Data Structures and Algorithms

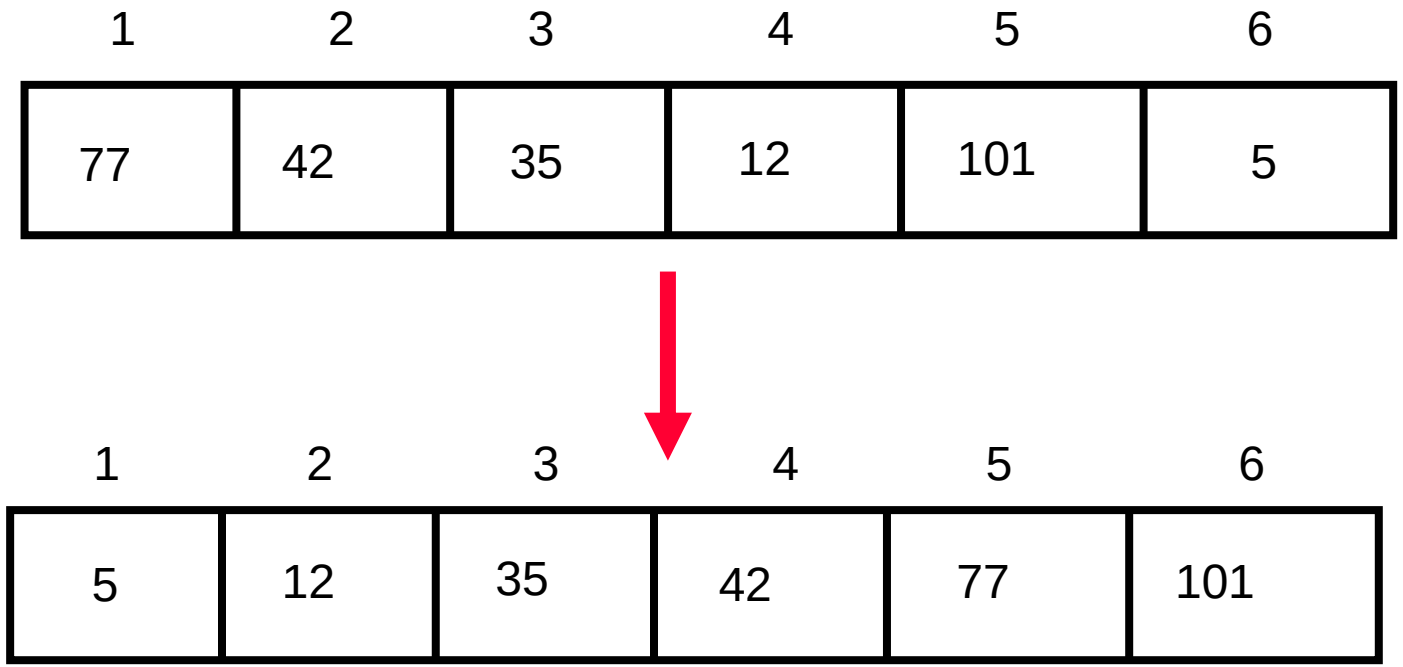
Mr. Tahir Iqbal

tahir.iqbal@bahria.edu.pk

Lecture 10: Sorting

Sorting

- Sorting is a process in which records are arranged in ascending or descending order



Types of sorting

- Selection sort
- Insertion sort
- Bubble sort
- Merge sort
- Quick sort
- Heap sort
- Shell sort

Selection Sort

Selection Sort

- **Selection sort** is a sorting algorithm which works as follows:
  - Find the minimum value in the list
  - Swap it with the value in the first position
  - Repeat the steps above for remainder of the list (starting at the second position)

Example: Selection Sort

- 26 33 43 100 46 88 52 17 53 77
- 17 | 33 43 100 46 88 52 26 53 77
- 17 26 | 43 100 46 88 52 33 53 77
- 17 26 33 | 100 46 88 52 43 53 77
- 17 26 33 43 | 46 88 52 100 53 77
- 17 26 33 43 46 | 88 52 100 53 77
- 17 26 33 43 46 52 | 88 100 53 77
- 17 26 33 43 46 52 53 | 100 88 77
- 17 26 33 43 46 52 53 77 | 88 100
- 17 26 33 43 46 52 53 77 88 | 100

Selection Sort

```
void selectionSort(int numbers[], int array_size) {
   int i, j, T, min, count;
   for (i = 0; i < array_size; i++) {
      min = i;
      for (j = i + 1; j < array_size; j++) {
         if (numbers[j] < numbers[min]) {
            min = j; }
      }
      T = numbers[min];
      numbers[min] = numbers[i];
      numbers[i] = T;
    }
}
```

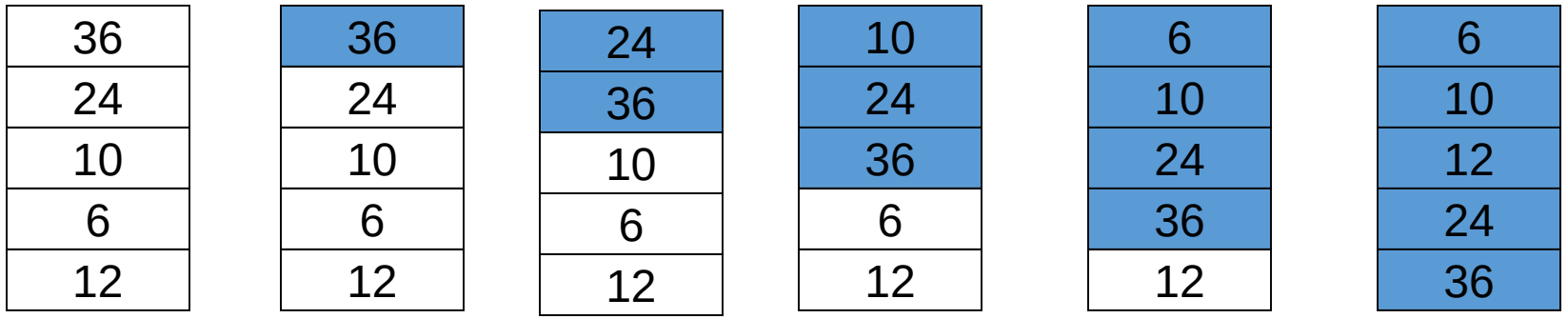
Insertion Sort

Insertion Sort

- In insertion sort, each successive element in the array to be sorted is inserted into its proper place with respect to the other, already sorted elements.
- We divide our array in a sorted and an unsorted array
- Initially the sorted portion contains only one element: the first element in the array.
- We take the second element in the array, and put it into its correct place

Insertion Sort

- That is, array[0] and array[1] are in order with respect to each other.
- Then the value in array[2] is put into its proper place, so array [0]…. array[2] is sorted and so on.

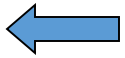


Insertion Sort

- Our strategy is to search for insertion point from the beginning of the array and shift the element down to make room for new element
- We compare the item at array[current] to one before it, and swap if it is less.
- We then compare array[current-1] to one before it and swap if necessary.
-

Example: Insertion Sort

- 99 | 55 4 66 28 31 36 52 38 72
- 55 99 | 4 66 28 31 36 52 38 72
- 4 55 99 | 66 28 31 36 52 38 72
- 4 55 66 99 | 28 31 36 52 38 72
- 4 28 55 66 99 | 31 36 52 38 72
- 4 28 31 55 66 99 | 36 52 38 72
- 4 28 31 36 55 66 99 | 52 38 72
- 4 28 31 36 52 55 66 99 | 38 72
- 4 28 31 36 38 52 55 66 99 | 72
- 4 28 31 36 38 52 55 66 72 99 |



99 554 66 28 31 36 52 38 72

Example: Insertion Sort

- 99 55 4 66 28 31 36 52 38 72
- 99 66 55 4 28 31 36 52 38 72
- 99 66 55 28 4 31 36 52 38 72
- 99 66 55 31 28 4 36 52 38 72
- 99 66 55 36 31 28 4 52 38 72
- 99 66 55 52 36 31 28 4 38 72
- 99 66 55 52 38 36 31 28 4 72
- 99 72 66 55 52 38 36 31 28 4

Example: Insertion Sort

- 99 554 66 28 31 36 52 38 72
- 554 99 66 28 31 36 52 38 72
- 554 99 66 31 28 36 52 38 72
- 554 99 66 36 31 28 52 38 72
- 554 99 66 52 36 31 28 38 72
- 554 99 66 52 38 36 31 28 72
- 554 99 72 66 52 38 36 31 28

Insertion Sort Algorithm

```
void insertionSort(int array[], int length)
{
  int i, j, value;
  for(i = 1; i < length; i++)
  {
      value = a[i];
      for (j = i - 1; j >= 0 && a[ j ] > value; j--)
      {
          a[j + 1] = a[ j ];
      }
      a[j + 1] = value;
   }
}
```

Bubble Sort

Bubble Sort

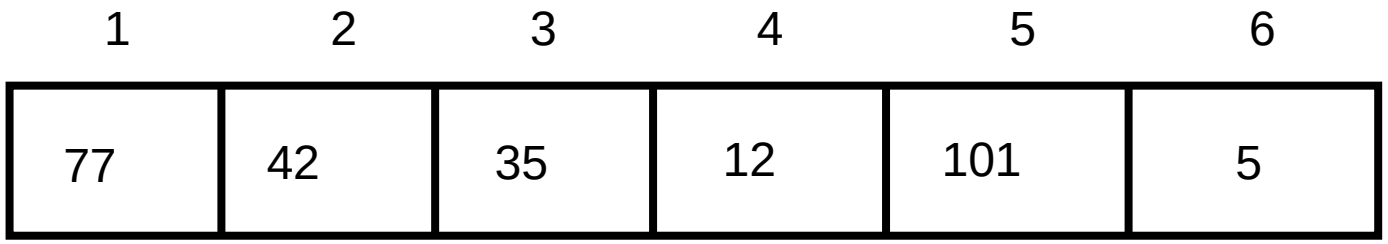
- Bubble sort is similar to selection sort in the sense that it repeatedly finds the largest/smallest value in the unprocessed portion of the array and puts it back.
- However, finding the largest value is not done by selection this time.
- We "bubble" up the largest value instead.

Bubble Sort

- Compares adjacent items and exchanges them if they are out of order.
- Comprises of several passes.
- In one pass, the largest value has been “bubbled” to its proper position.
- In second pass, the last value does not need to be compared.

