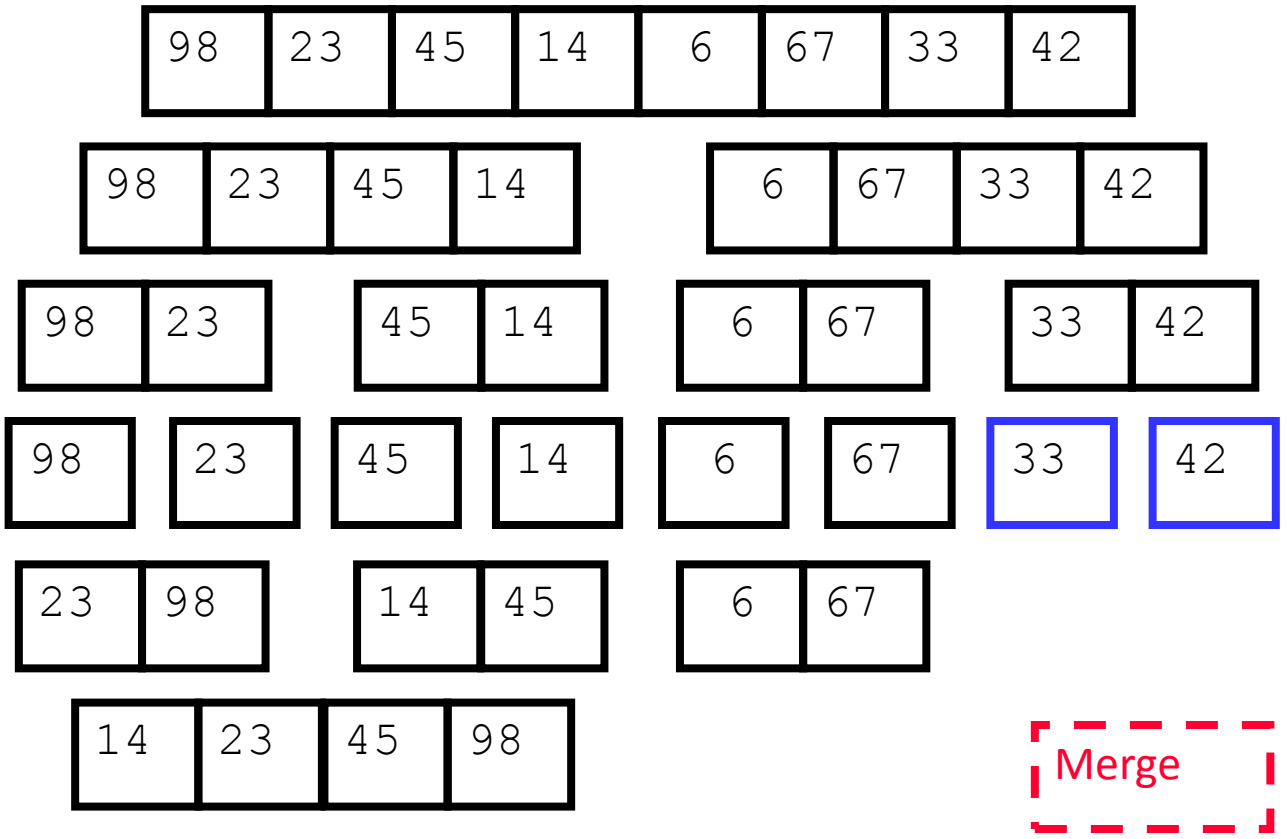
Merge Sort (Example)



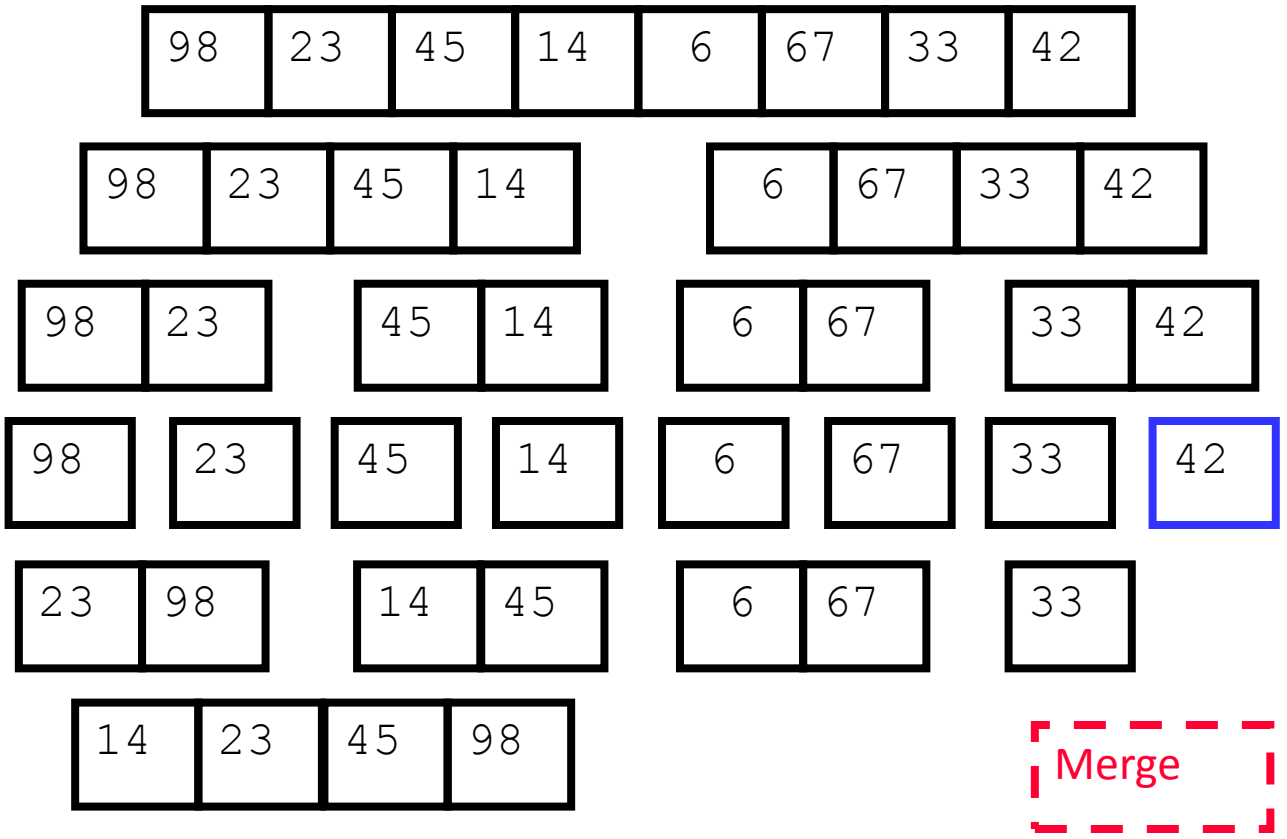
Merge Sort (Example)



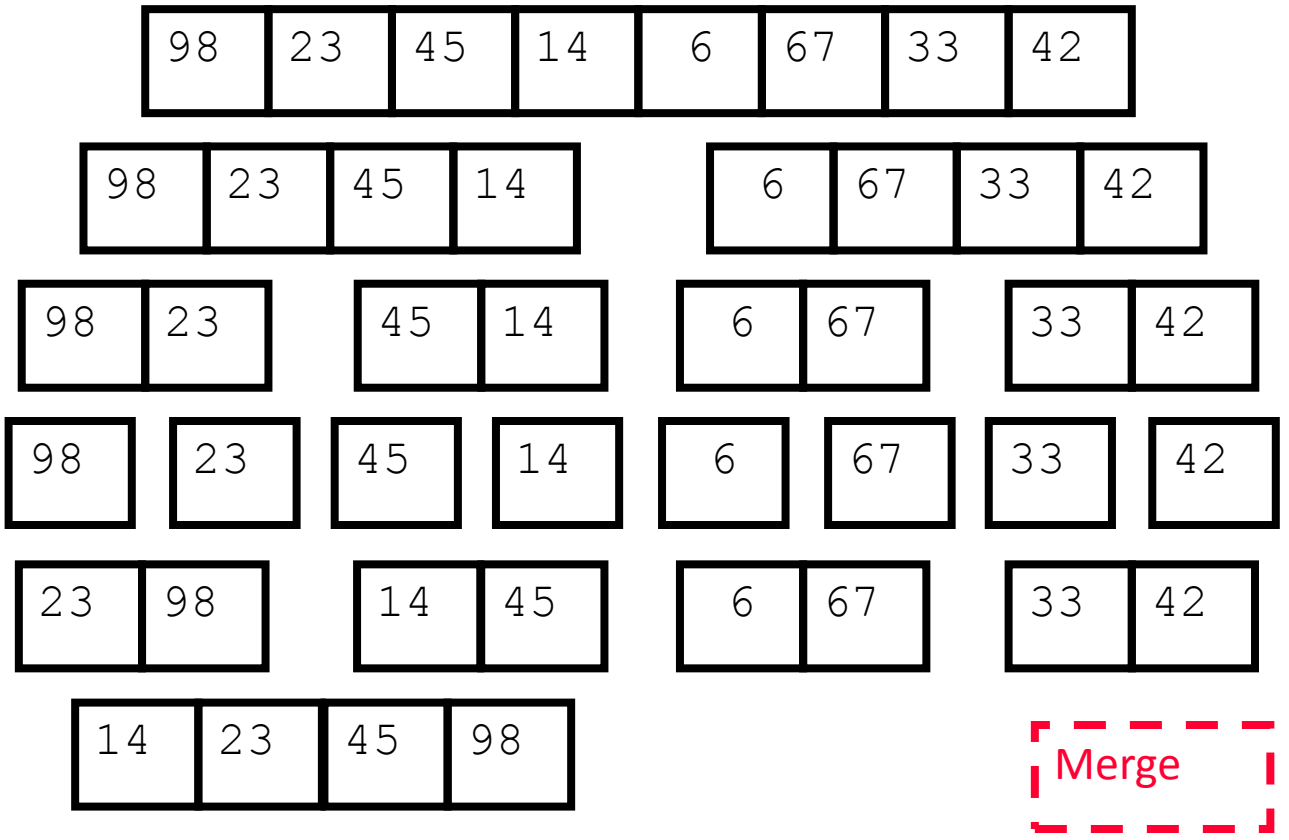
Merge Sort (Example)



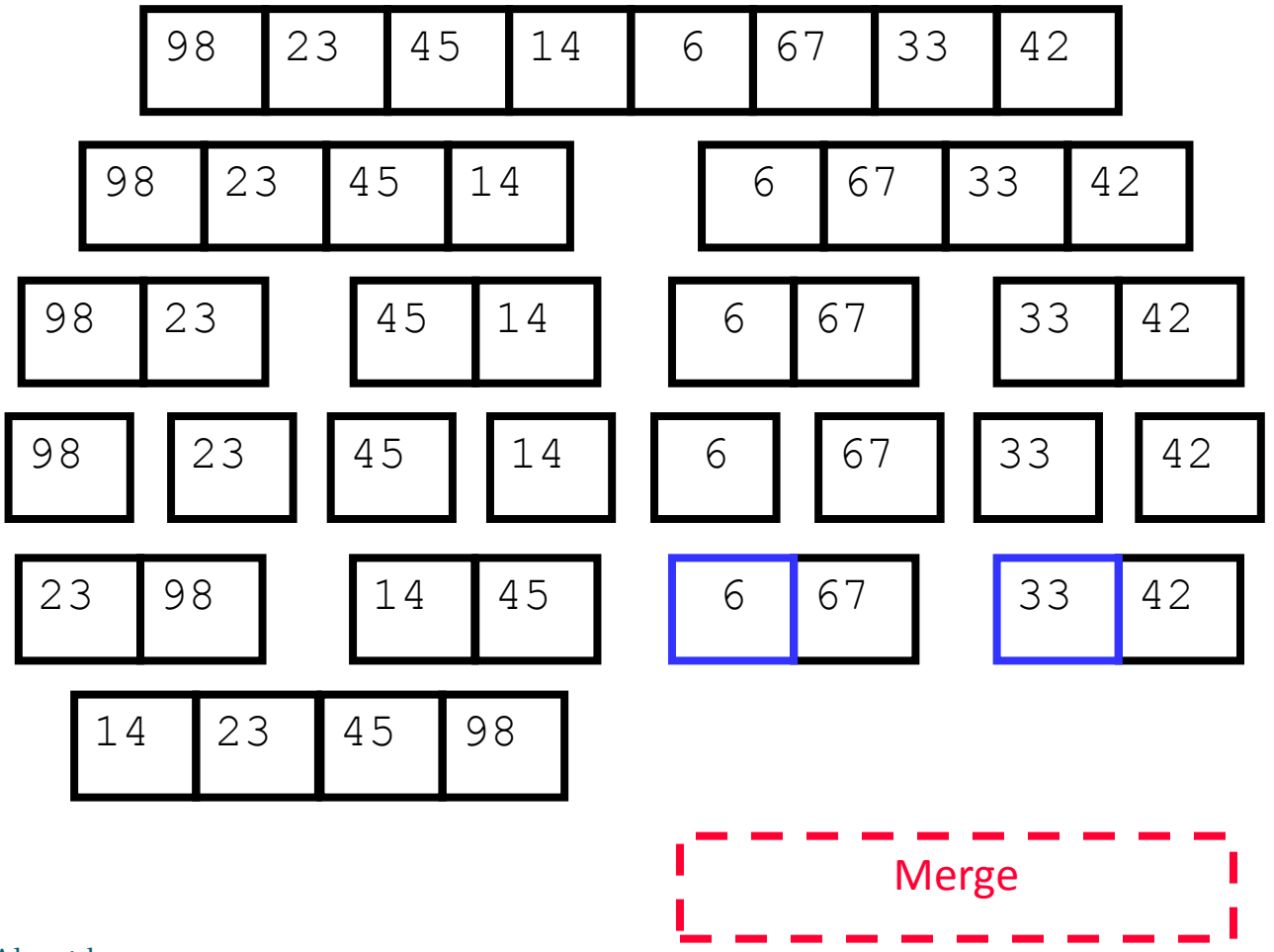
Merge Sort (Example)



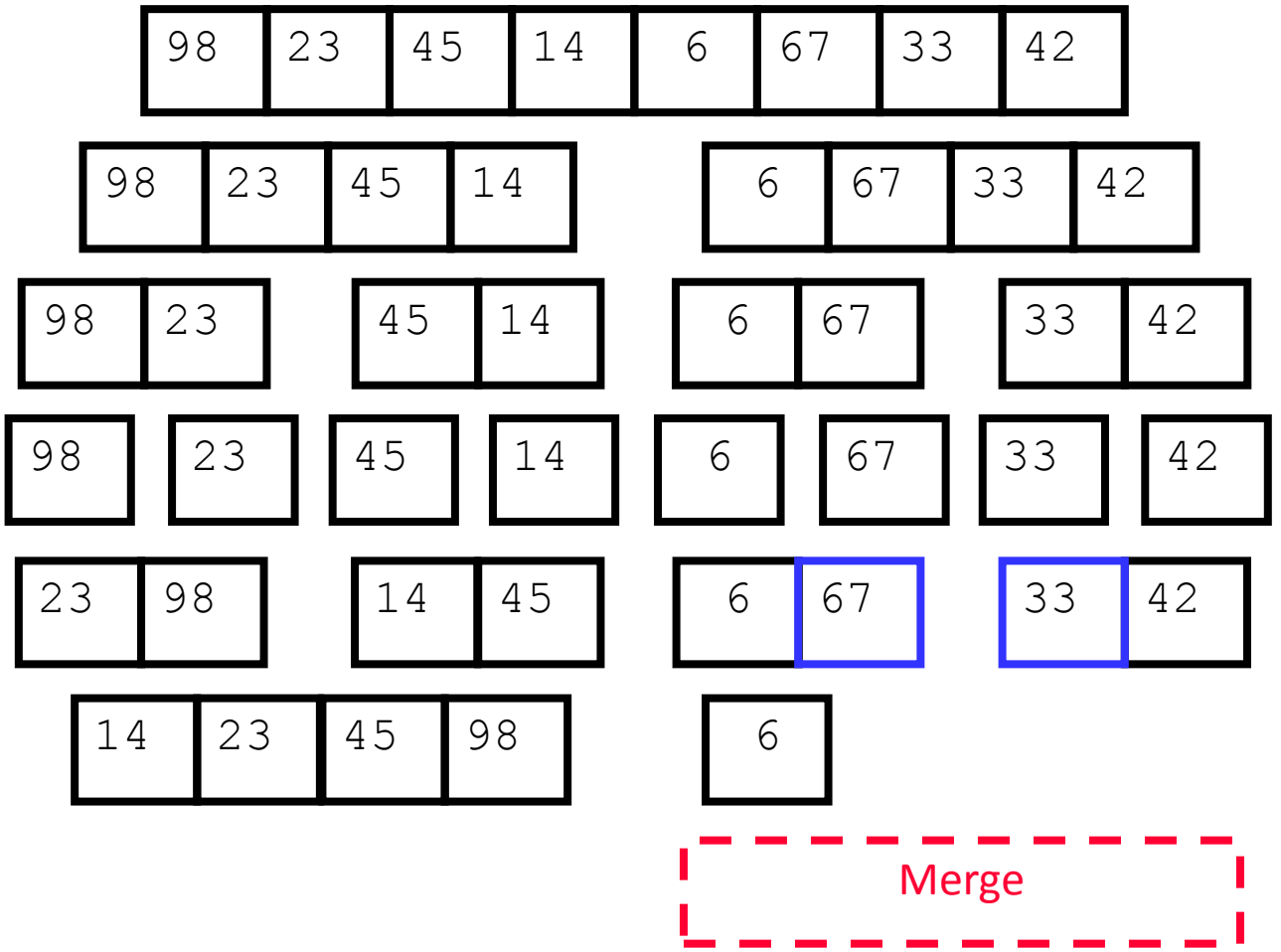
Merge Sort (Example)