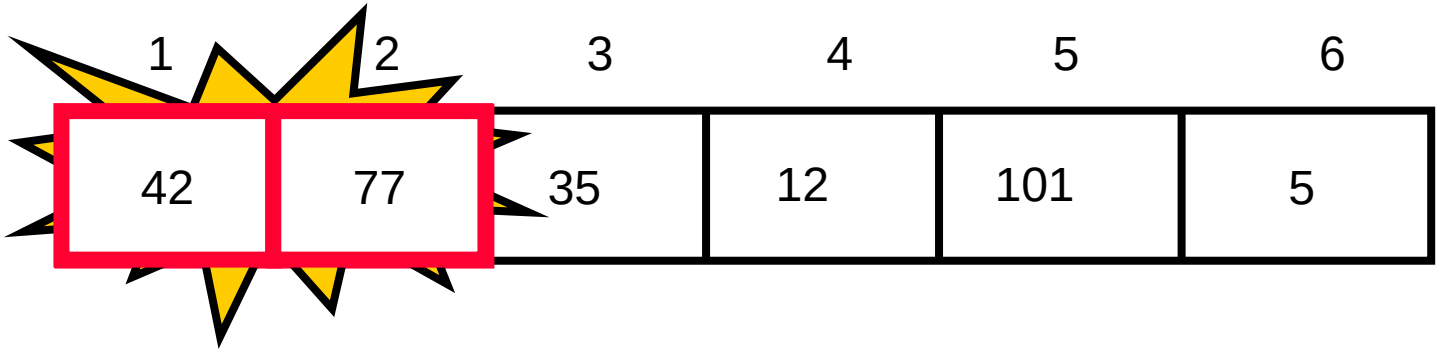
Bubble Sort

- 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



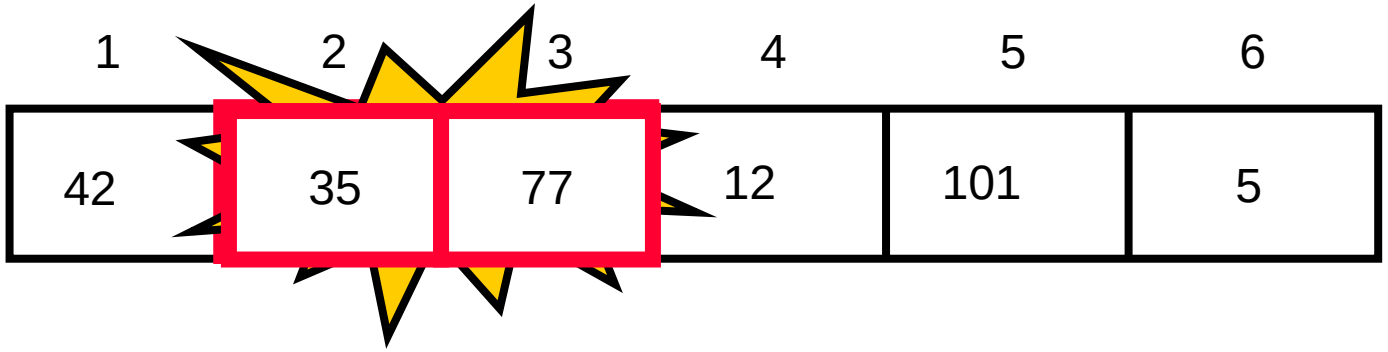
Bubble Sort

- 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



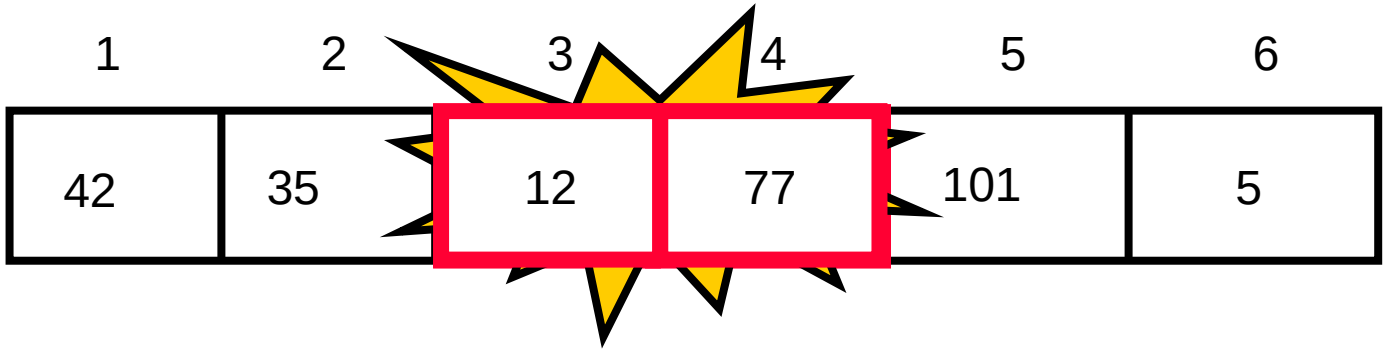
Bubble Sort

- 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



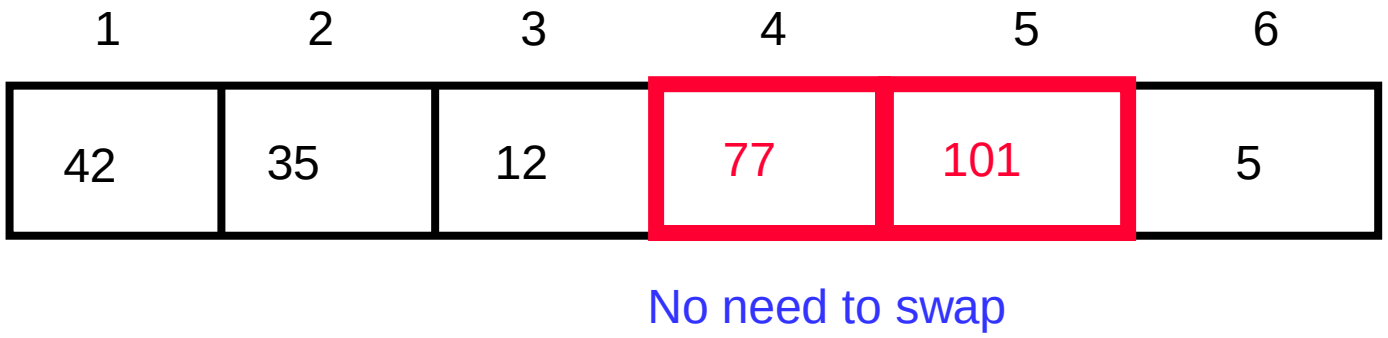
Bubble Sort

- 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



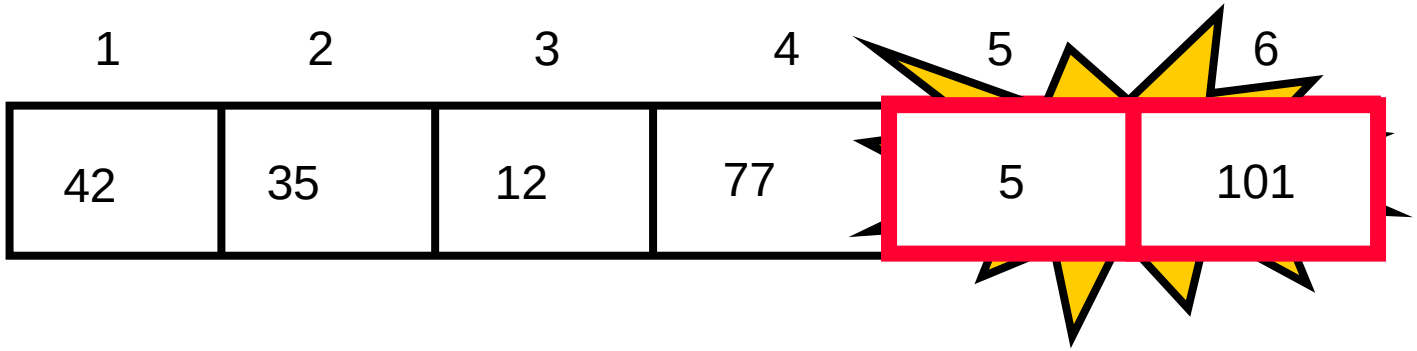
Bubble Sort

- 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



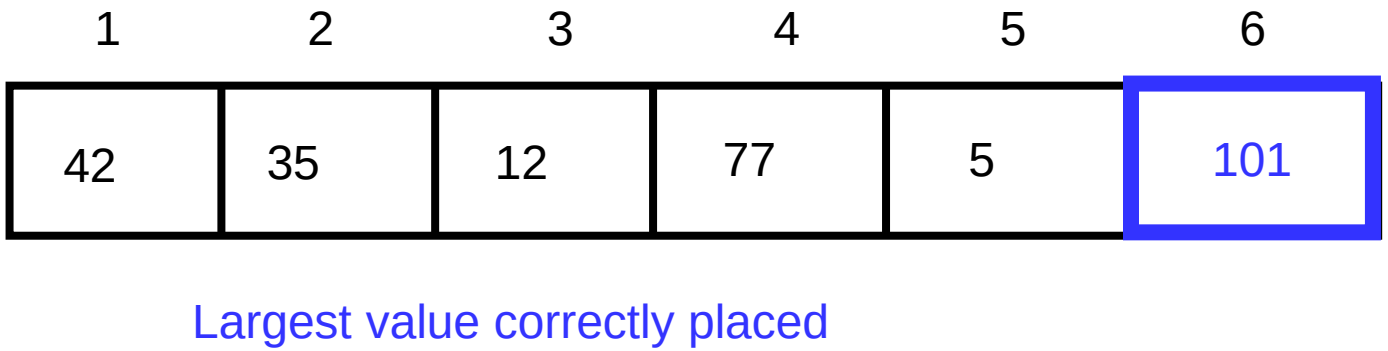
Bubble Sort

- 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



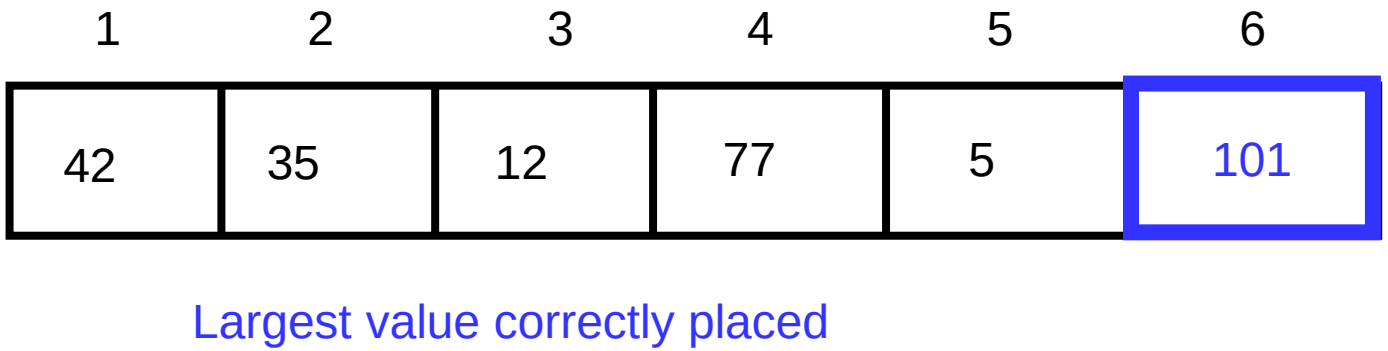
Bubble Sort

- 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

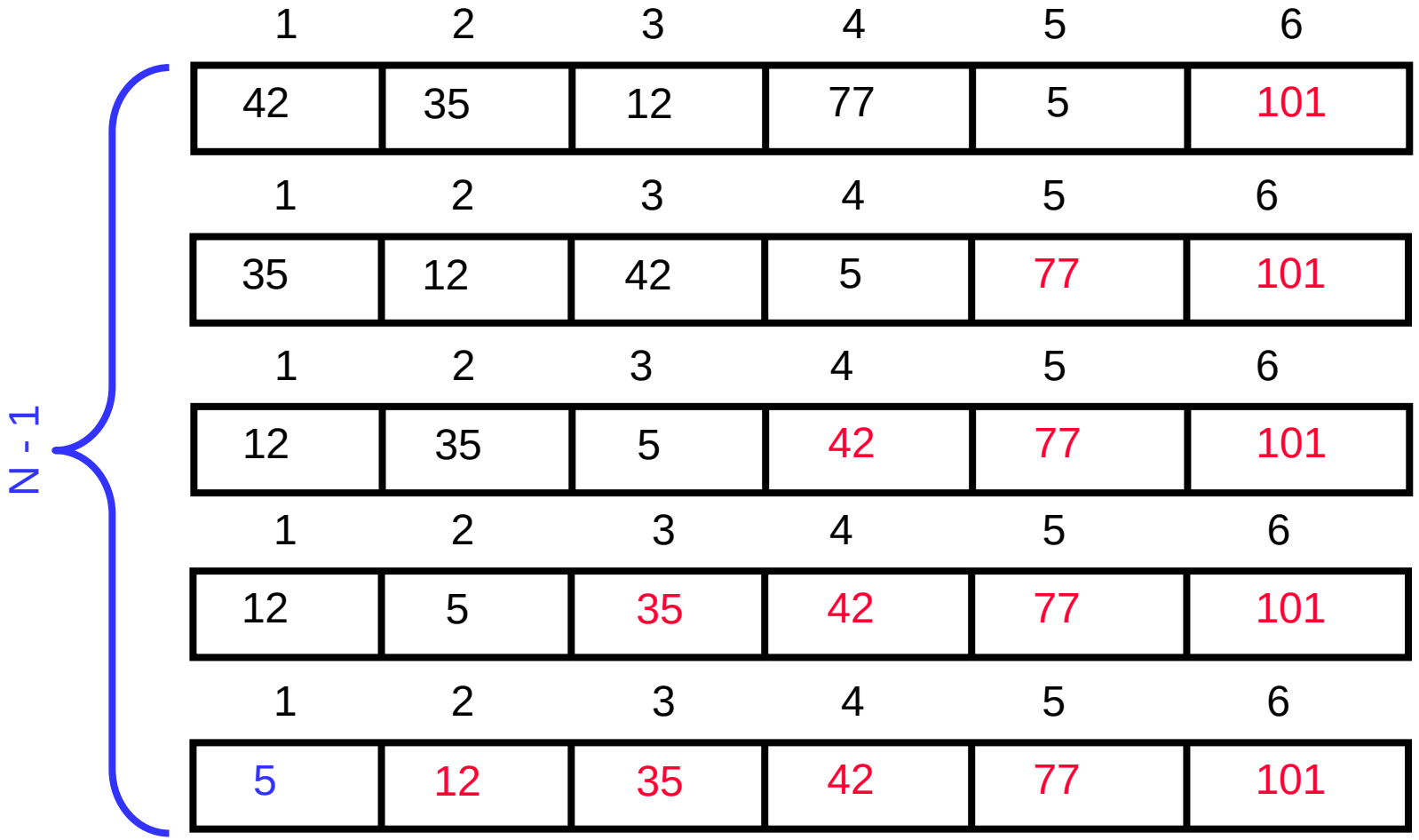


Items of Interest

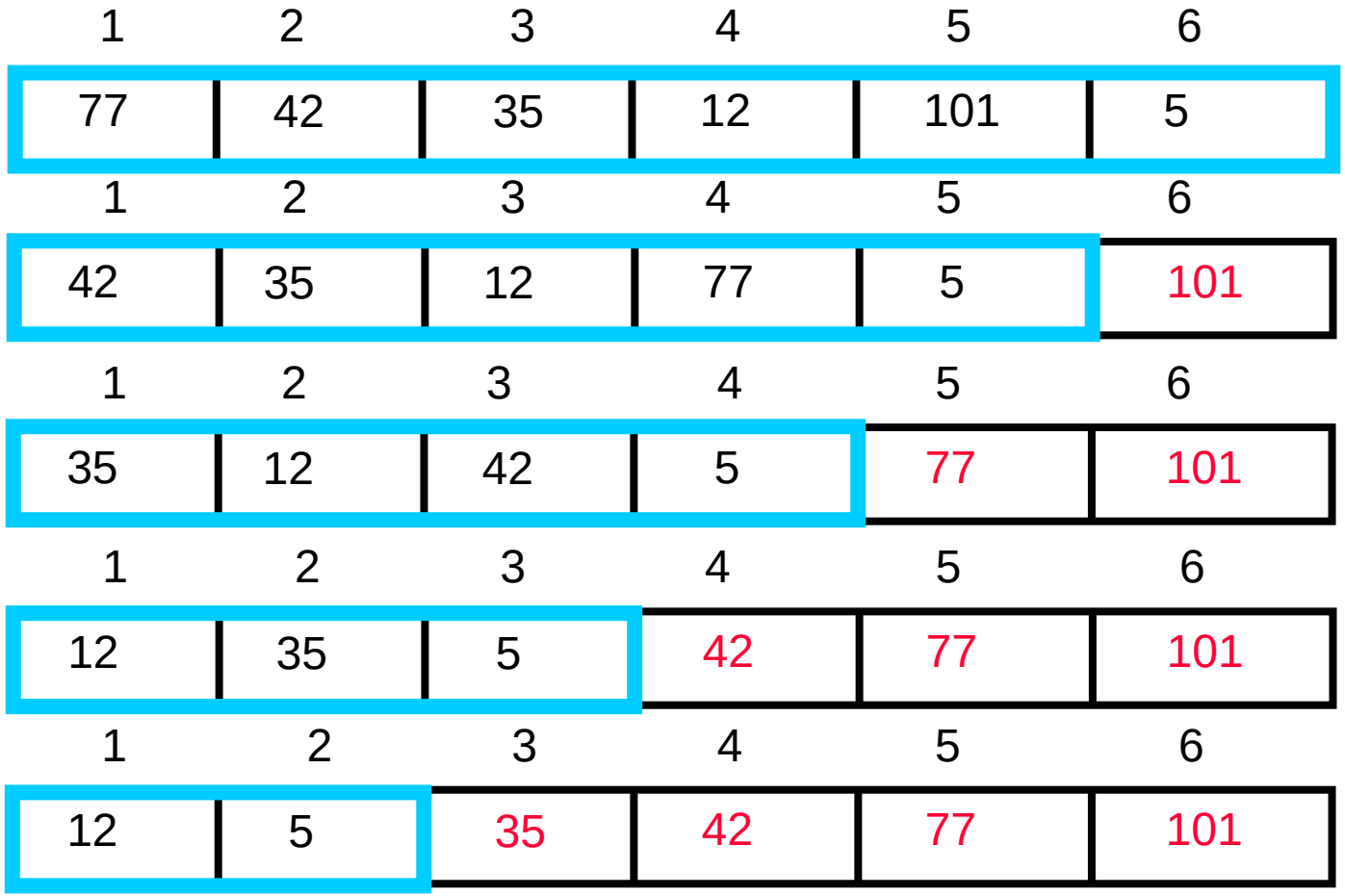
- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process



“Bubbling” All the Elements



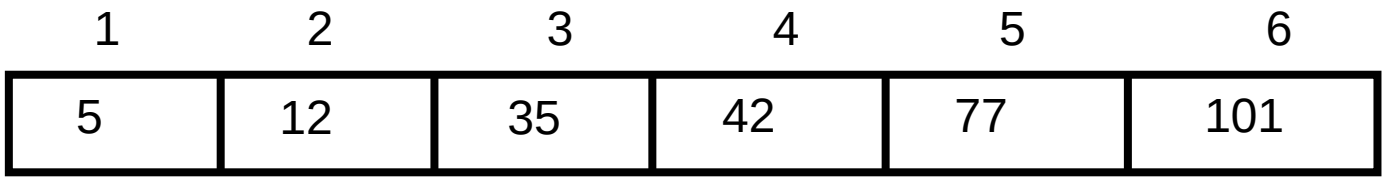
Reducing the Number of Comparisons



Already Sorted Collections?

- What if the collection was already sorted?
- What if only a few elements were out of place and after a couple of “bubble ups,” the collection was sorted?

- We want to be able to detect this and “stop early”!

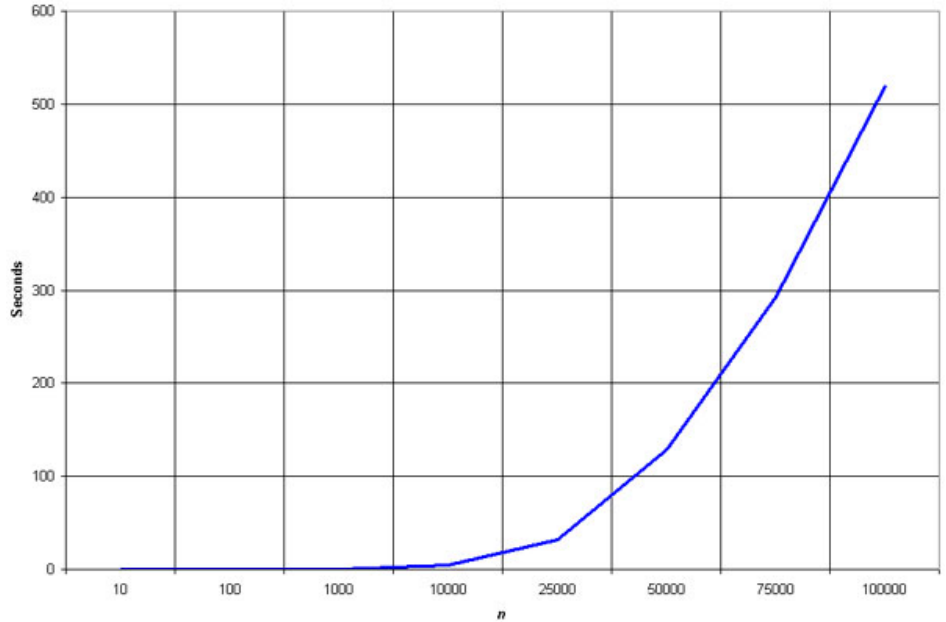


Using a Boolean “Flag”

- We can use a boolean variable to determine if any swapping occurred during the “bubble up.”
- If no swapping occurred, then we know that the collection is already sorted!
- This boolean “flag” needs to be reset after each “bubble up.”

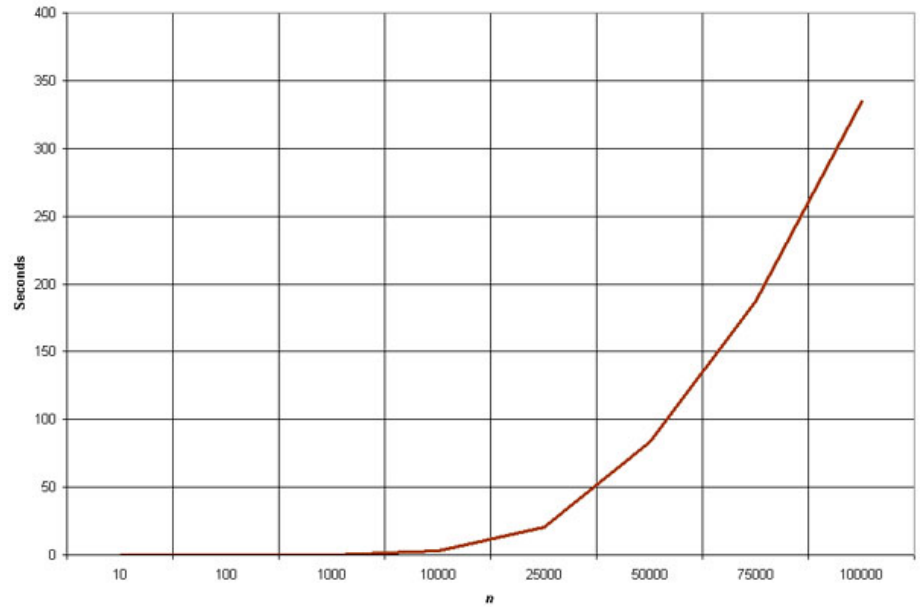
Bubble Sort Algorithm

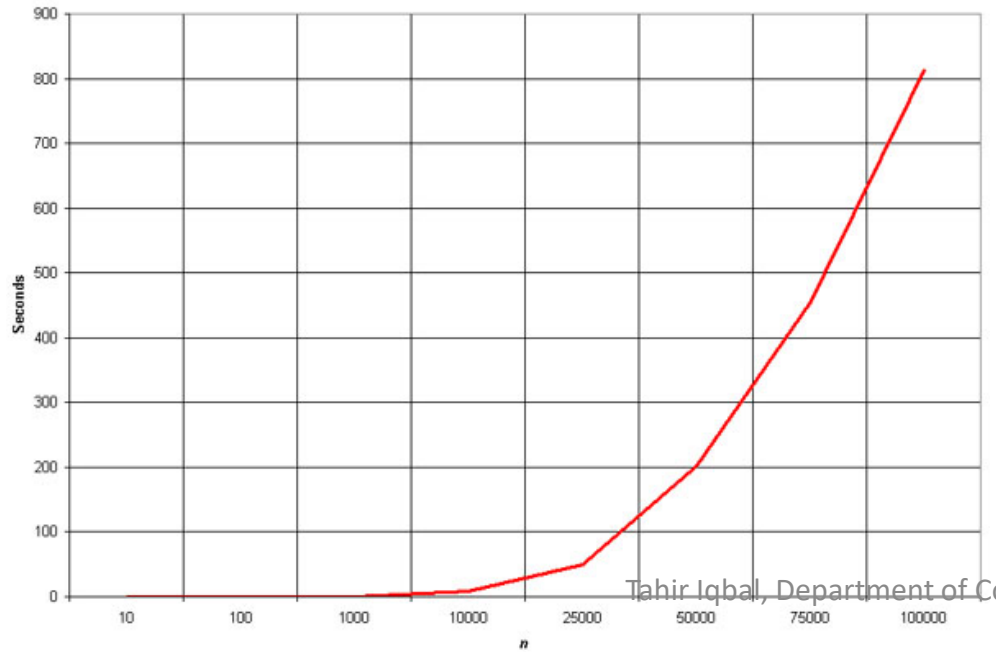
```
void bubbleSort (int S[ ],  int length) {
    bool isSorted = false;
    while(!isSorted)
    {
            isSorted = true;
            for(int i = 0; i<length; i++)
            {
                if(S[i] > S[i+1])
                {
                        int temp = S[i];
                        S[i] = S[i+1];
                                S[i+1] = temp;
                        isSorted = false;
                 }
            }
            length--;
}
}
```



Selection Sort

Insertion Sort





Bubble Sort

Summary

The insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less.

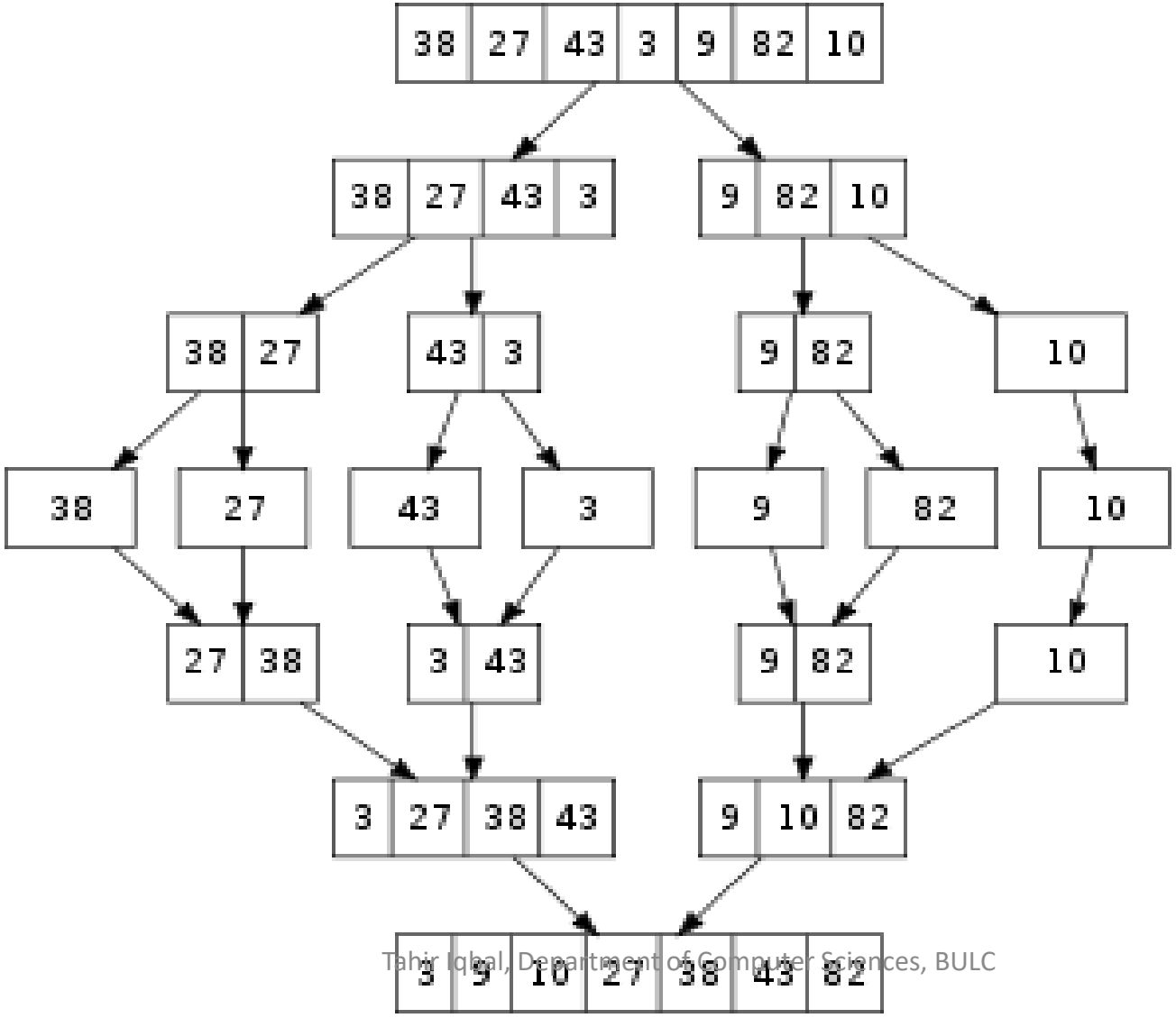
The insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items.