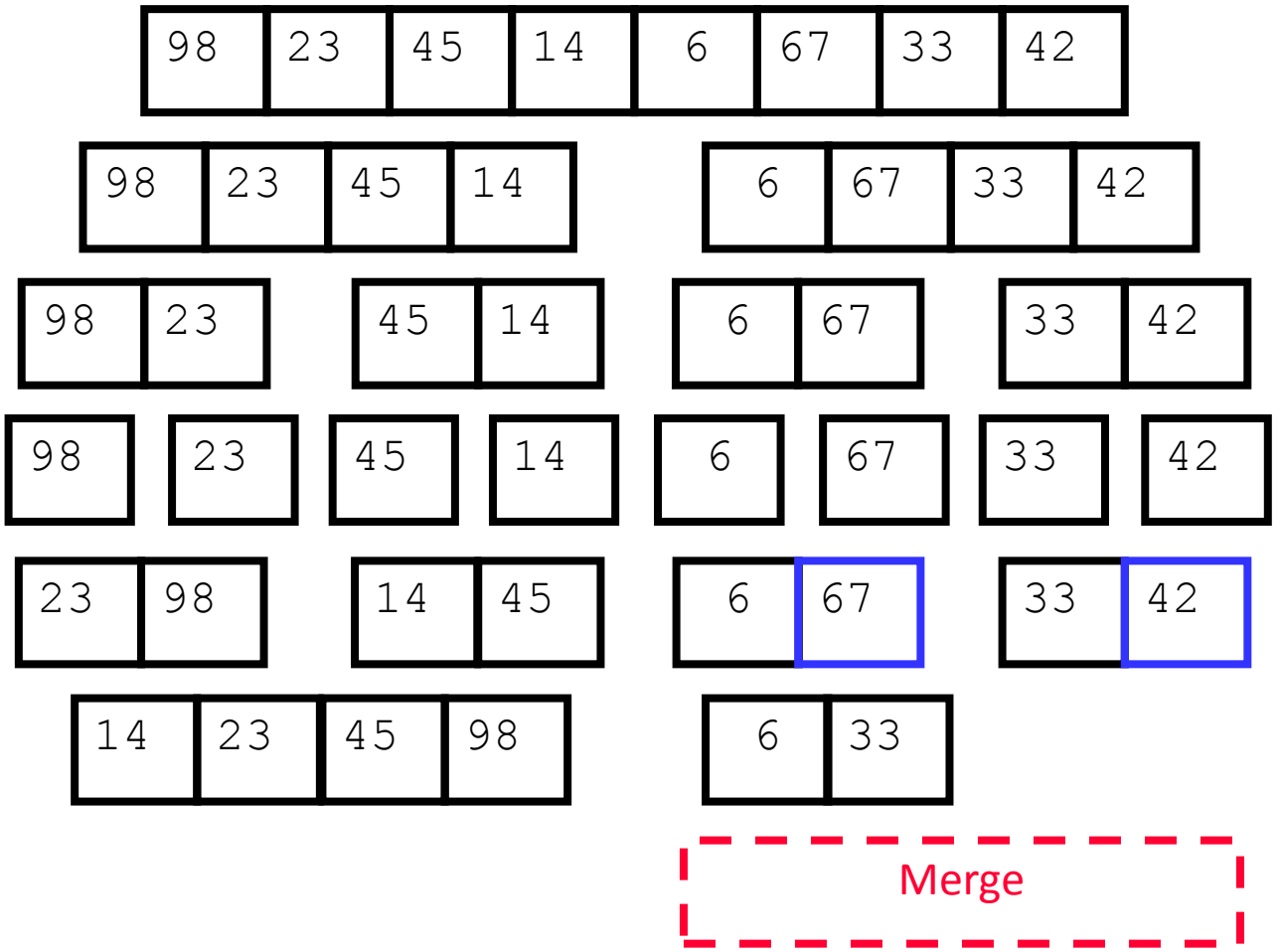
Merge Sort (Example)



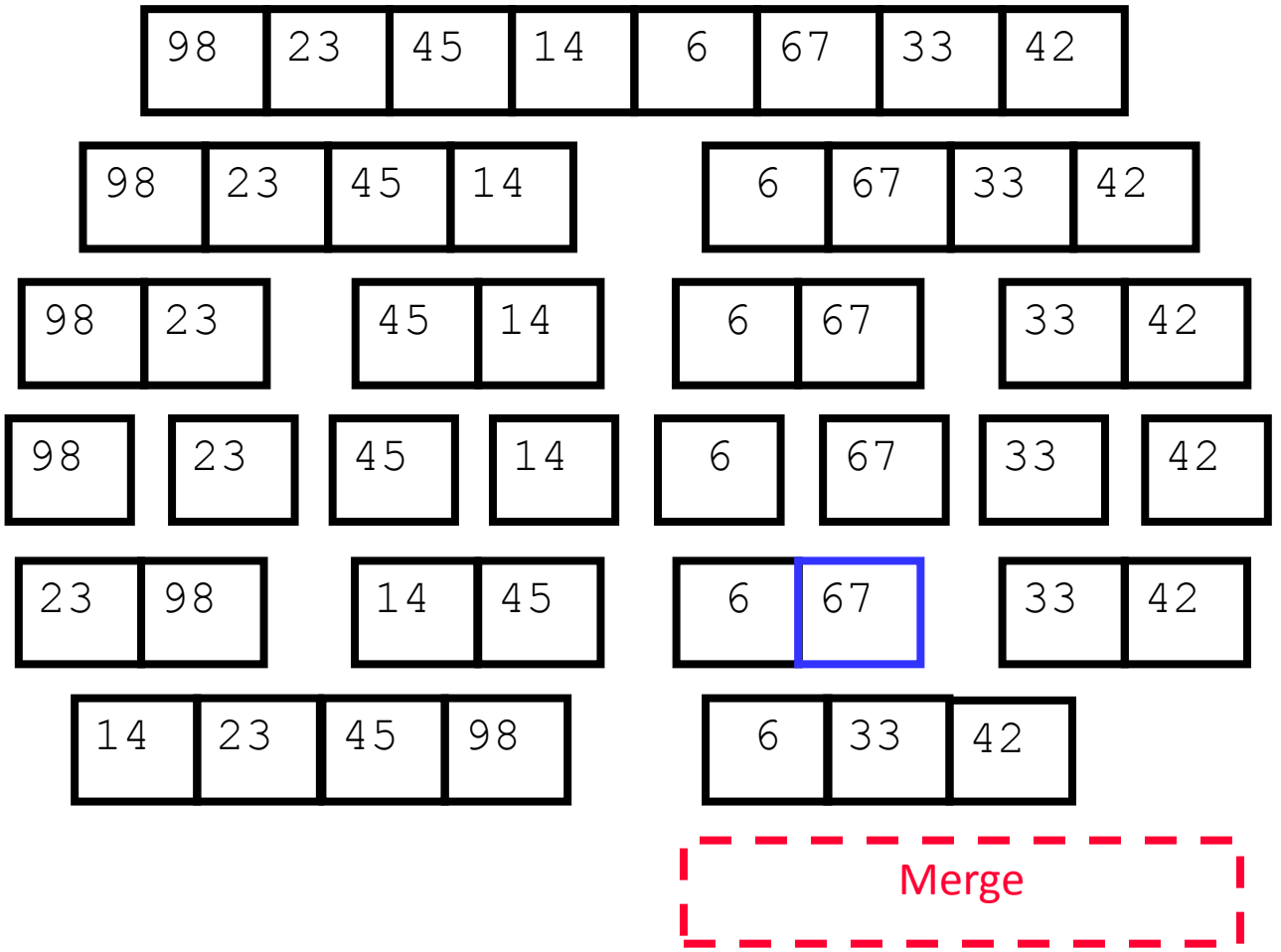
Merge Sort (Example)