Mergesort

Divide and Conquer

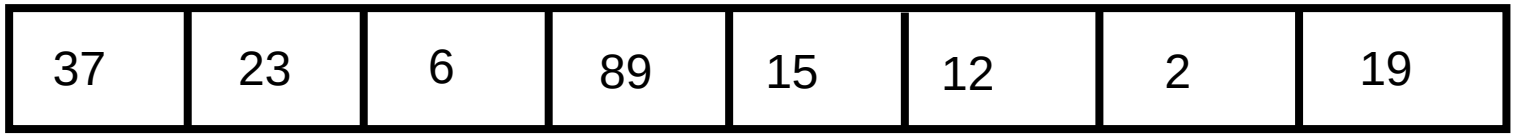
- **Divide and Conquer cuts the problem in half each time, but uses the result of both halves:**
  - **cut the problem in half until the problem is trivial**
  - **solve for both halves**
  - **combine the solutions**

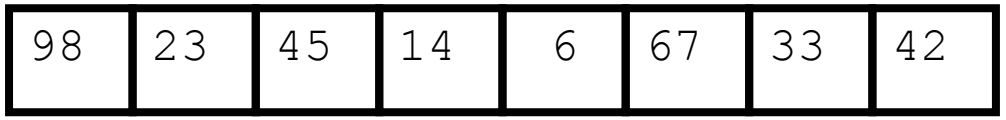
Merge Sort

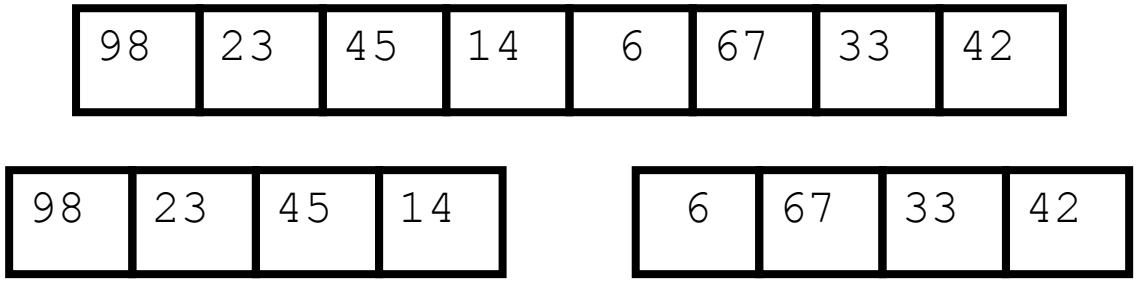

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

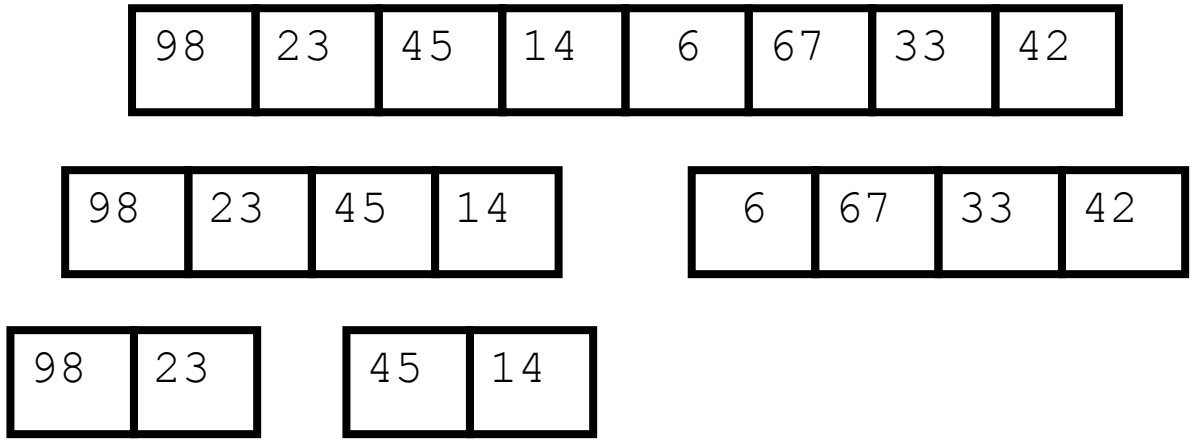
Mergesort

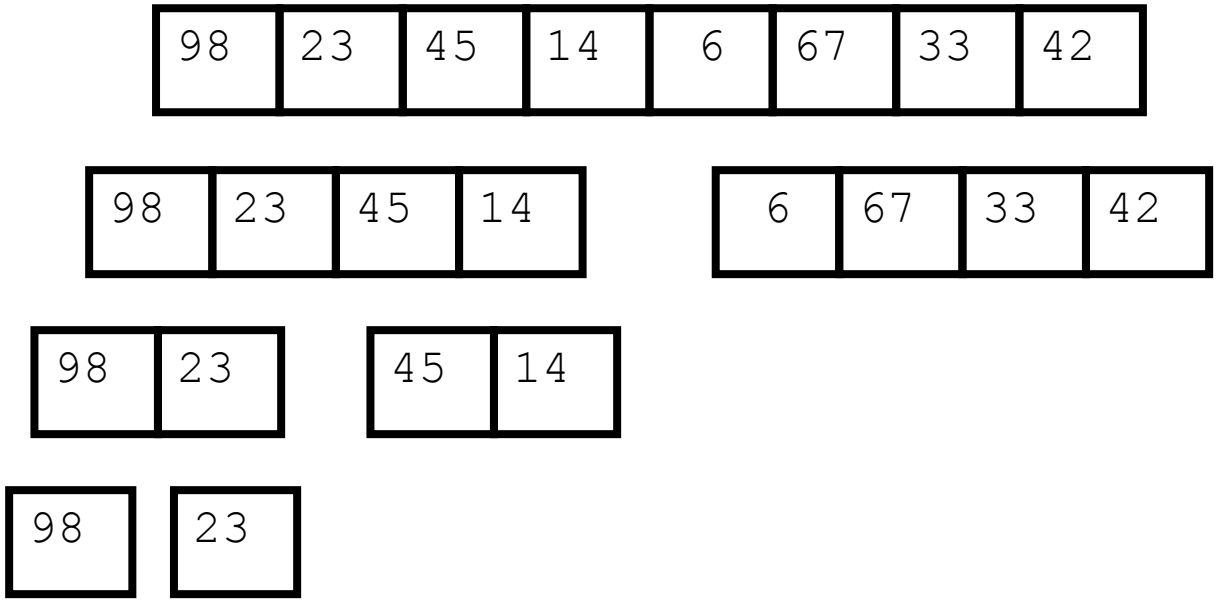
- **A divide-and-conquer algorithm:**
- **Divide the unsorted array into 2 halves until the sub-arrays only contain one element**
- **Merge the sub-problem solutions together:**
  - **Compare the sub-array's first elements**
  - **Remove the smallest element and put it into the result array**
  - **Continue the process until all elements have been put into the result array**

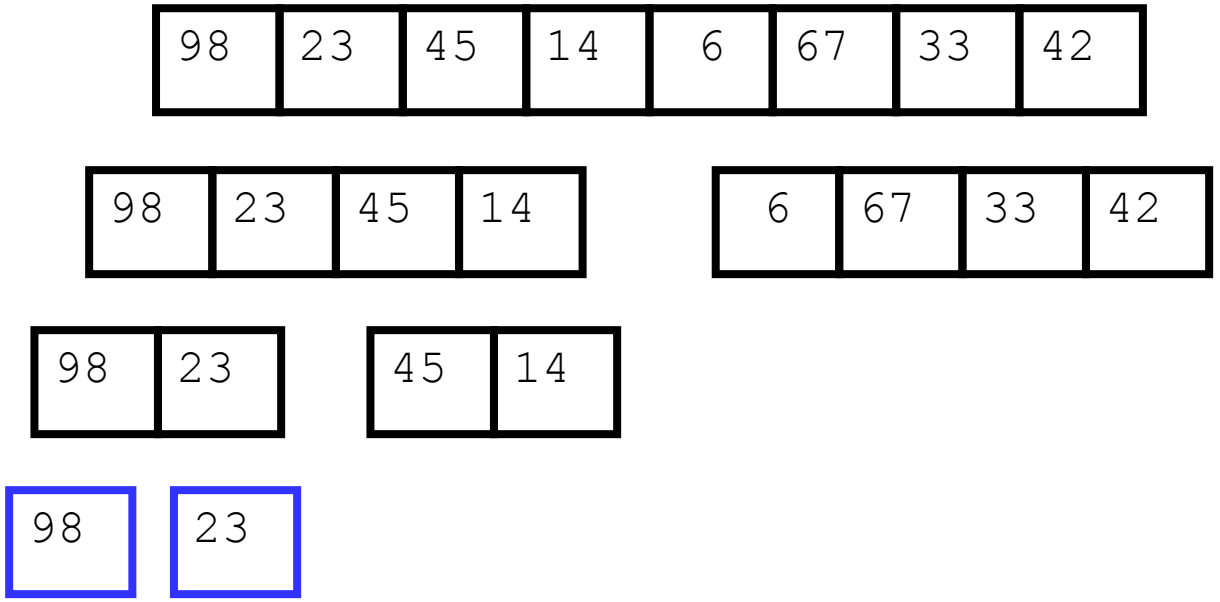


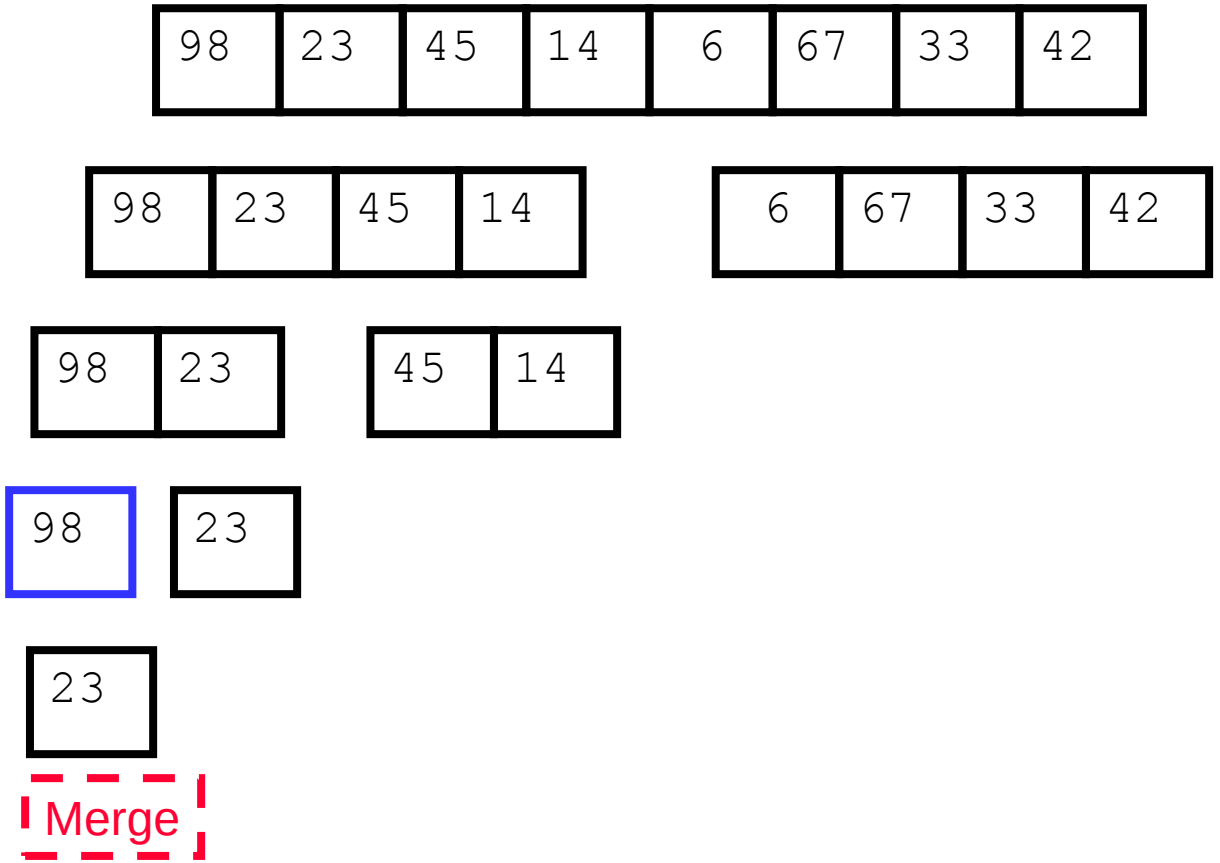

98
23
45
14
6
67
33
42

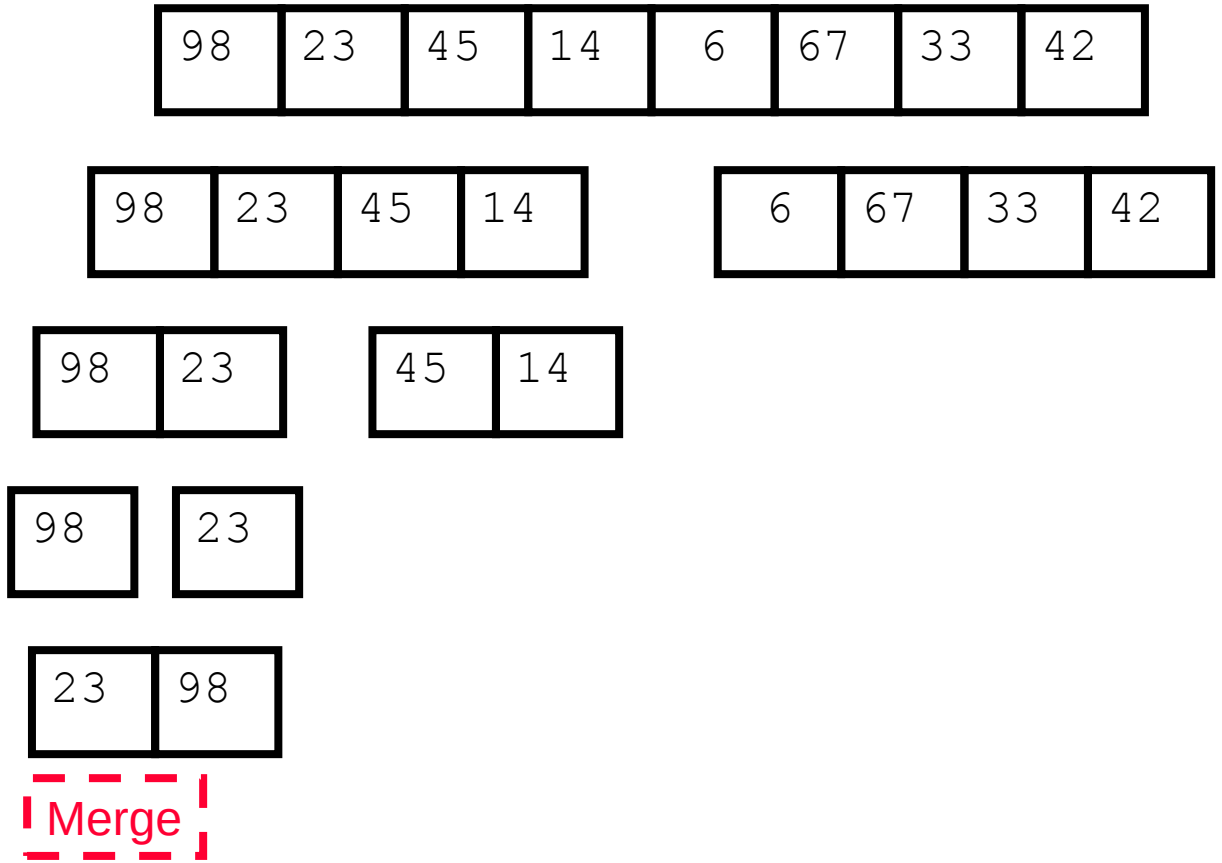

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42

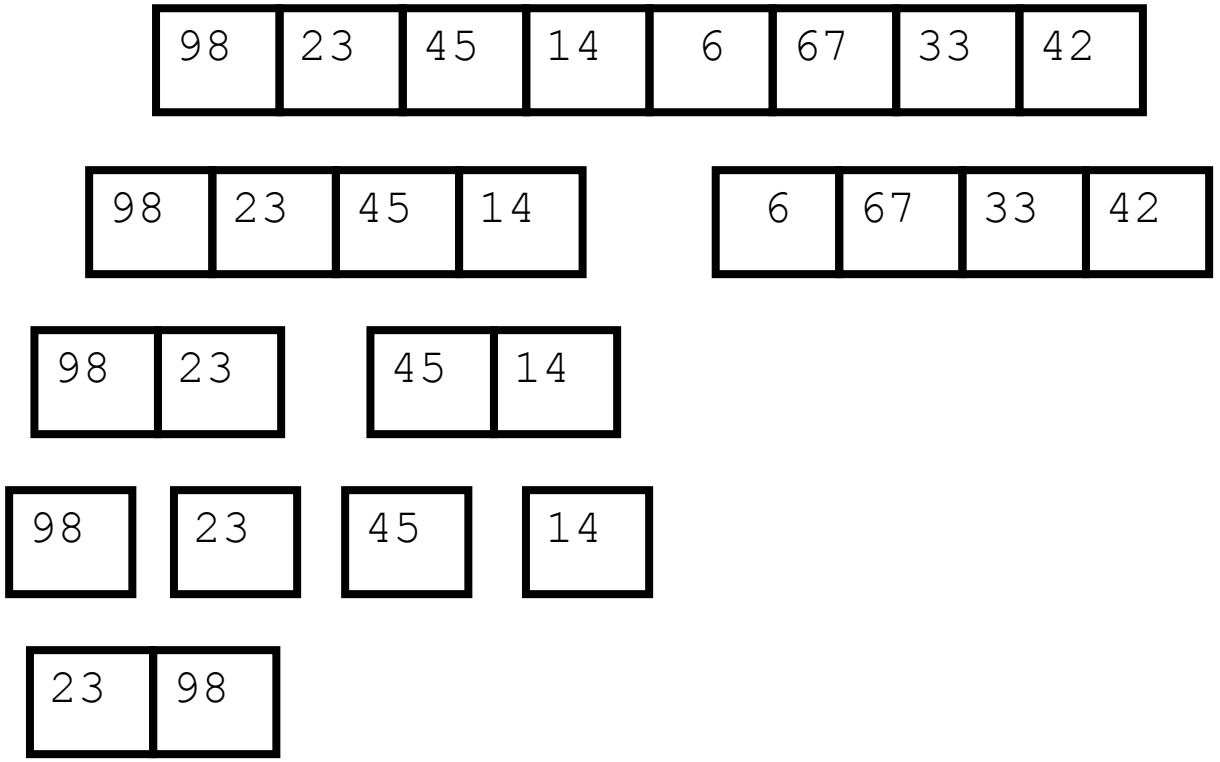

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

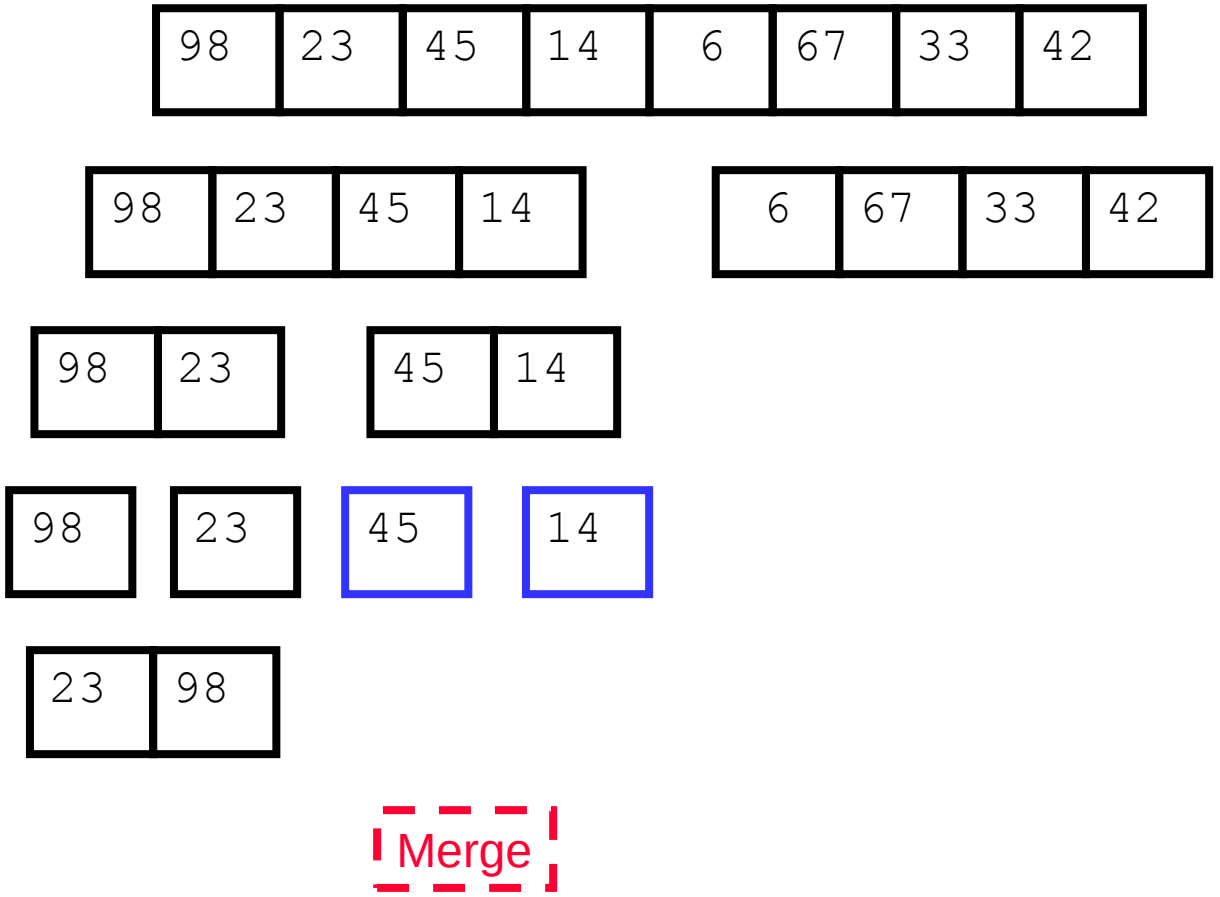

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

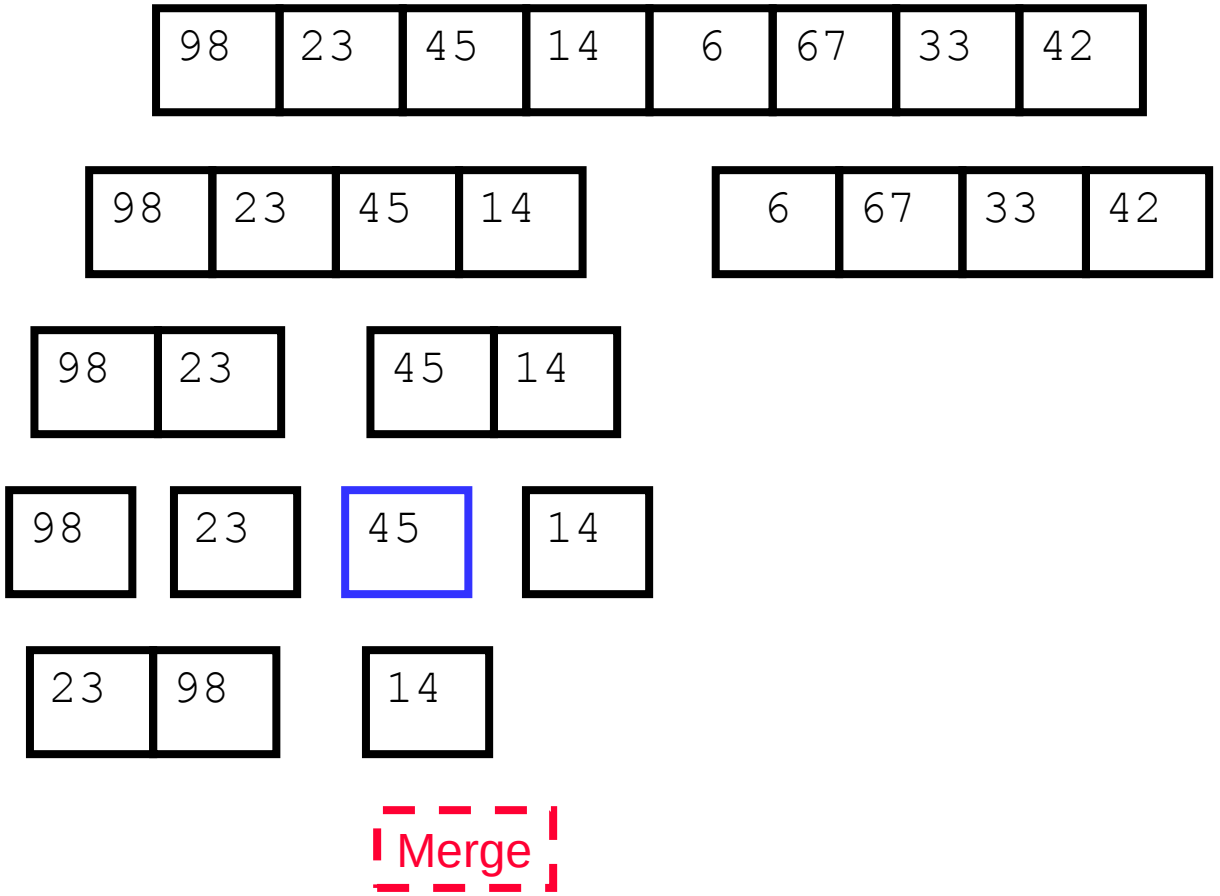

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

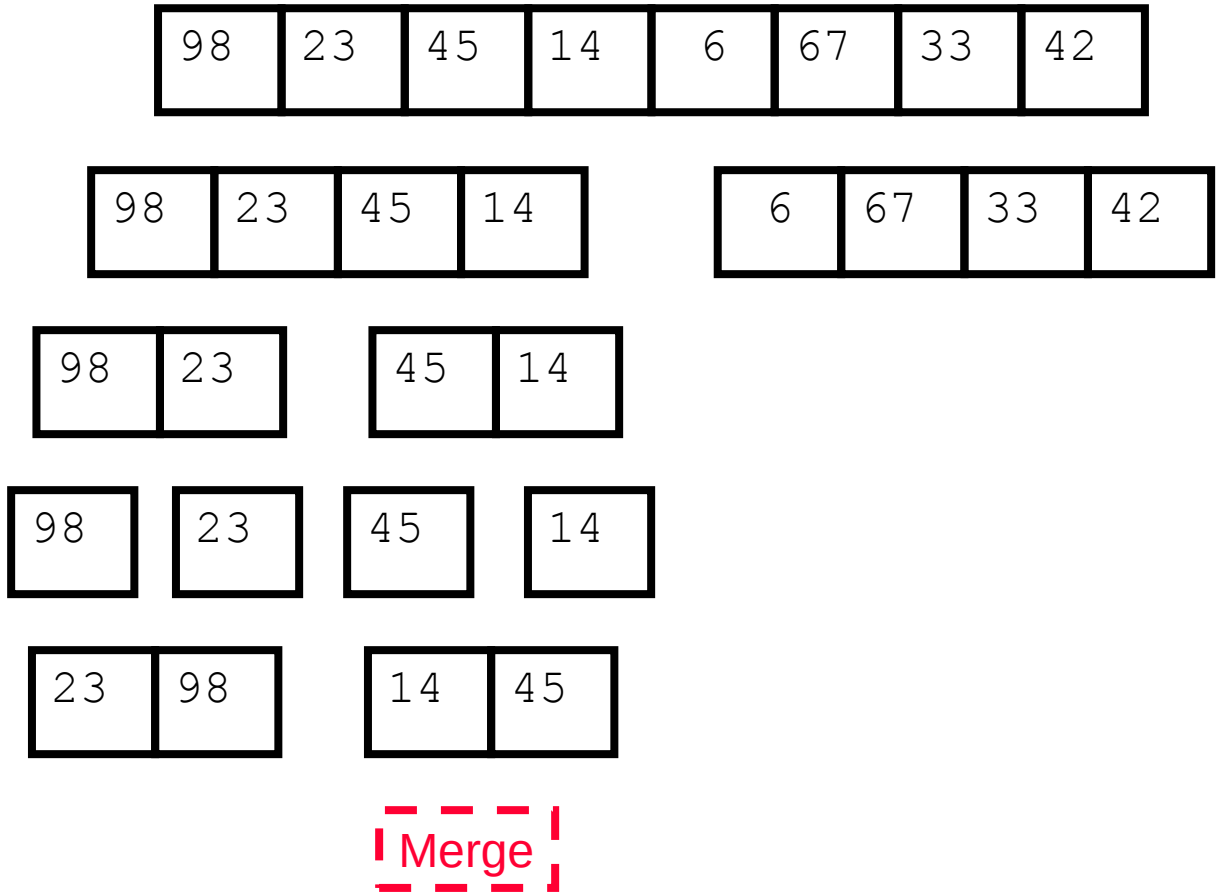

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