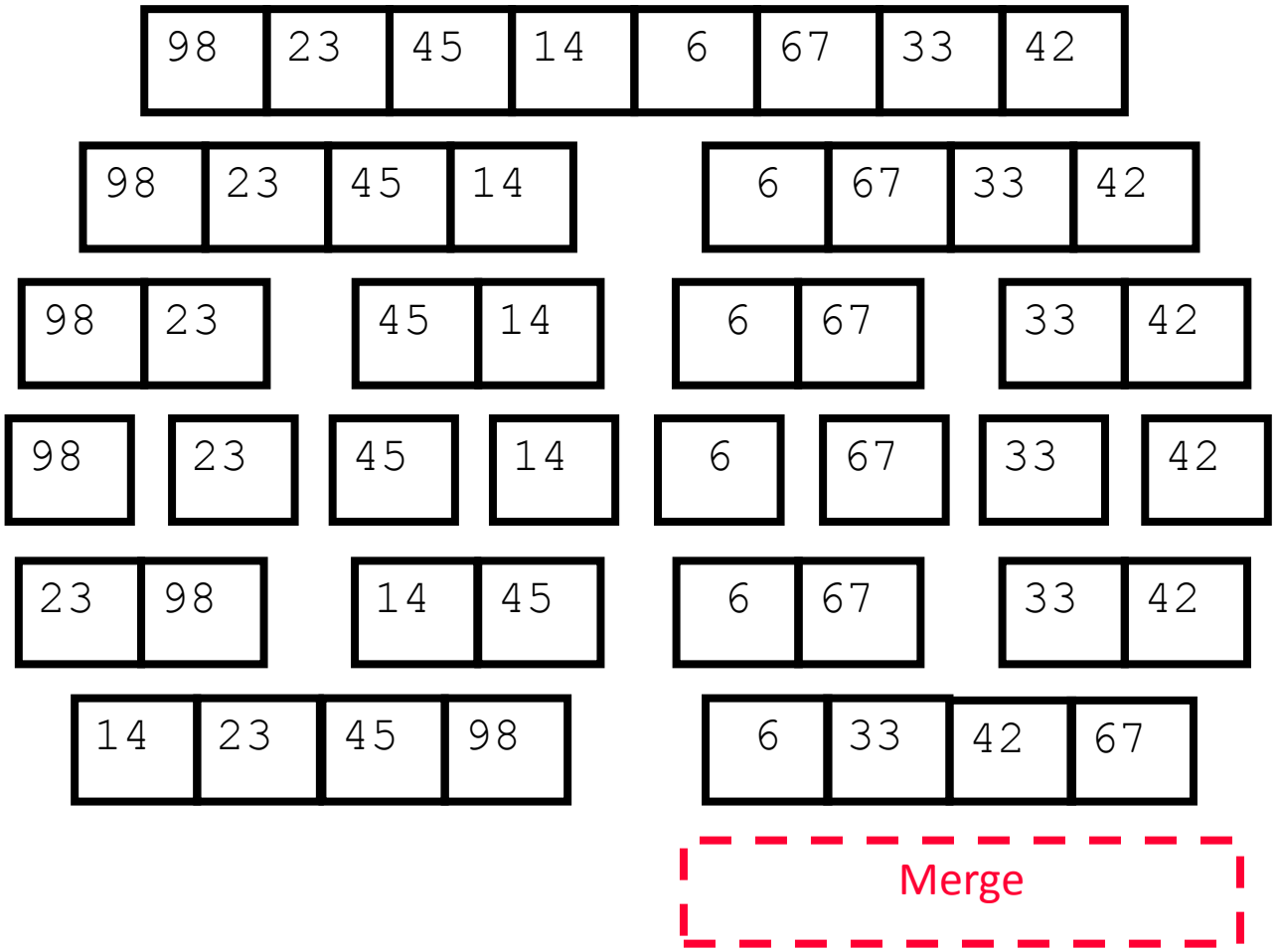
Merge Sort (Example)



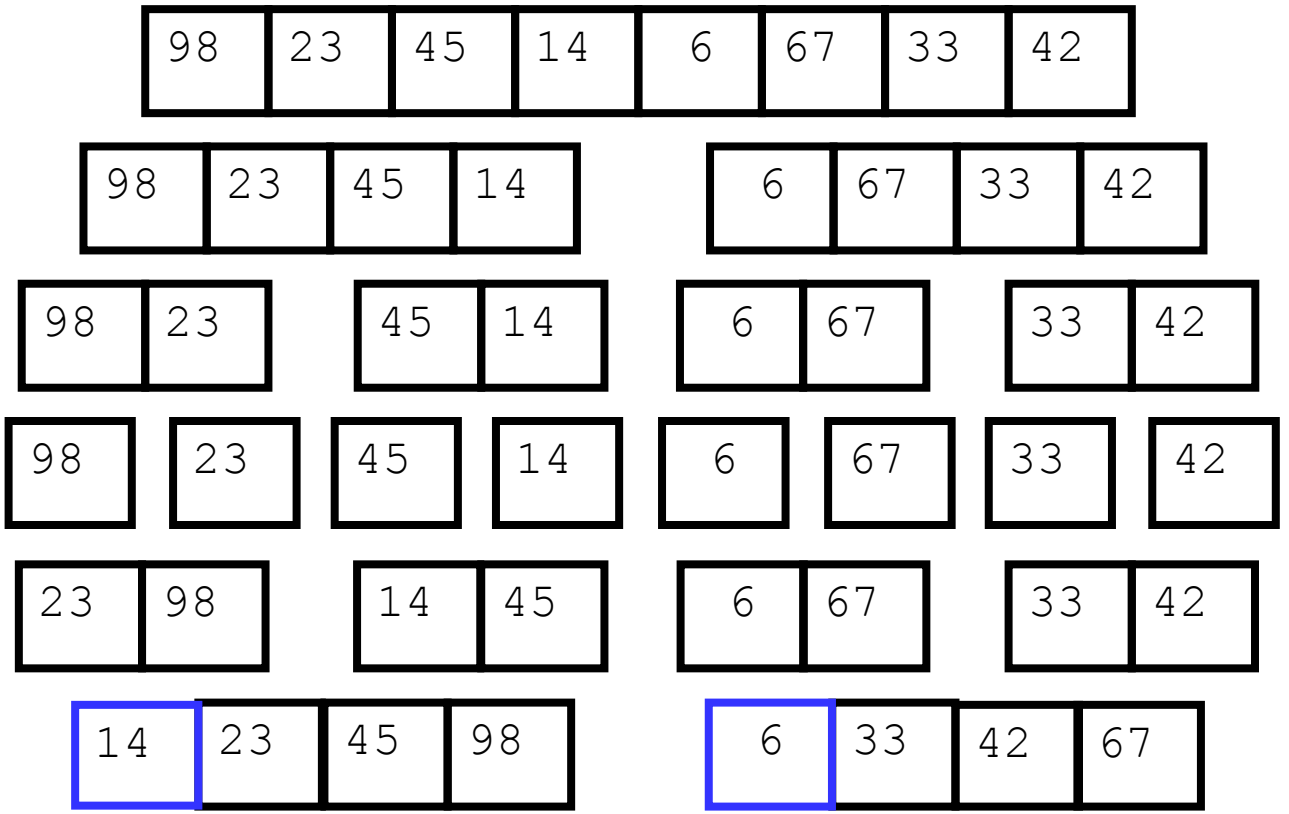
Merge Sort (Example)



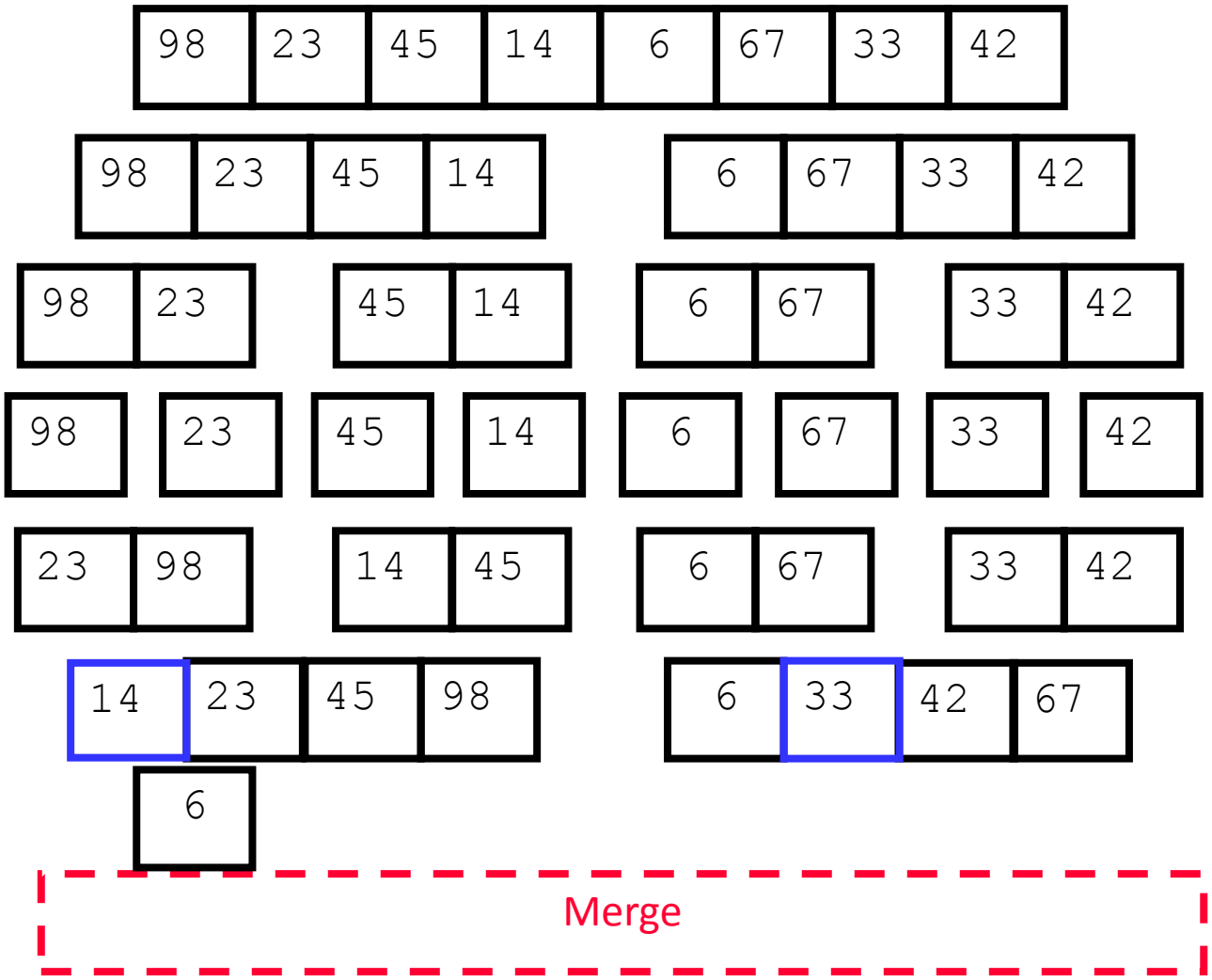
Merge Sort (Example)