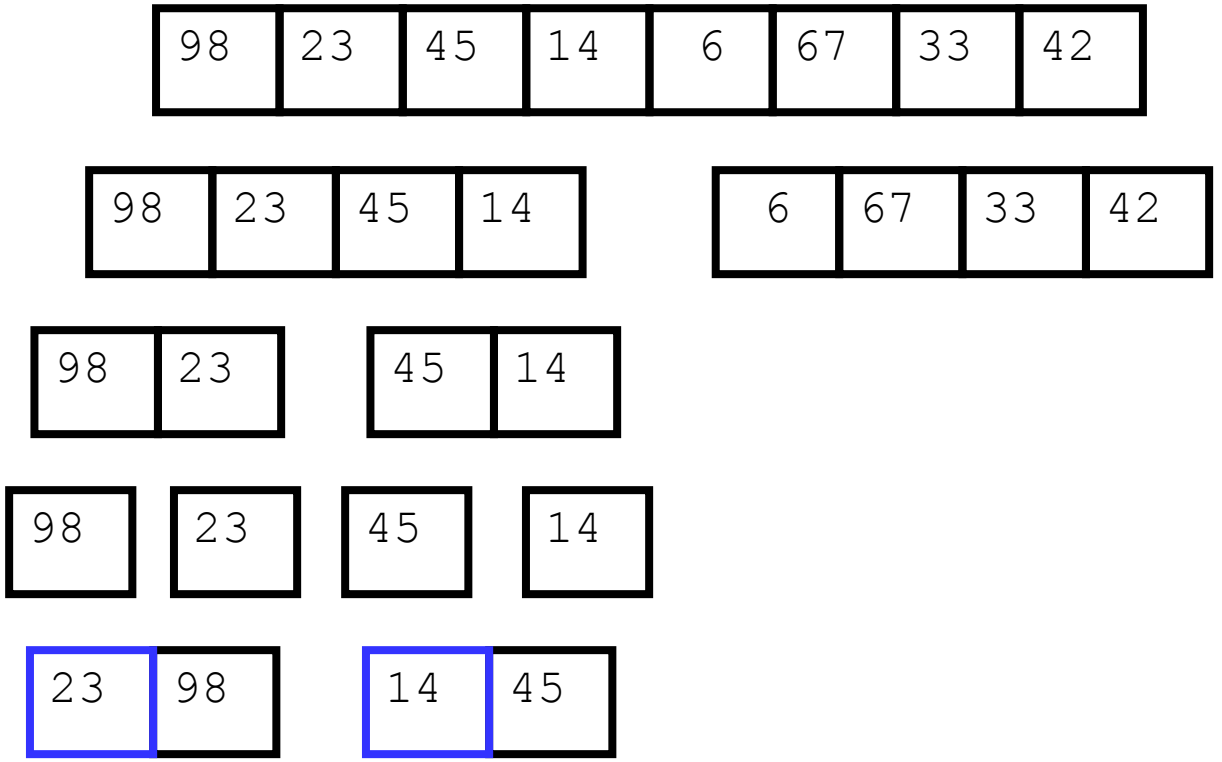

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge

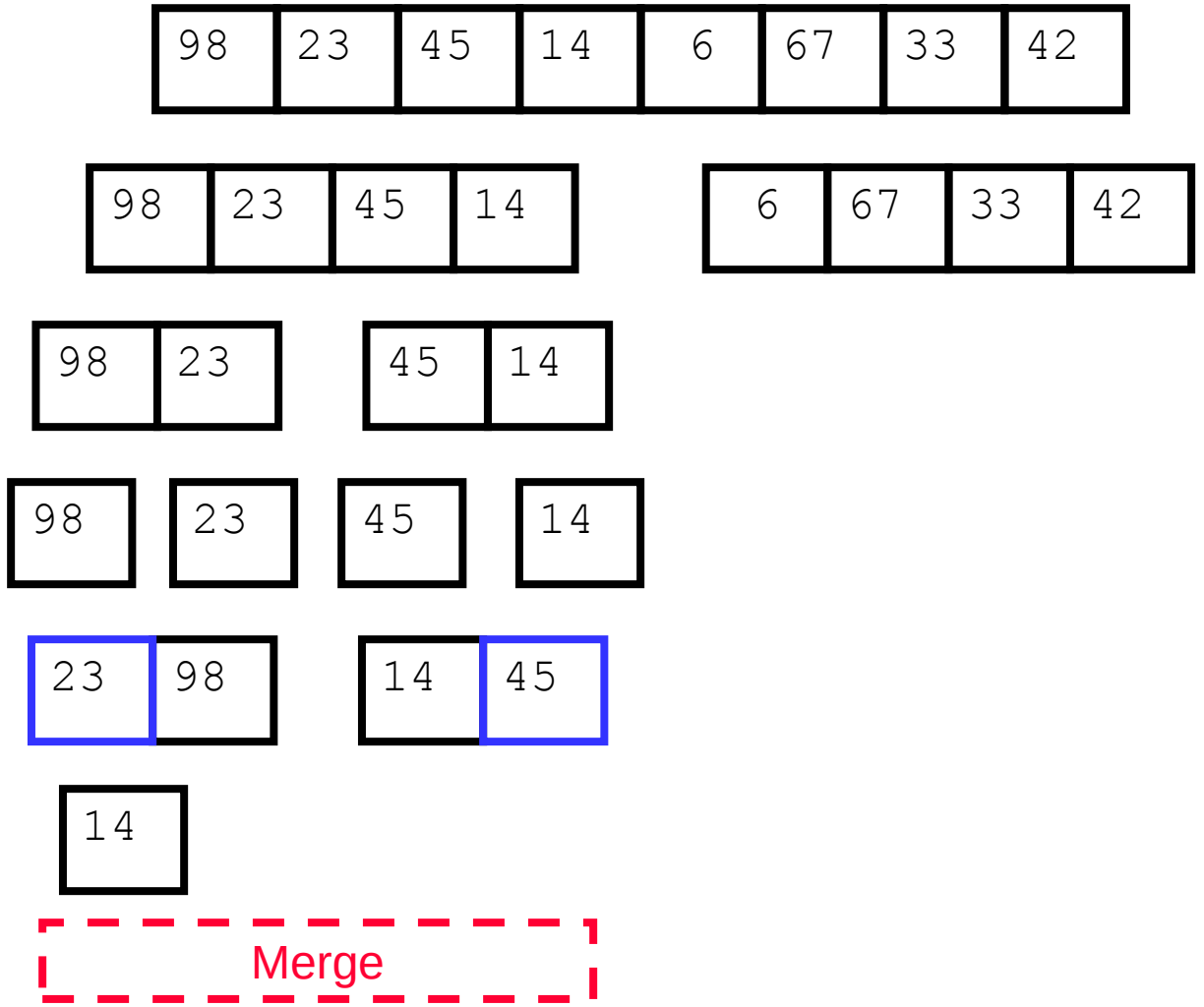

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98

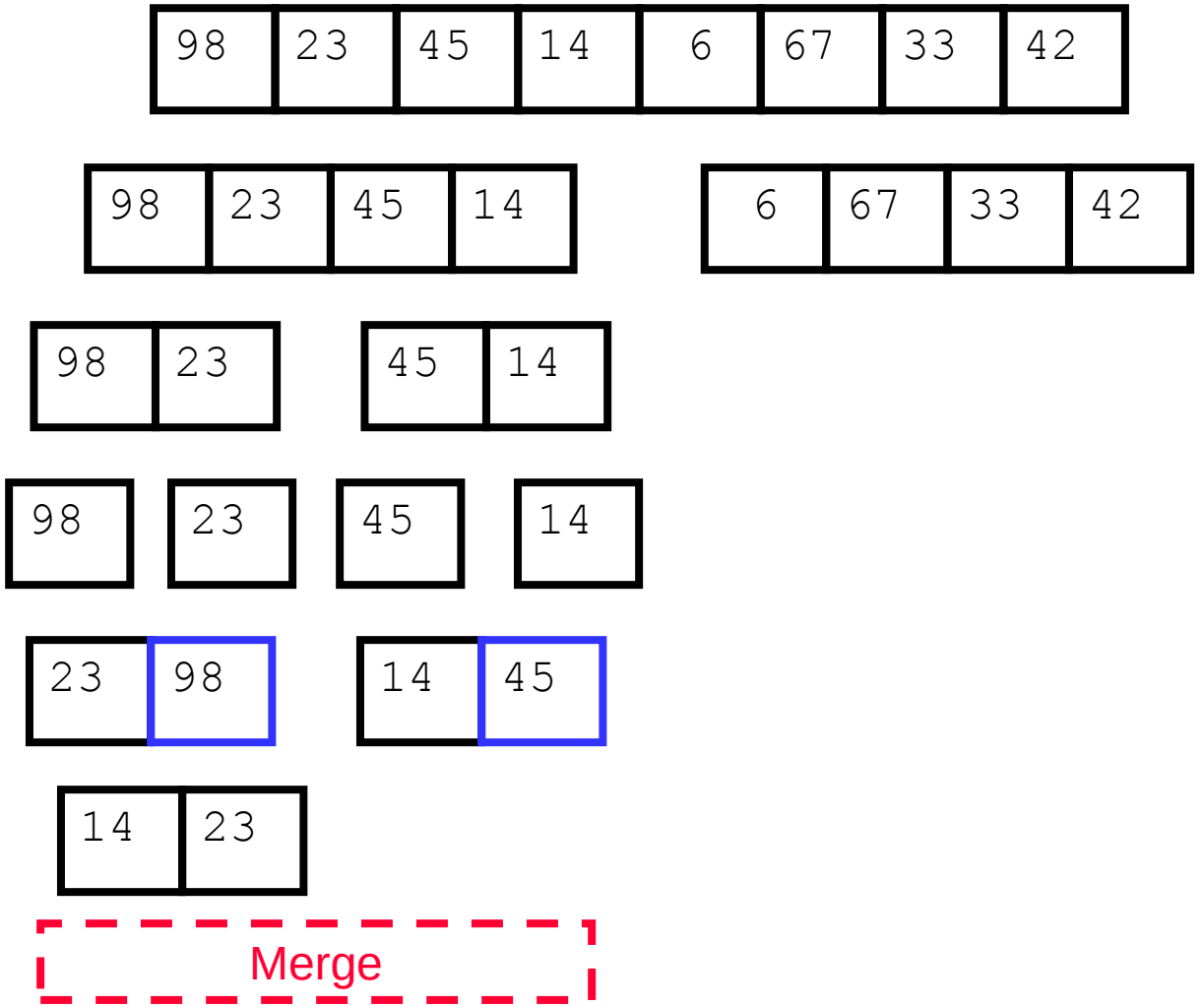

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
Merge

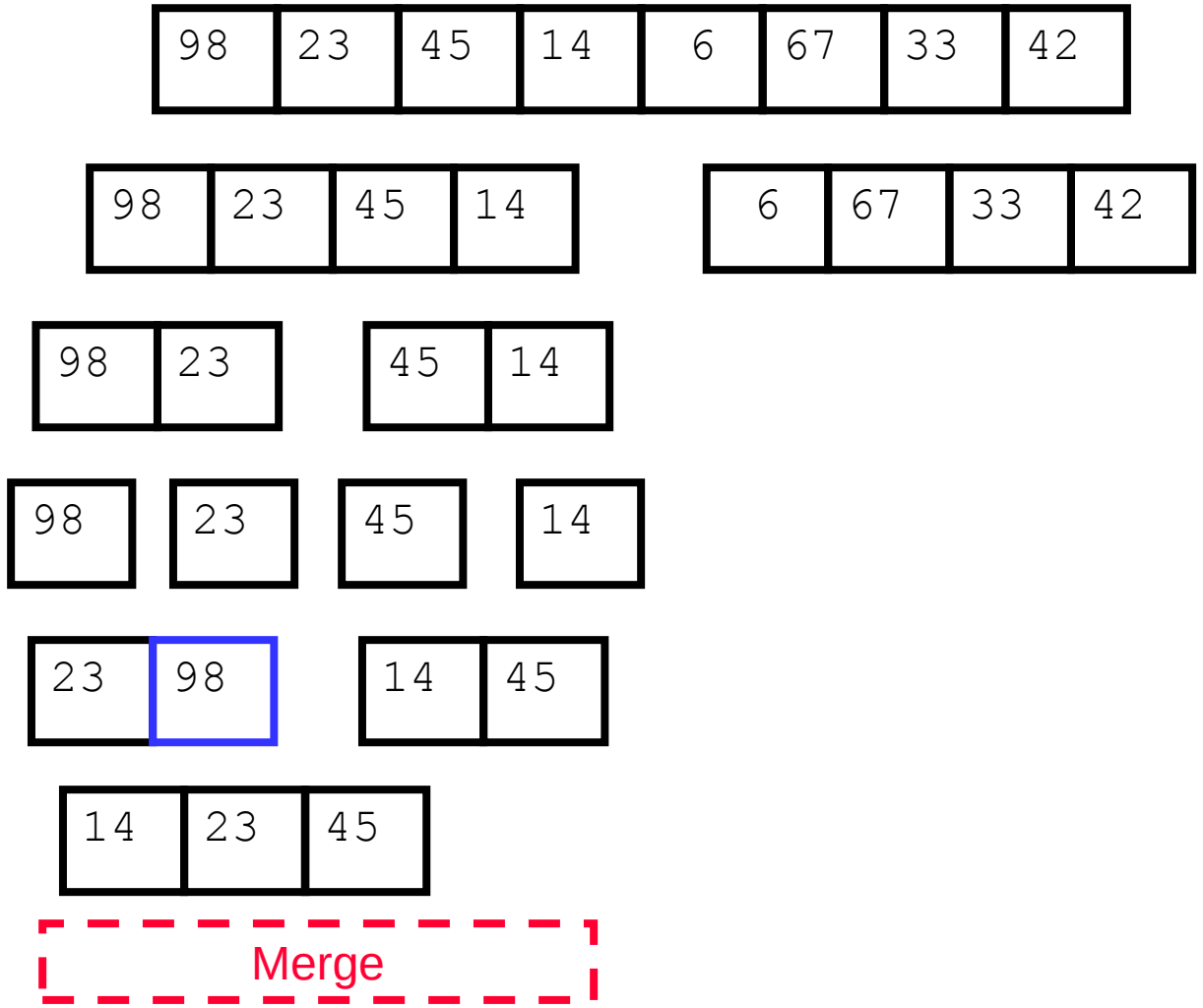

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14
Merge

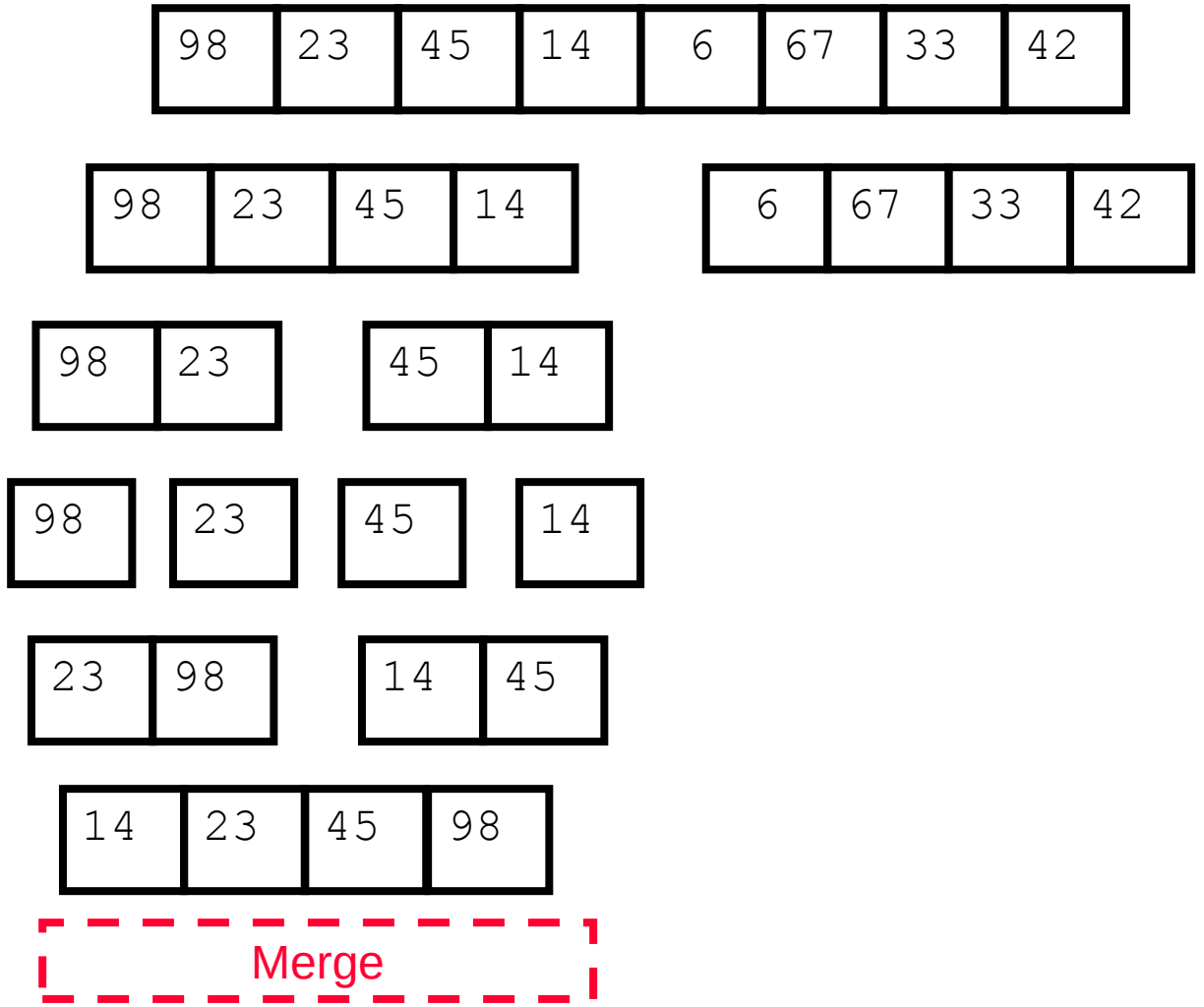

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

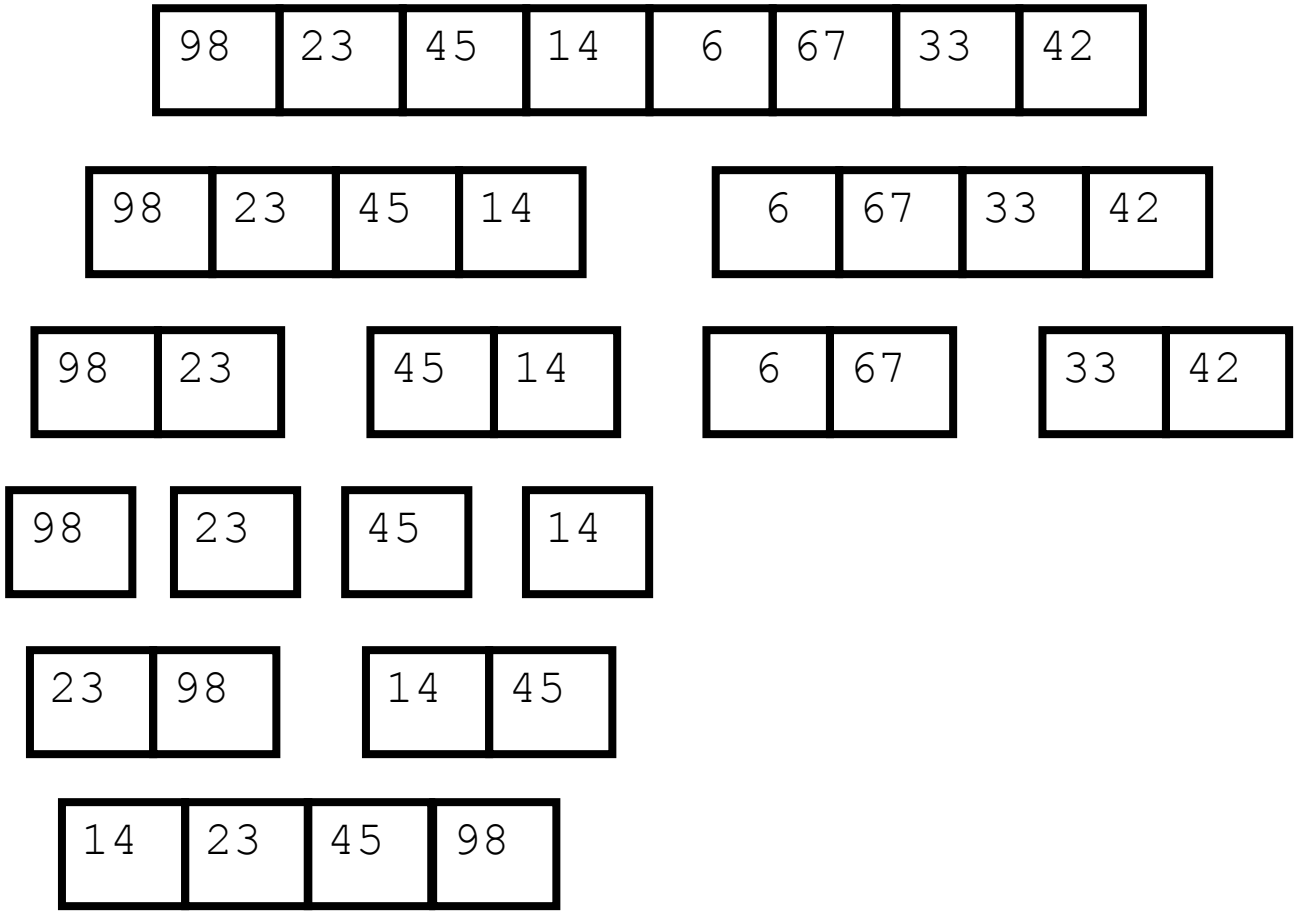

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

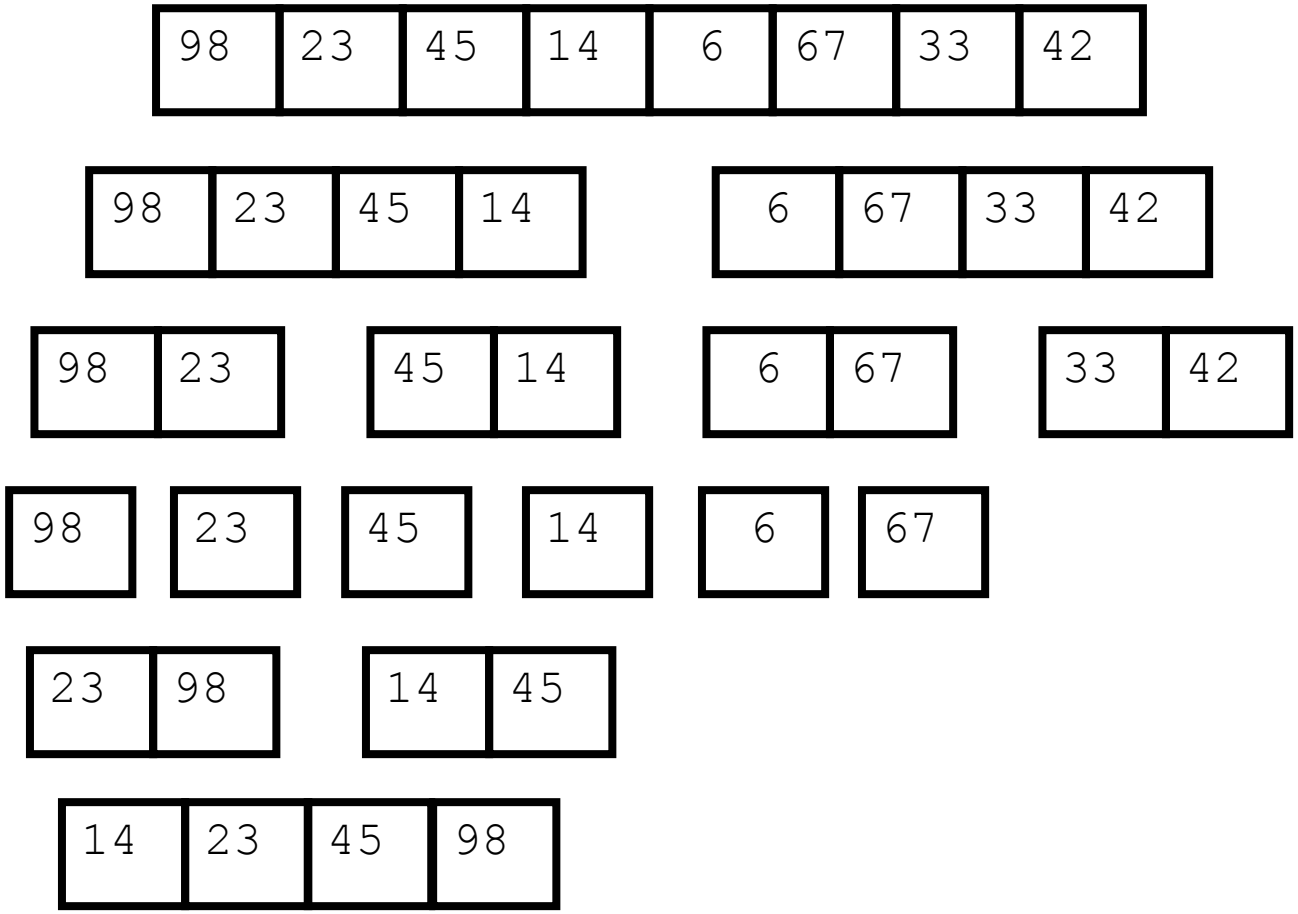

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

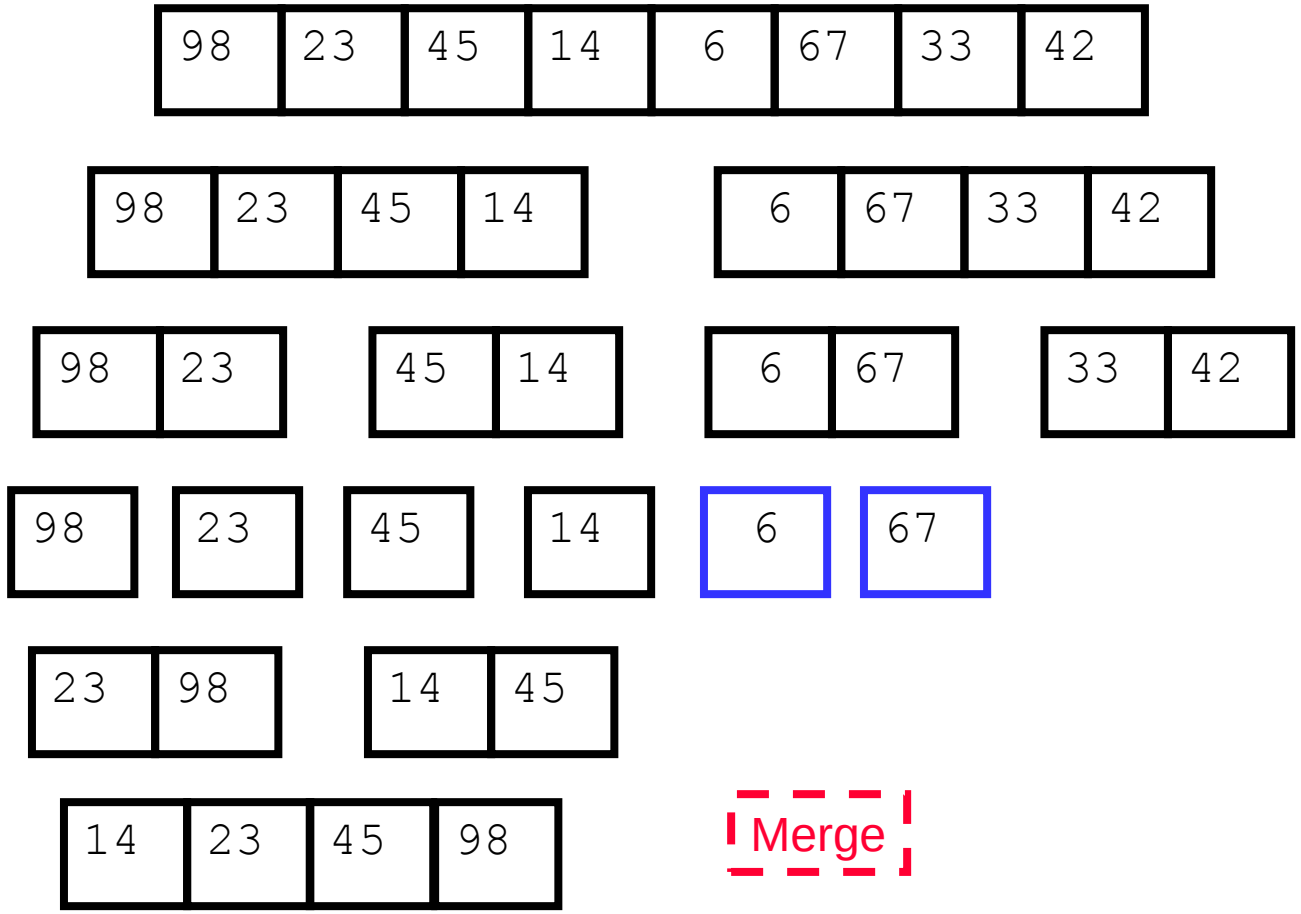

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

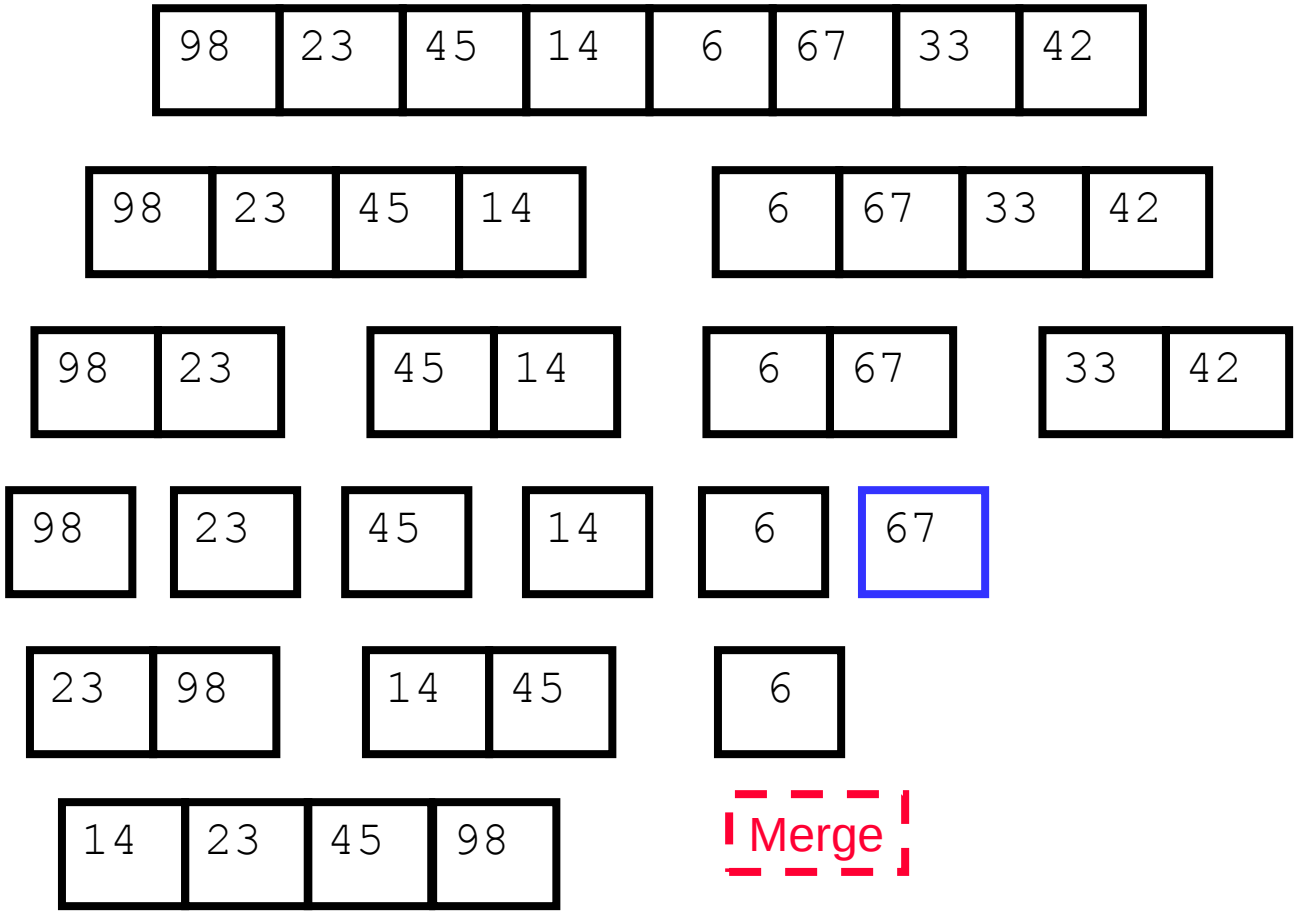

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

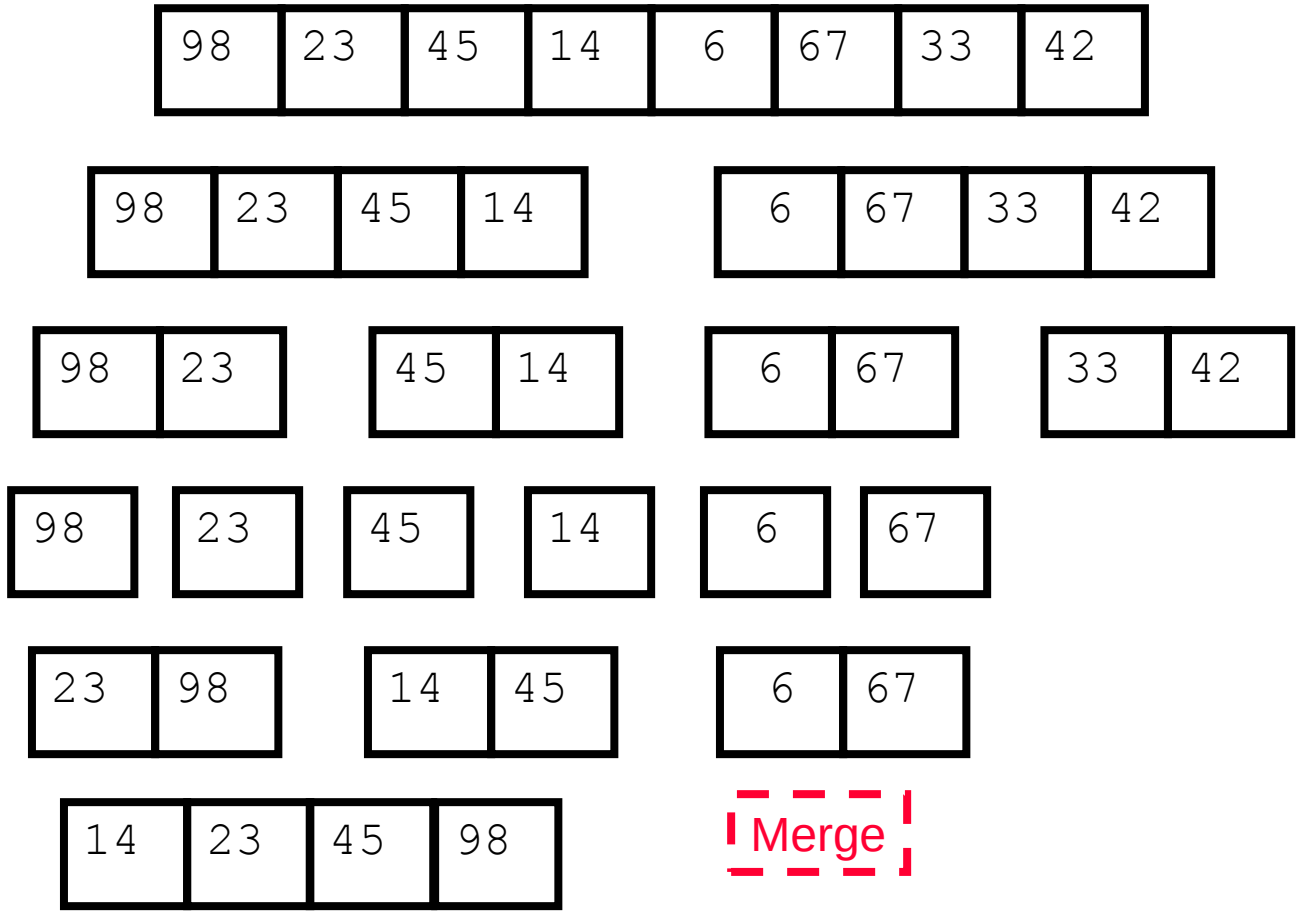

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

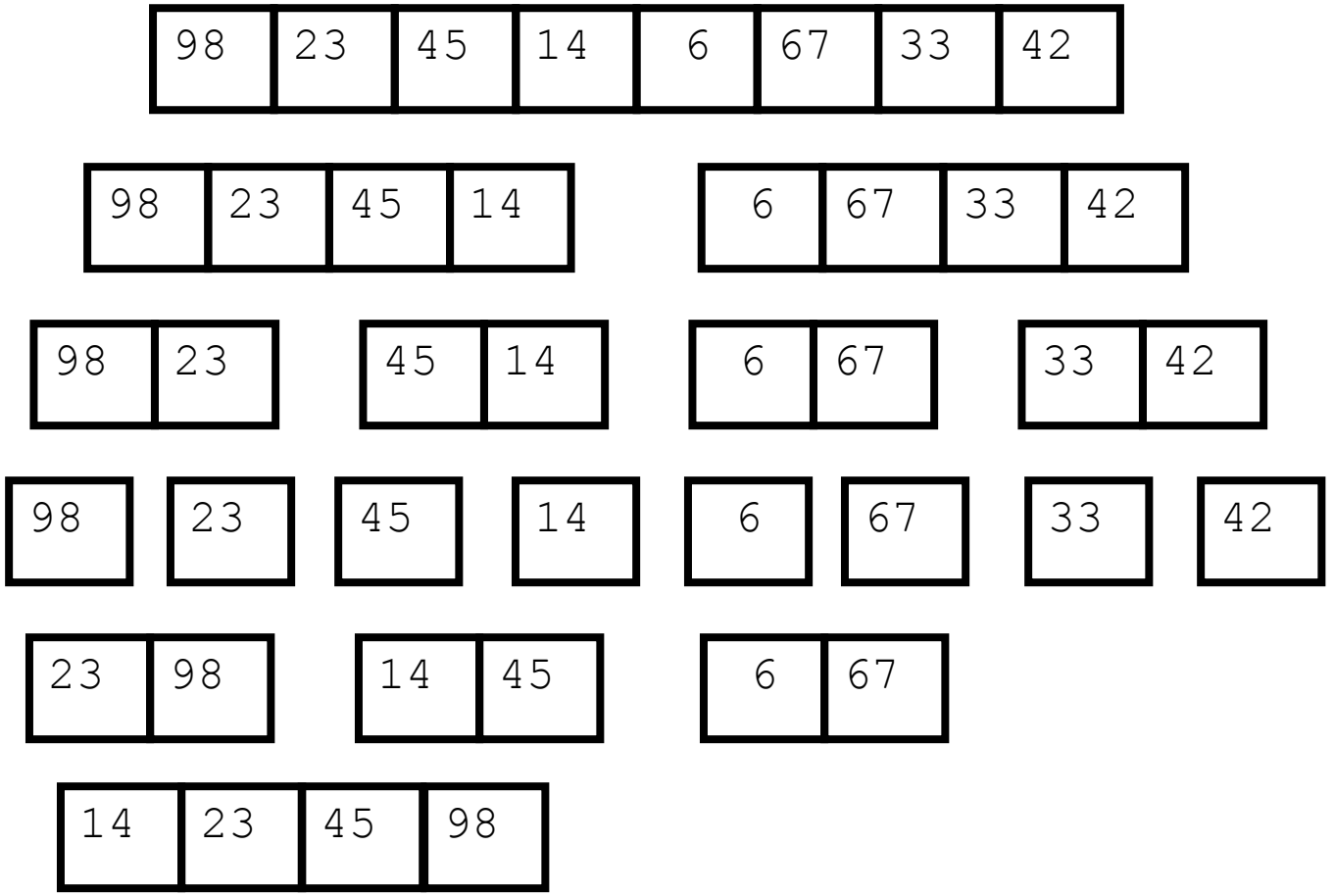

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

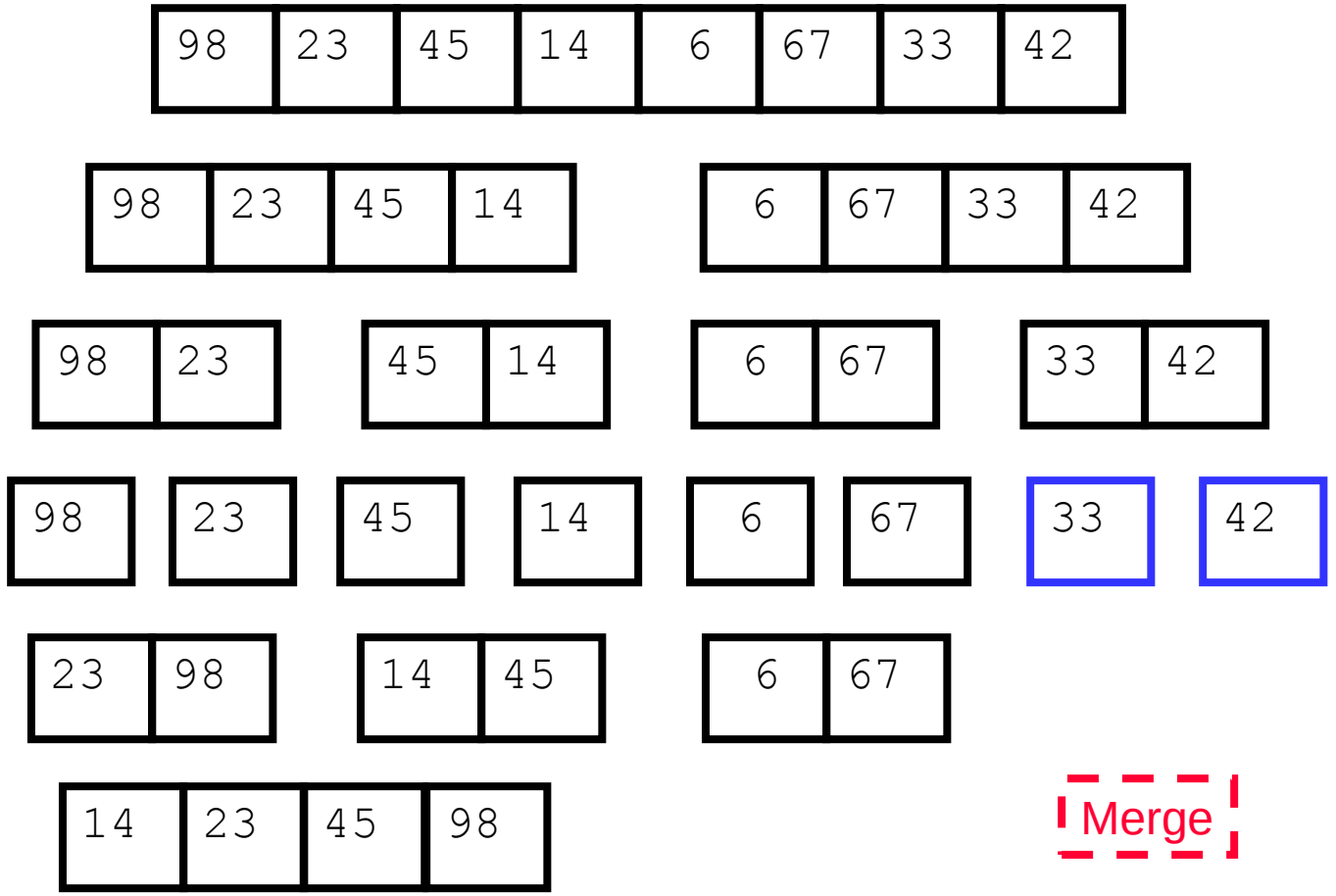

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

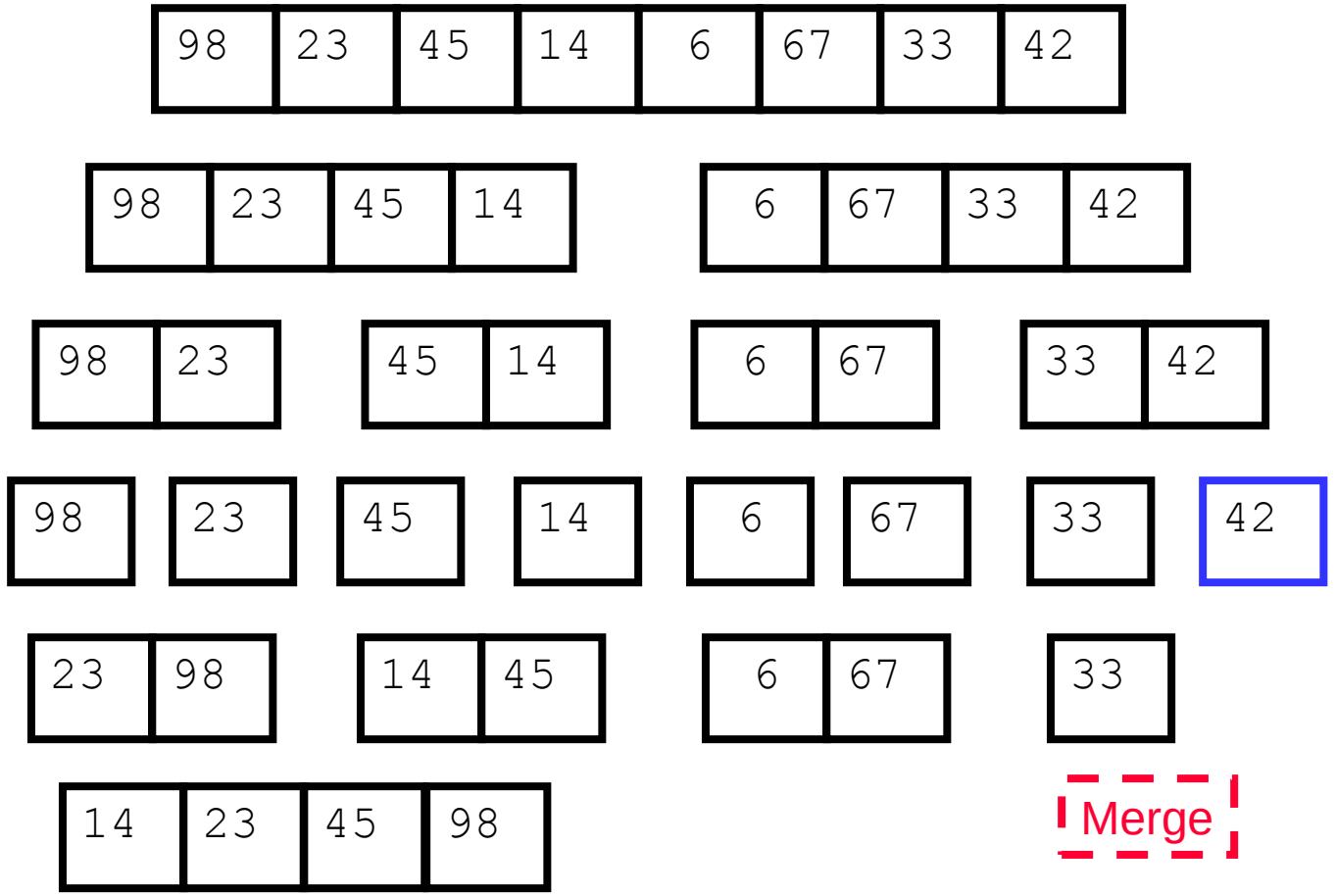