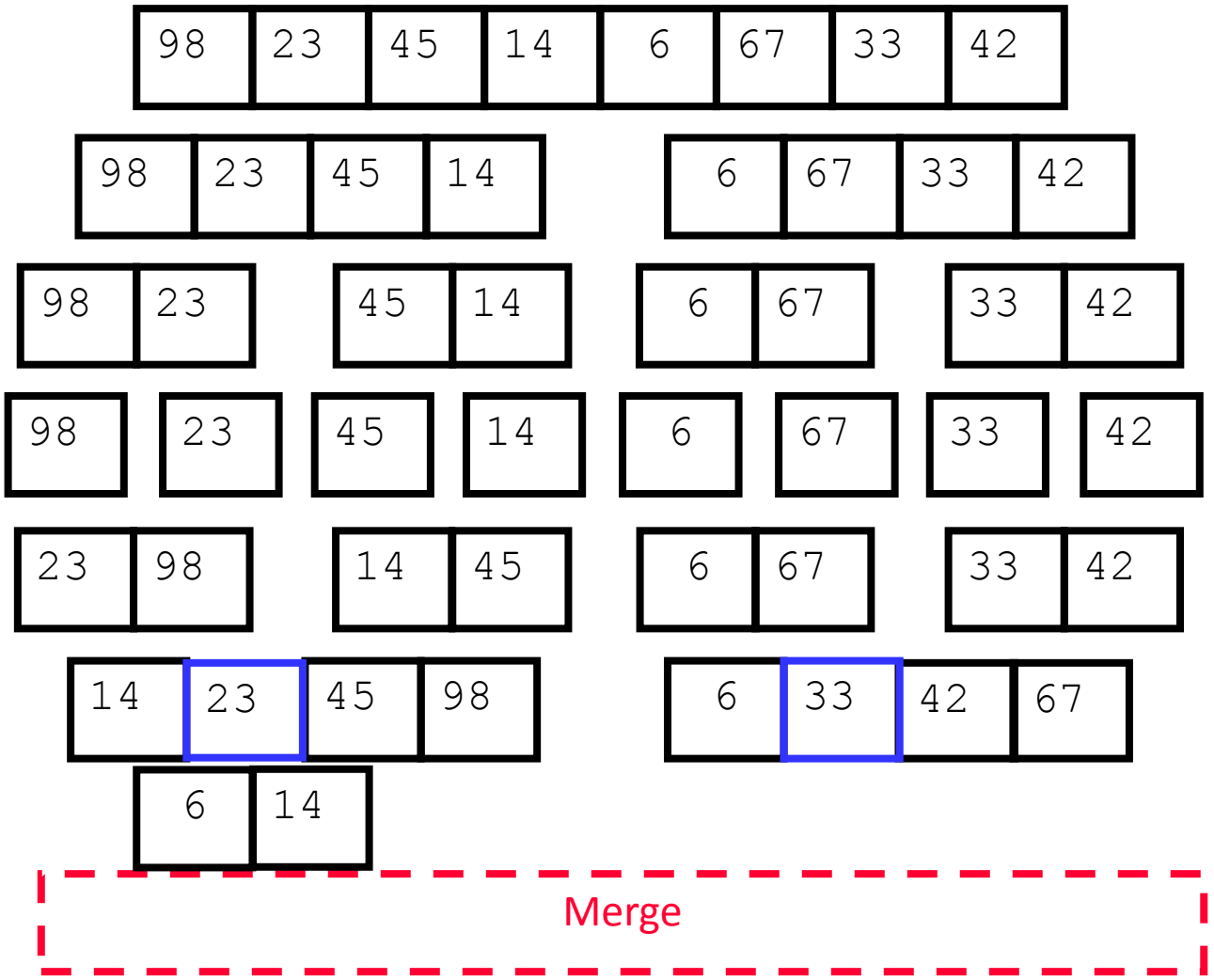
Merge Sort (Example)



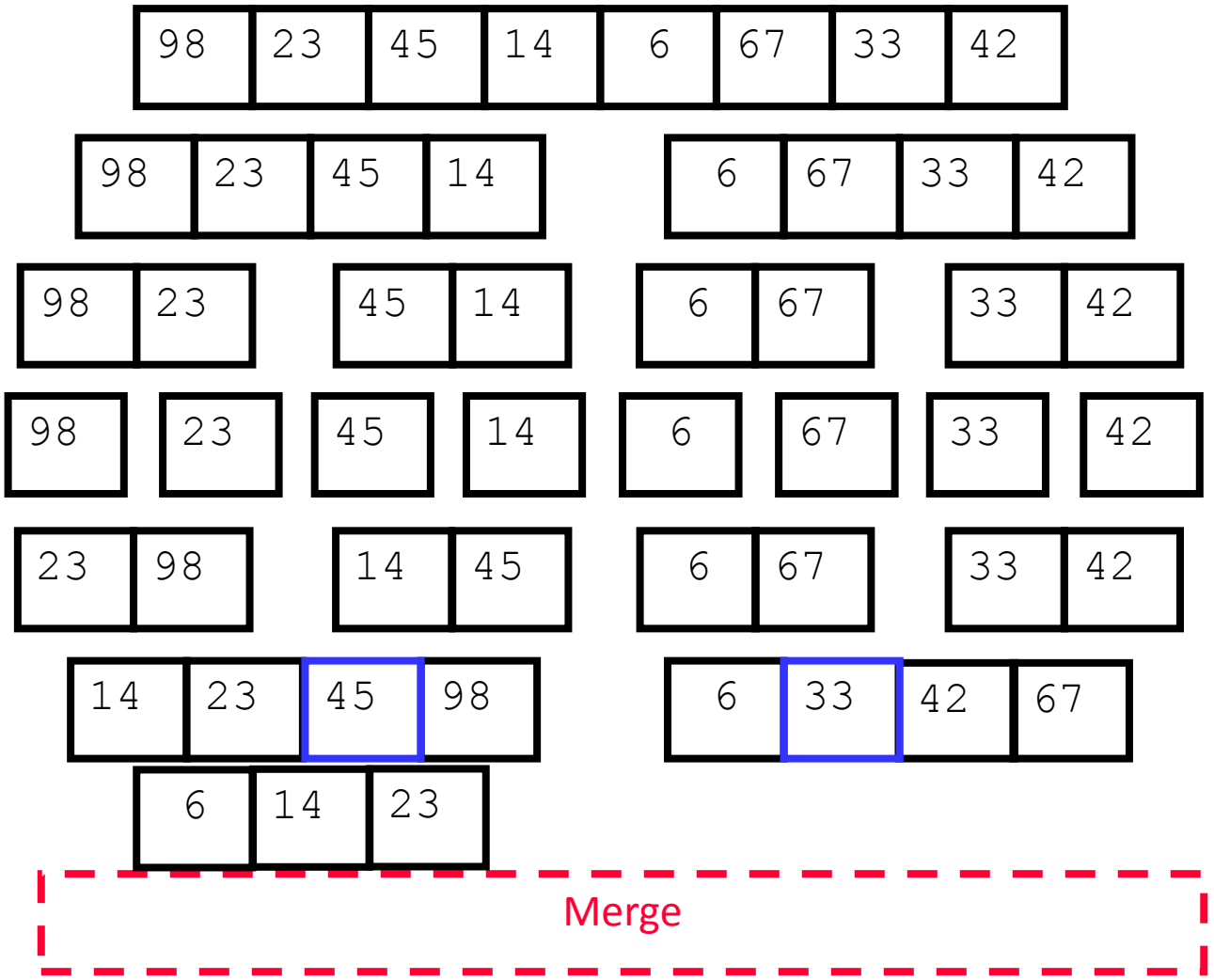
Merge Sort (Example)



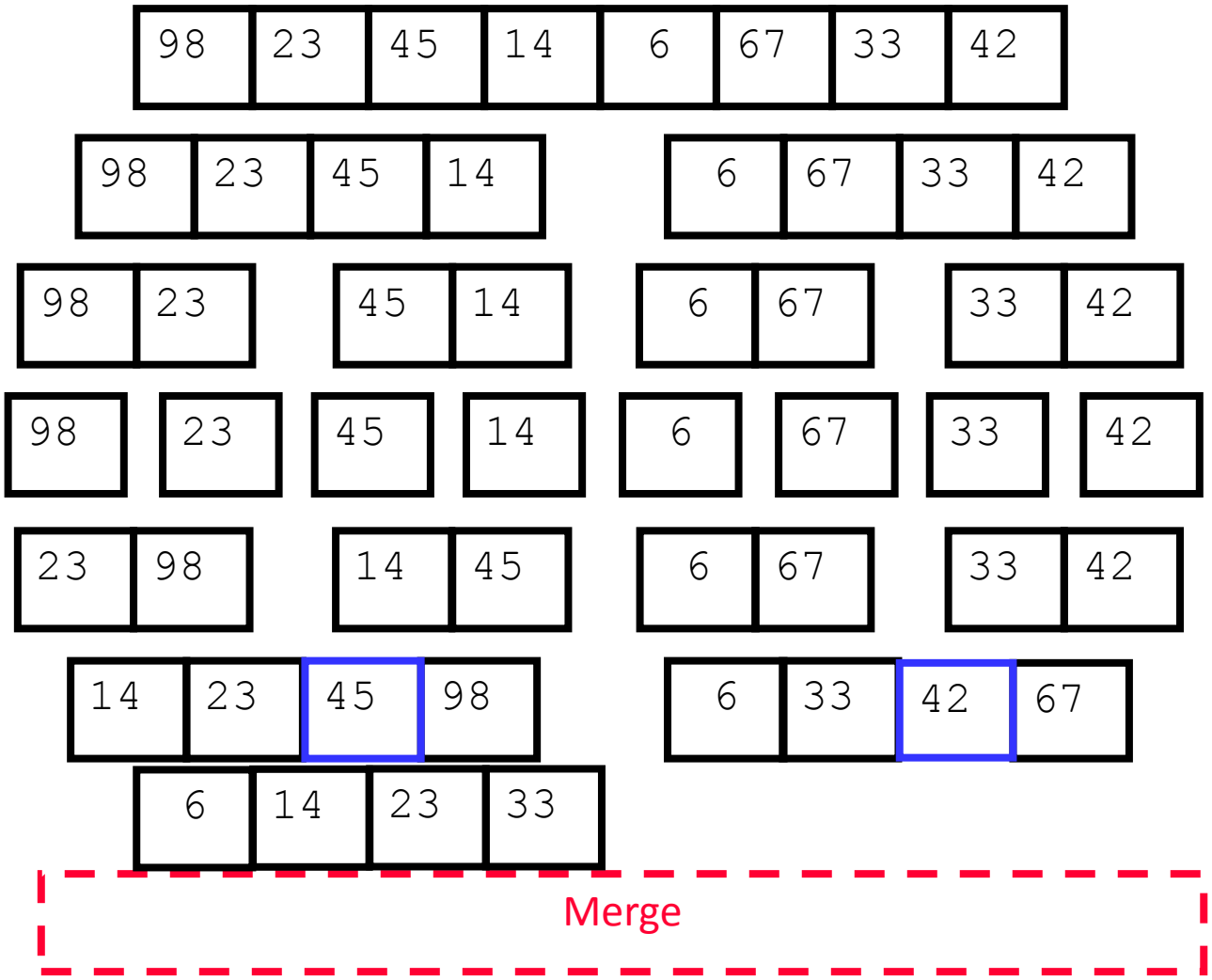
Merge Sort (Example)



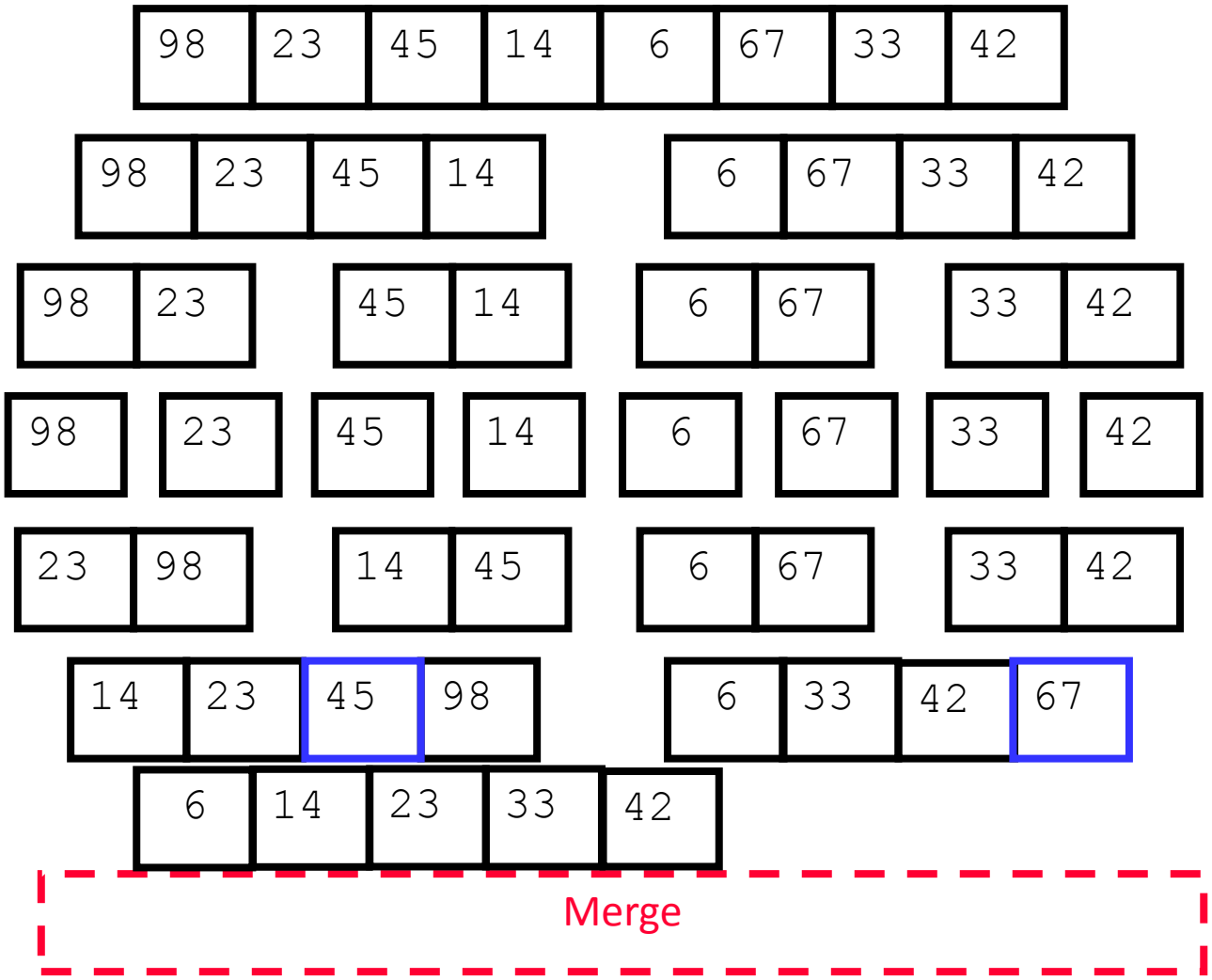
Merge Sort (Example)