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

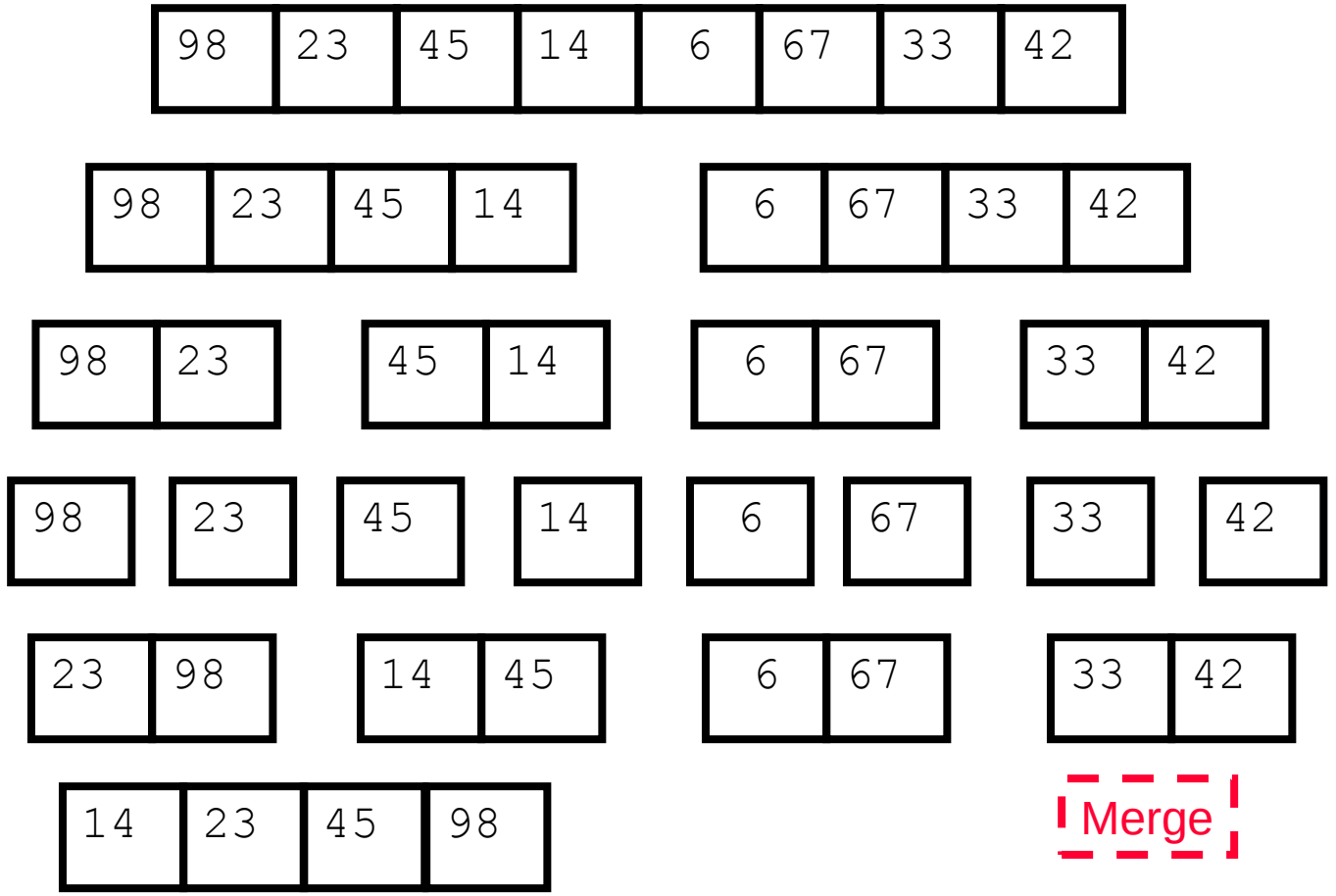

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge

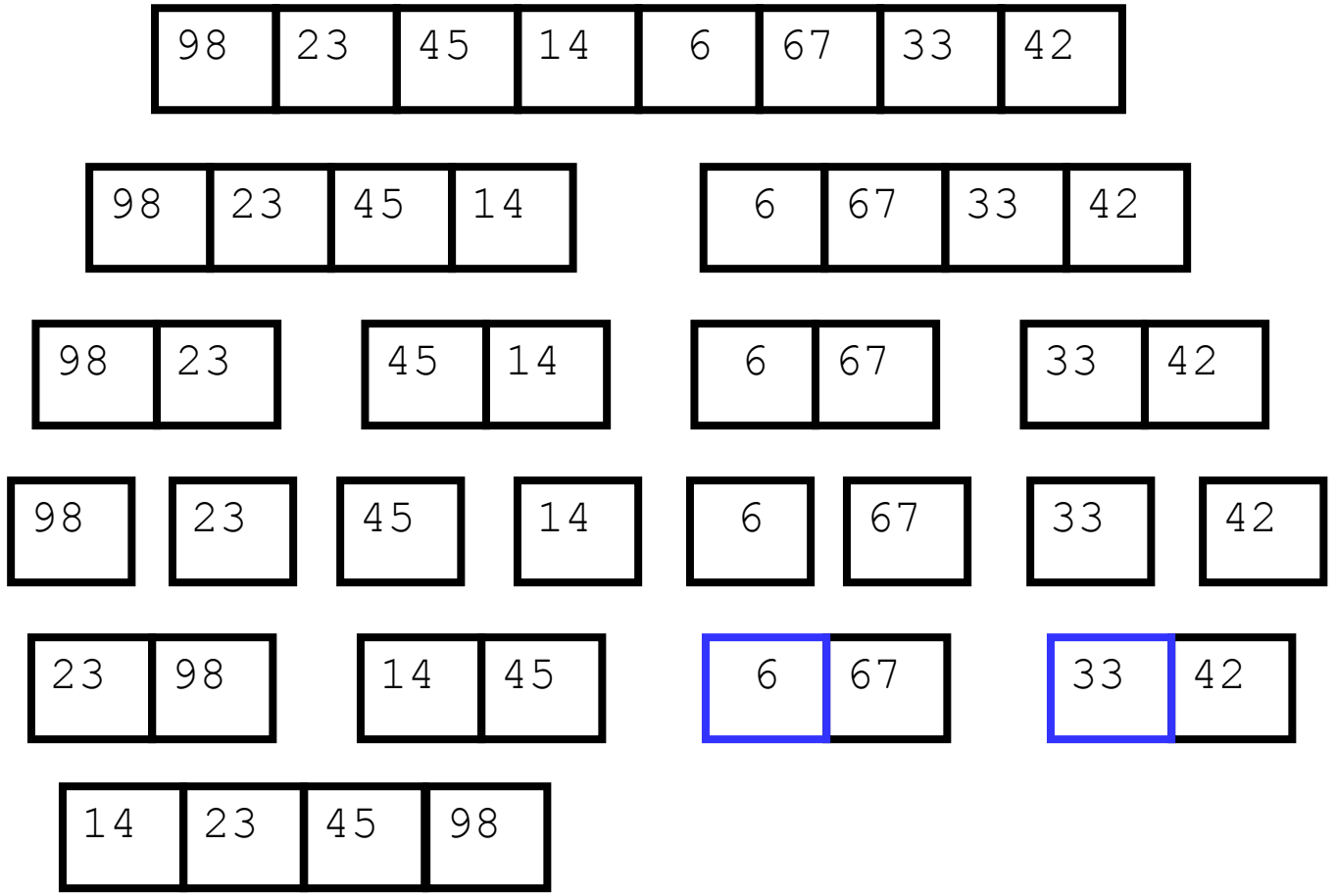

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6 67
14 23 45 98
Merge

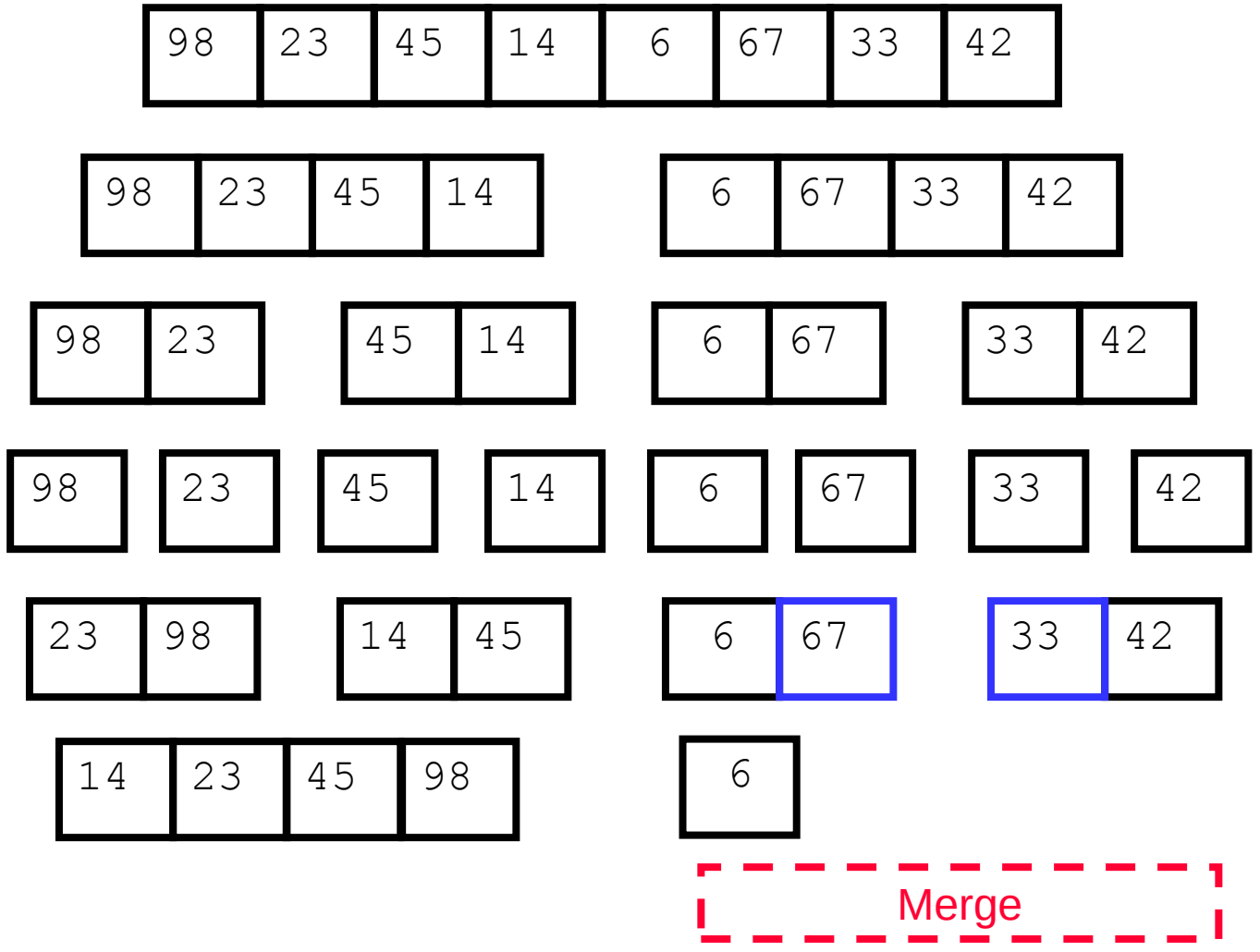

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98

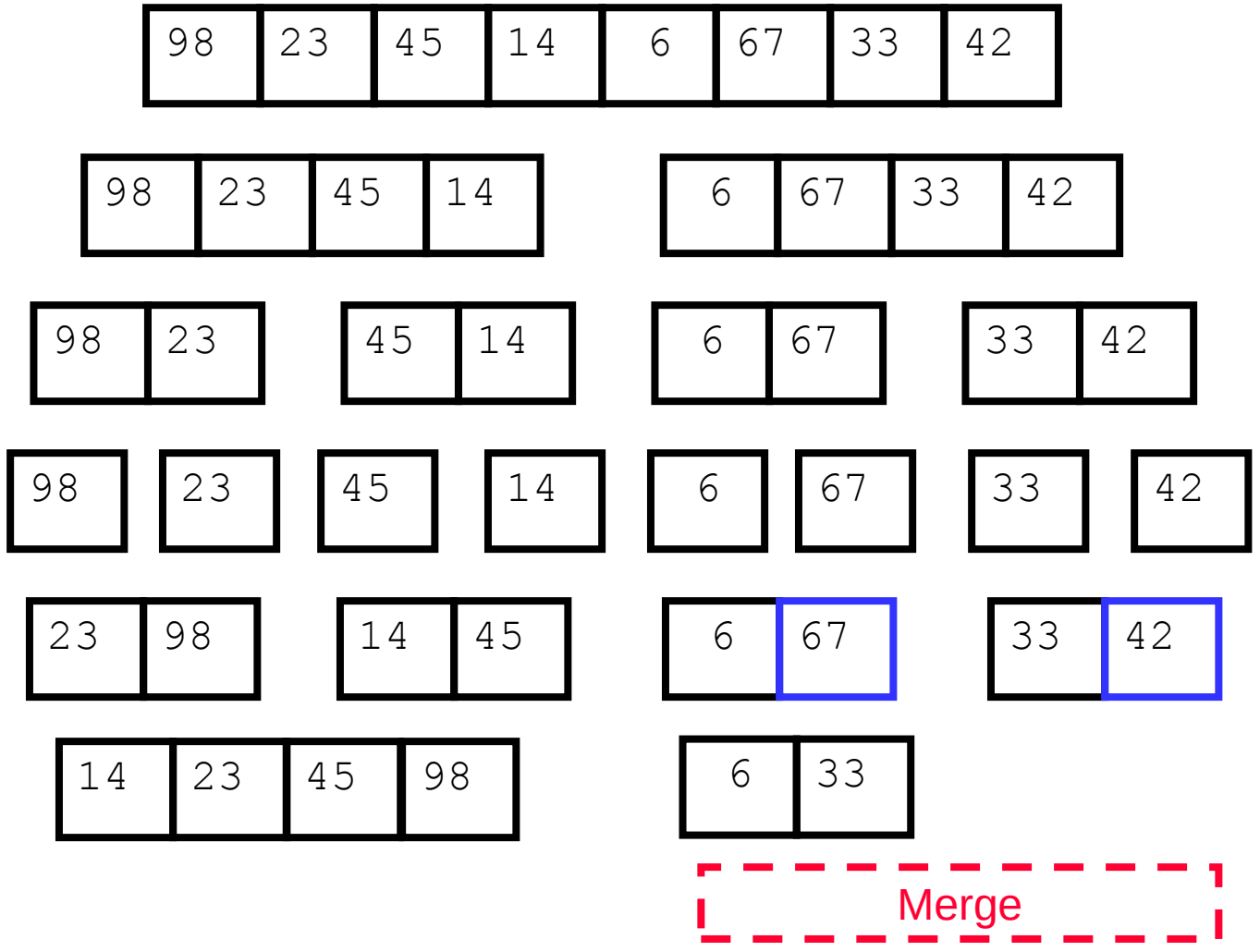

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98
Merge

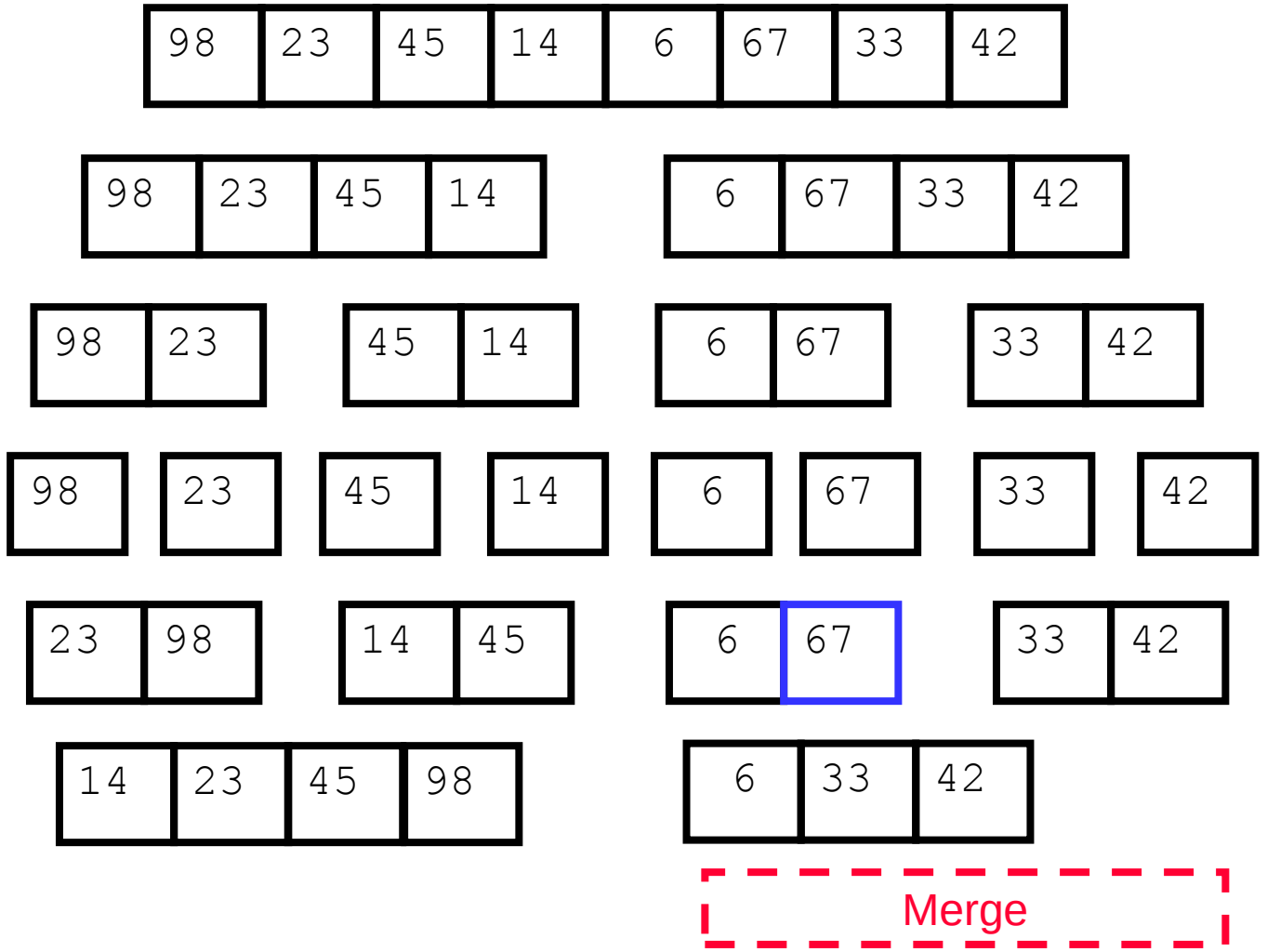

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge

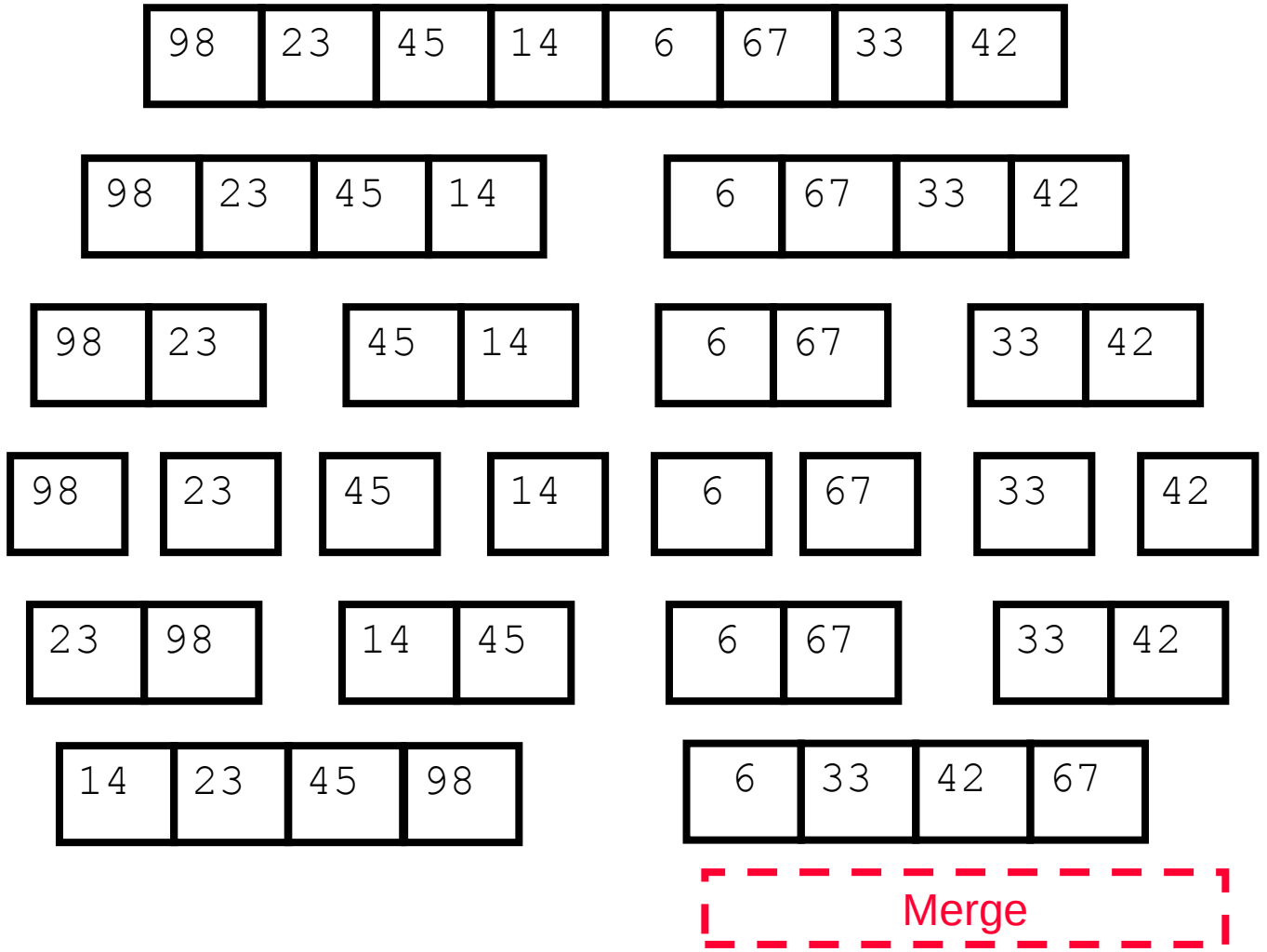