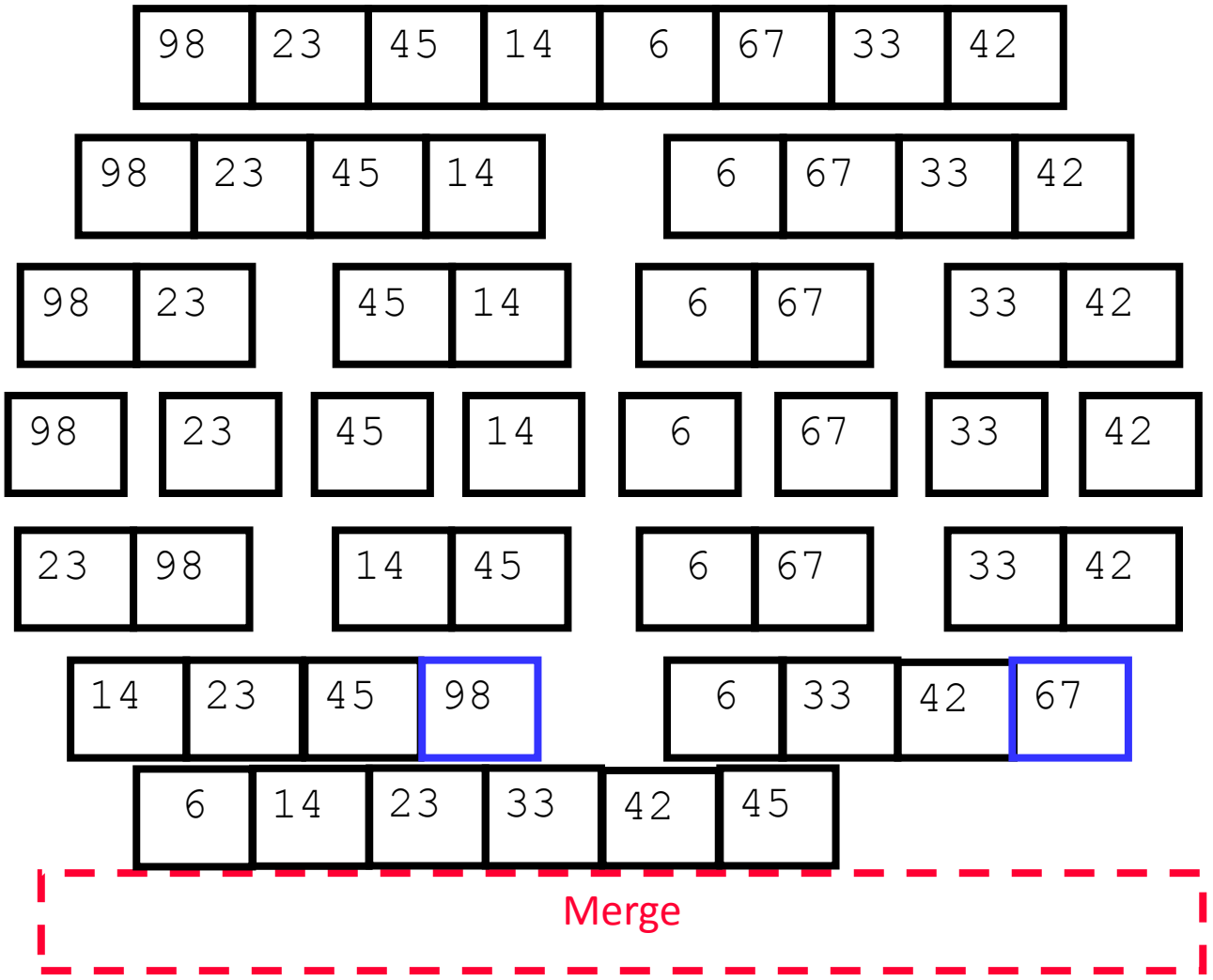
Merge Sort (Example)



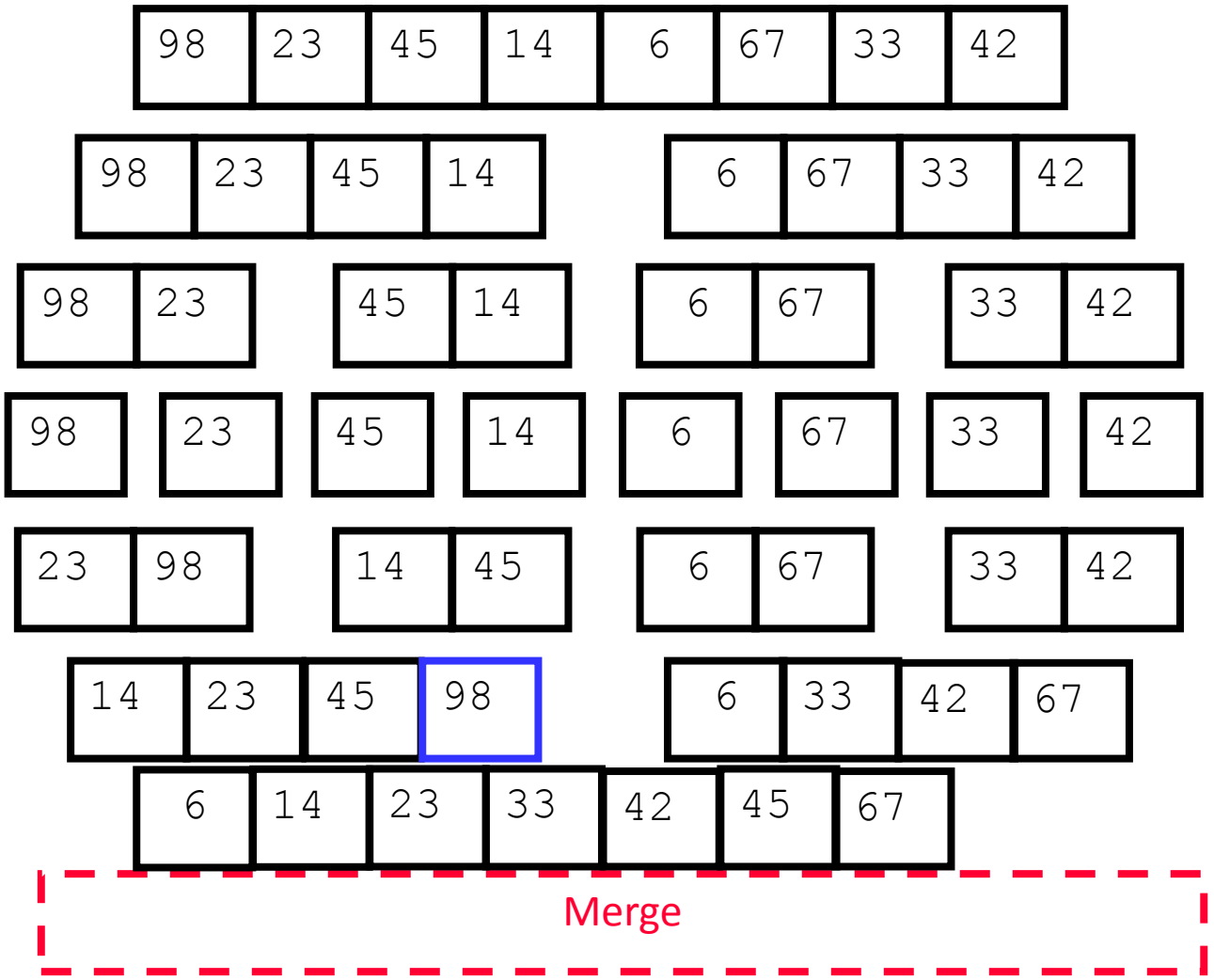
Merge Sort (Example)



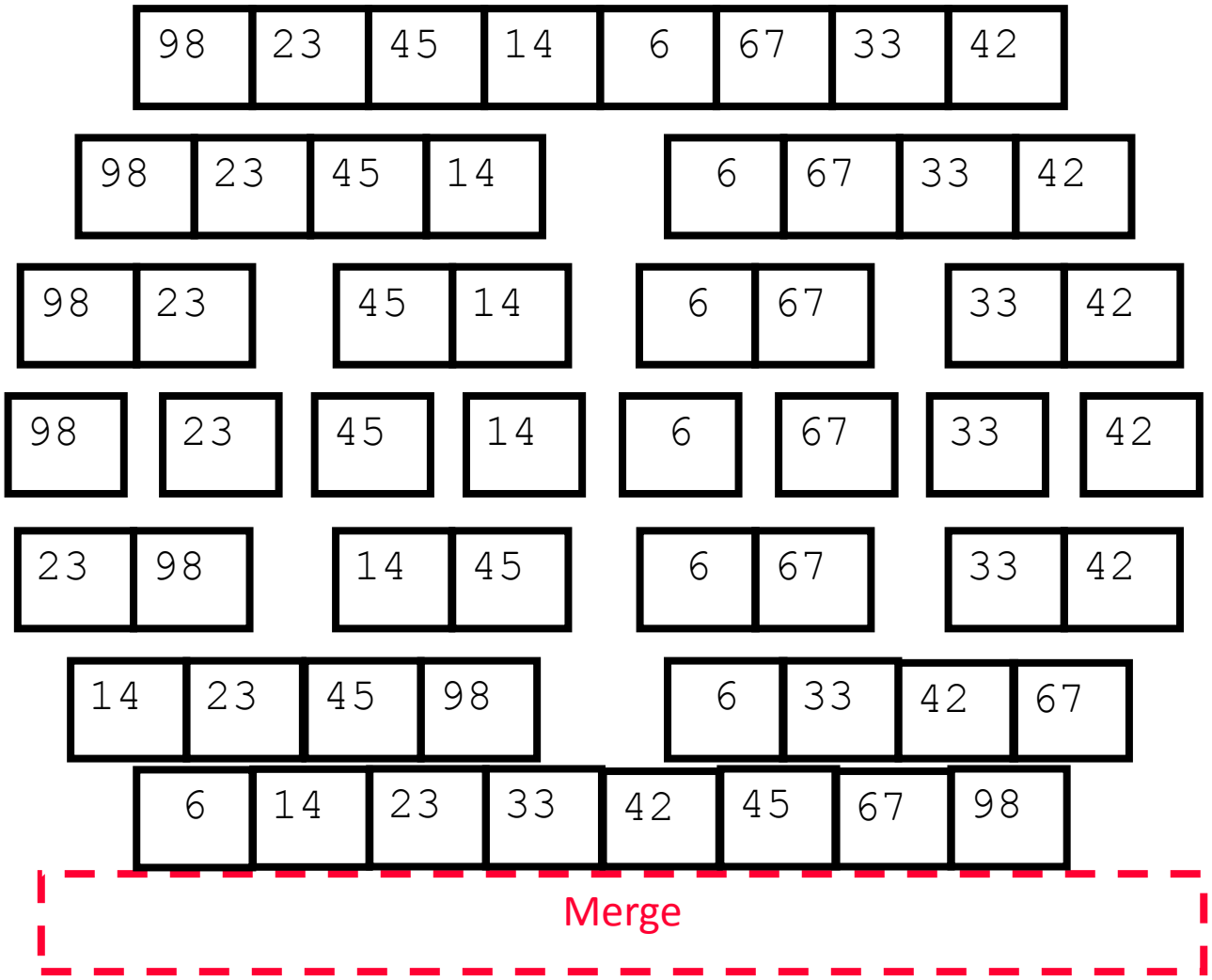
Merge Sort (Example)



Merge Sort (Example)



Merge Sort (Example)



Quicksort

Quicksort

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, $A.length$).

Quicksort

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

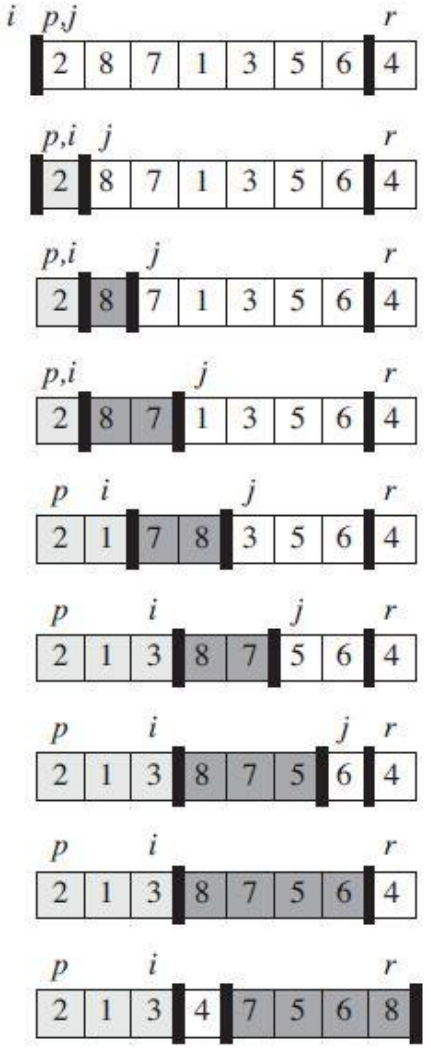
```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Quicksort

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```



Radix sort

Radix sort

- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.

Radix sort

- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.
- There are two classification of Radix sort

1. Least significant digit (LSD)
2. Most significant digit (MSD)

Radix sort

Least significant digit (LSD)

Definition: Each key is first figuratively dropped into one level of buckets corresponding to the value of the rightmost digit. Each bucket preserves the original order of the keys as the keys are dropped into the bucket. There is a one-to-one correspondence between the buckets and the values that can be represented by the rightmost digit. Then, the process repeats with the next neighboring more significant digit until there are no more digits to process.

Radix sort

In other words

1. Take the least significant digit of each key
2. Group the keys based on that digit, but otherwise keep the original order of keys.
3. Repeat the grouping process with each more significant digit.

Radix sort

Example

Original, unsorted list:

170, 45, 75, 90, 802, 2, 24, 66

Radix sort

Example

Original, unsorted list:

170, 45, 75, 90, 802, 2, 24, 66

Sorting by least significant digit (1s place) gives:

17<u>0</u>, 9<u>0</u>, 80<u>2</u>, <u>2</u>, 2<u>4</u>, 4<u>5</u>, 7<u>5</u>, 6<u>6</u>

Radix sort

Example

Original, unsorted list:

170, 45, 75, 90, 802, 2, 24, 66

Sorting by least significant digit (1s place) gives:

17<u>0</u>, 9<u>0</u>, 80<u>2</u>, <u>2</u>, 2<u>4</u>, 4<u>5</u>, 7<u>5</u>, 6<u>6</u>

Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.

Radix sort

Example

Original, unsorted list:

170, 45, 75, 90, 802, 2, 24, 66

Sorting by least significant digit (1s place) gives:

17<u>0</u>, 9<u>0</u>, 80<u>2</u>, <u>2</u>, 2<u>4</u>, 4<u>5</u>, 7<u>5</u>, 6<u>6</u>

Sorting by next digit (10s place) gives:

8<u>0</u>2, 2, <u>2</u>4, <u>4</u>5, <u>6</u>6, 1<u>7</u>0, <u>7</u>5, <u>9</u>0

Radix sort

Example

Original, unsorted list:

170, 45, 75, 90, 802, 2, 24, 66

Sorting by least significant digit (1s place) gives:

17<u>0</u>, 9<u>0</u>, 80<u>2</u>, <u>2</u>, 2<u>4</u>, 4<u>5</u>, 7<u>5</u>, 6<u>6</u>

Sorting by next digit (10s place) gives:

8<u>0</u>2, 2, <u>2</u>4, <u>4</u>5, <u>6</u>6, 1<u>7</u>0, <u>7</u>5, <u>9</u>0

Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.

Radix sort

Example

Original, unsorted list:

170, 45, 75, 90, 802, 2, 24, 66

Sorting by least significant digit (1s place) gives:

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives:

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

Radix sort

Iterative version using queues

A simple version of an LSD radix sort can be achieved using queues as buckets.

Radix sort

Iterative version using queues

1. The integers are enqueued into an array of ten separate queues based on their digits from right to left. Computers often represent integers internally as fixed-length binary digits. Here, we will do something analogous with fixed-length decimal digits. So, using the numbers from the previous example, the queues for the 1st pass would be:
   0: 170, 090
   1: none
   2: 802, 002
   3: none
   4: 024
   5: 045, 075
   6: 066
   7–9: none

Radix sort

Iterative version using queues

2. The queues are dequeued back into an array of integers, in increasing order. Using the same numbers, the array will look like this after the first pass:
   170, 090, 802, 002, 024, 045, 075, 066

Radix sort

Iterative version using queues

2. The queues are dequeued back into an array of integers, in increasing order. Using the same numbers, the array will look like this after the first pass:
   170, 090, 802, 002, 024, 045, 075, 066

3. For the second pass:
   Queues:
   - 0: 802, 002
   - 1: none
   - 2: 024
   - 3: none
   - 4: 045
   - 5: none
   - 6: 066
   - 7: 170, 075
   - 8: none
   - 9: 090

   Array:
   802, 002, 024, 045, 066, 170, 075, 090
   (note that at this point only 802 and 170 are out of order)

Radix sort

Iterative version using queues

4. For the third pass:

   Queues:

   0: 002, 024, 045, 066, 075, 090

   1: 170

   2–7: none

   8: 802

   9: none

Array:

002, 024, 045, 066, 075, 090, 170, 802 (sorted)

Radix sort

Iterative version using queues

4. For the third pass:

   Queues:

   0: 002, 024, 045, 066, 075, 090

   1: 170

   2–7: none

   8: 802

   9: none

Array:

002, 024, 045, 066, 075, 090, 170, 802 (sorted)

Radix sort

General algorithm for Radix sort

1. Repeat thru step 6 for each digit in the key
2. Initialize the pockets
3. Repeat thru step 5 until the end of the linked list
4. Obtain the next digit of the key
5. Insert the record in the appropriate pocket
6. Combine the pockets to form a new linked list

Radix sort

Input : 42, 23, 74, 11, 65, 57, 94, 36, 99, 87, 70, 81, 61

Radix sort

Input : 42, 23, 74, 11, 65, 57, 94, 36, 99, 87, 70, 81, 61

After the first pass on the unit digit position of each number we have:

	61								
	81			94			87		
70	11	42	23	74	65	36	57		99
Pocket:	0	1	2	3	4	5	6	7	8	9