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

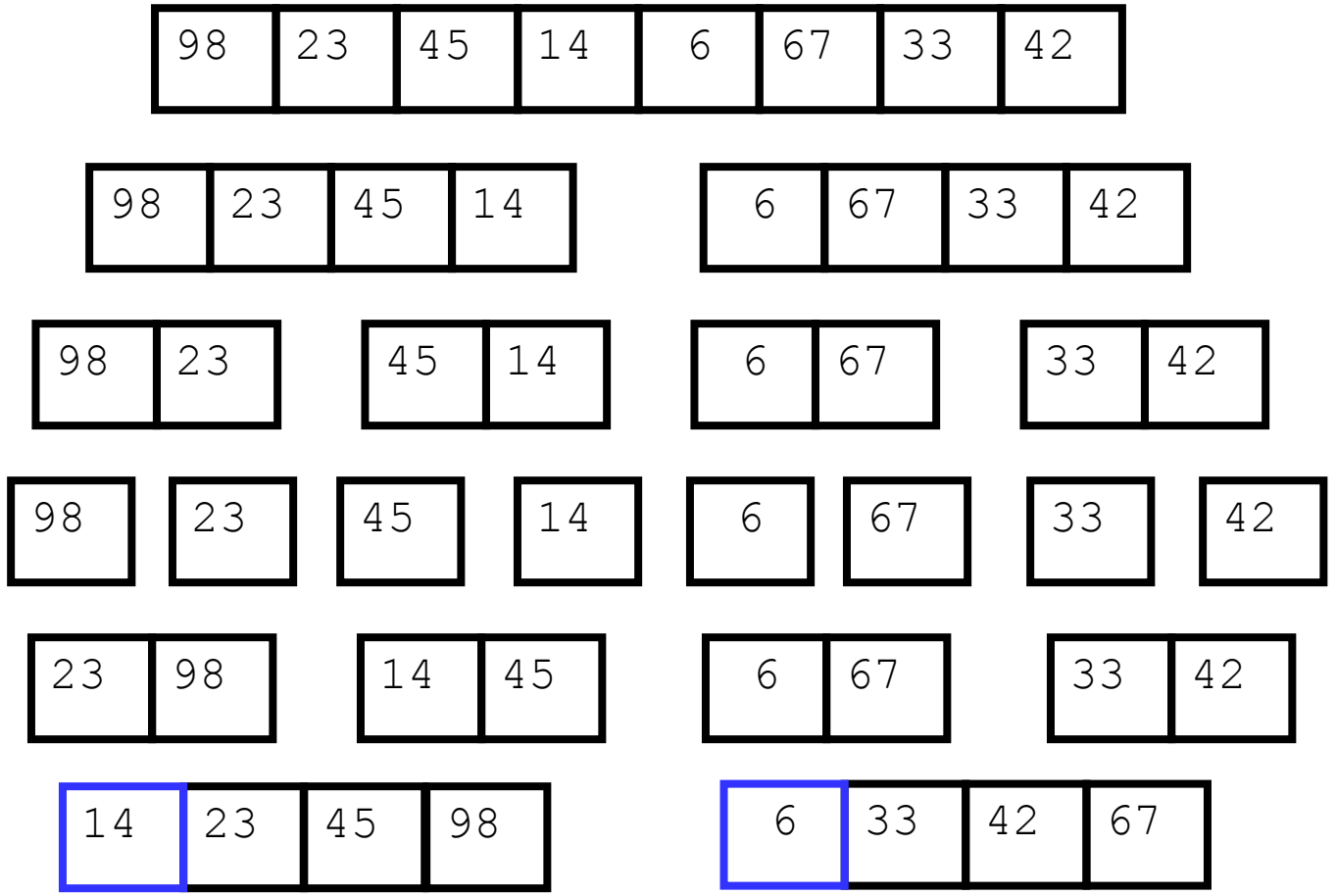

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge


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

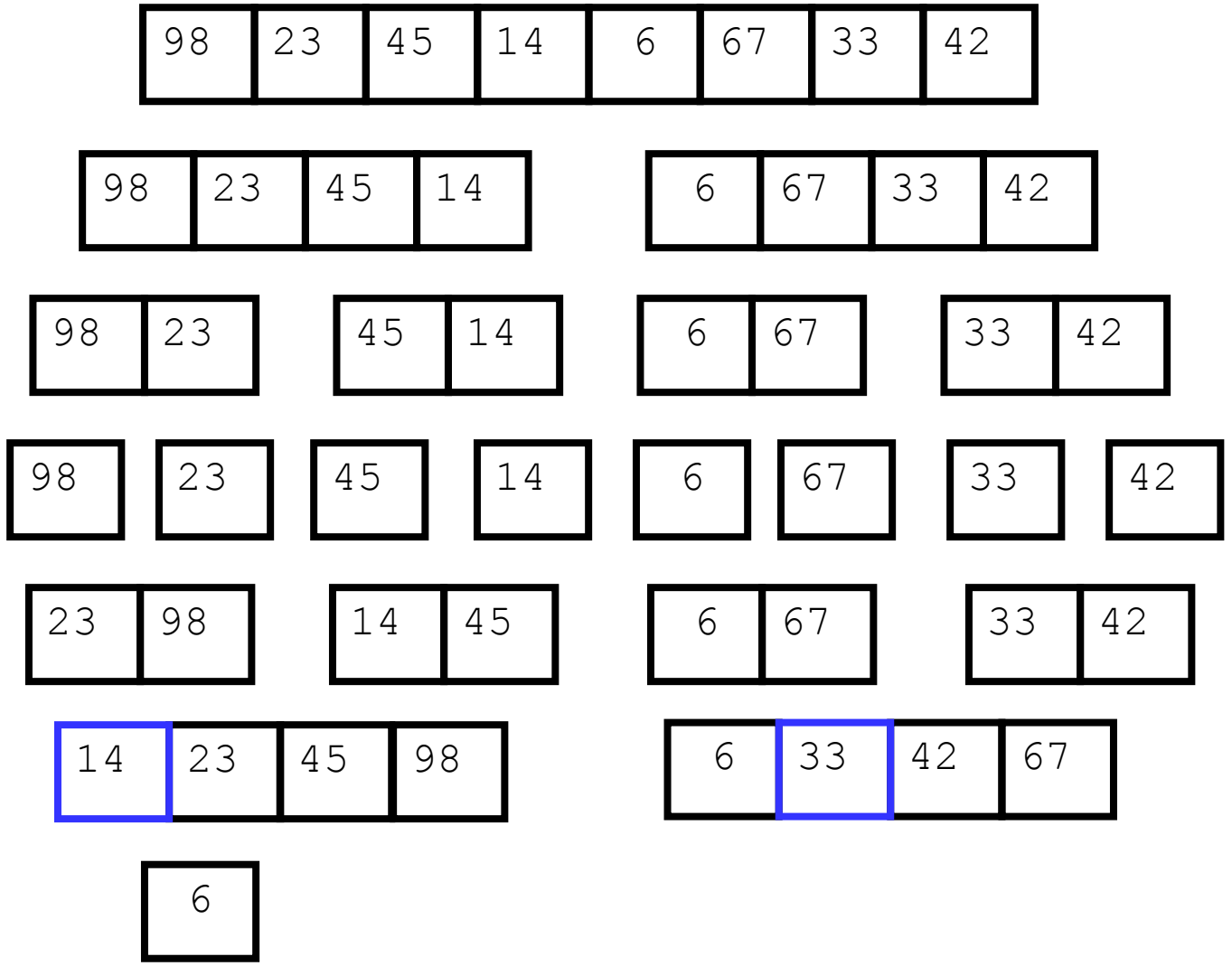

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33
Merge

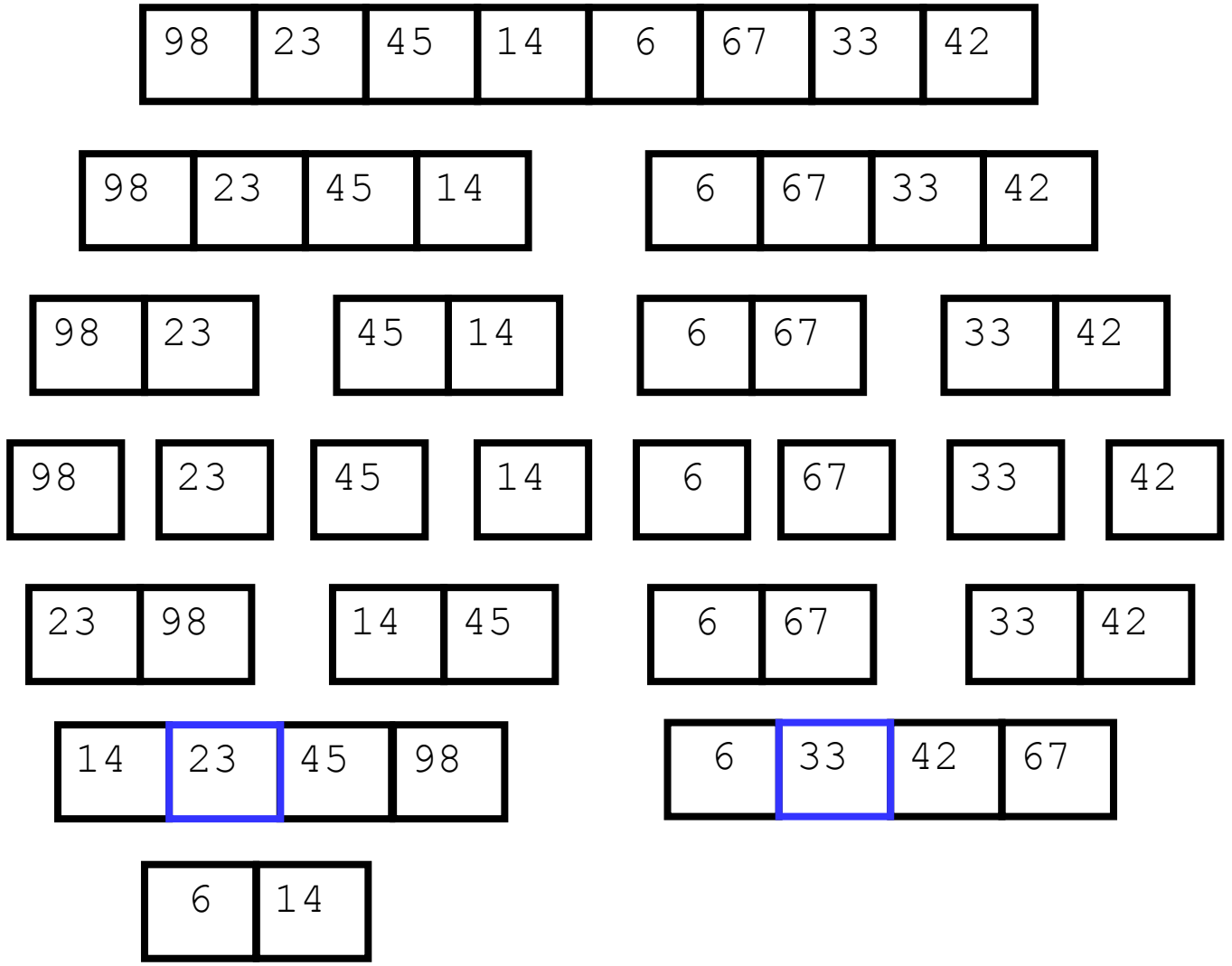

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42
Merge

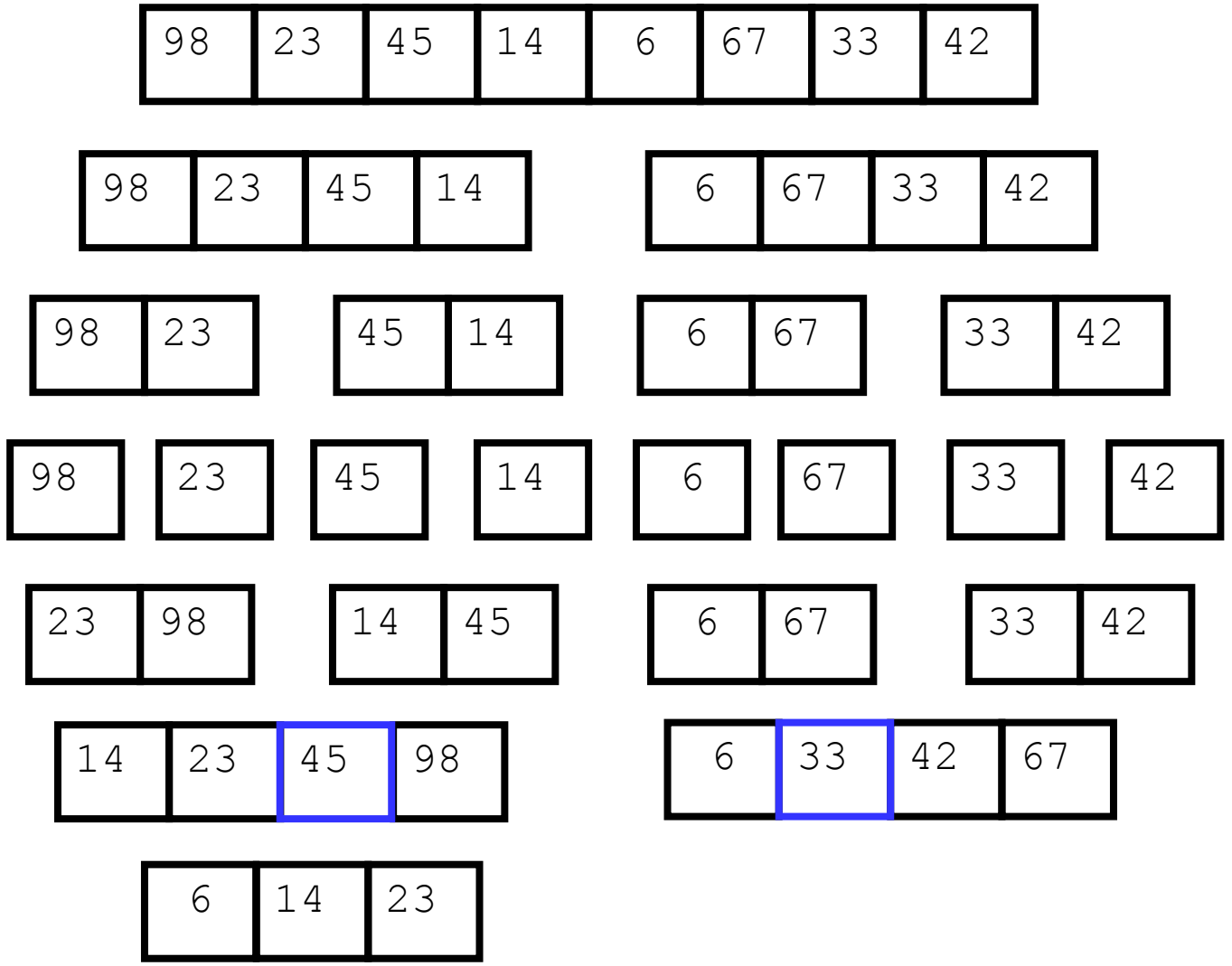

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

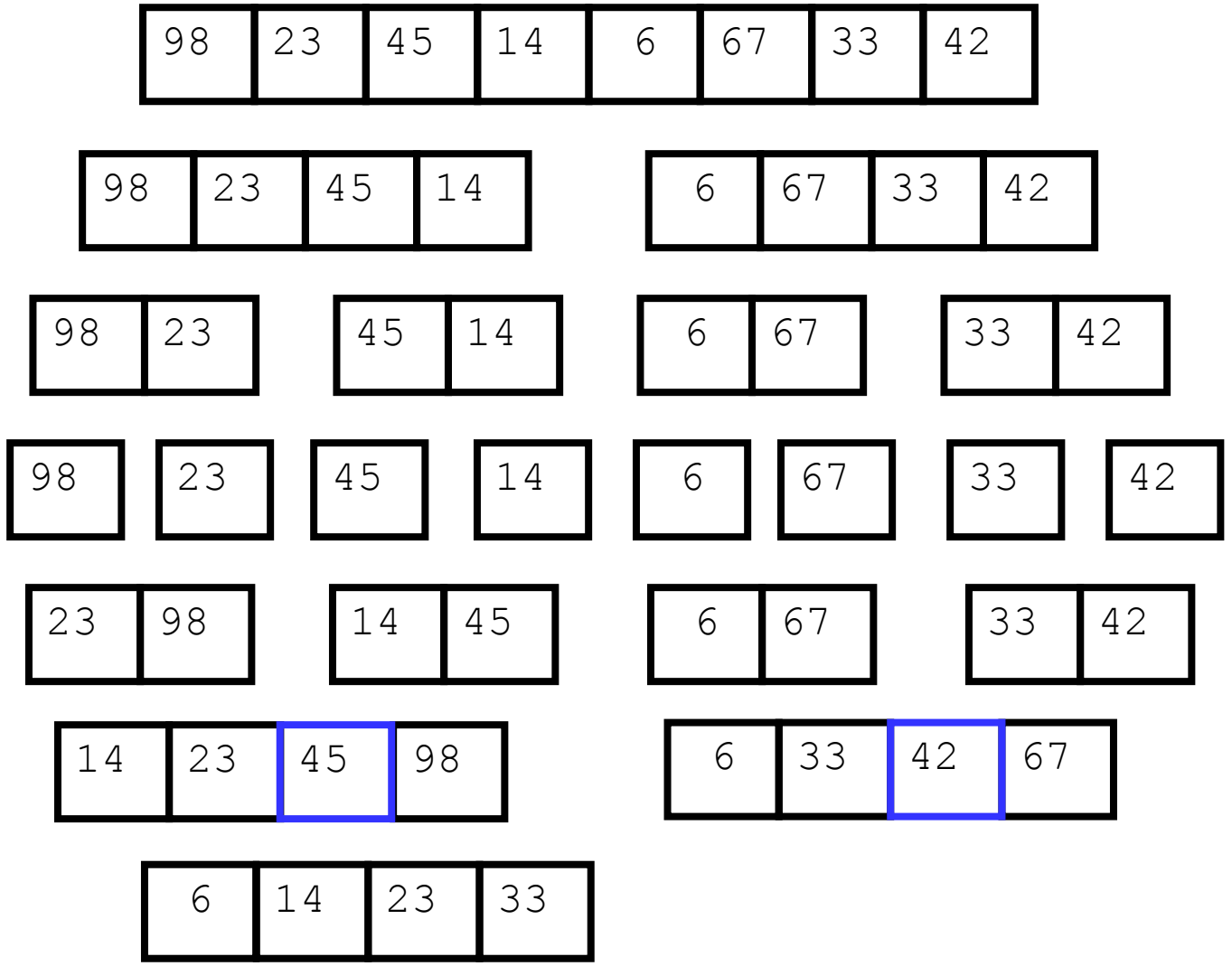

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67

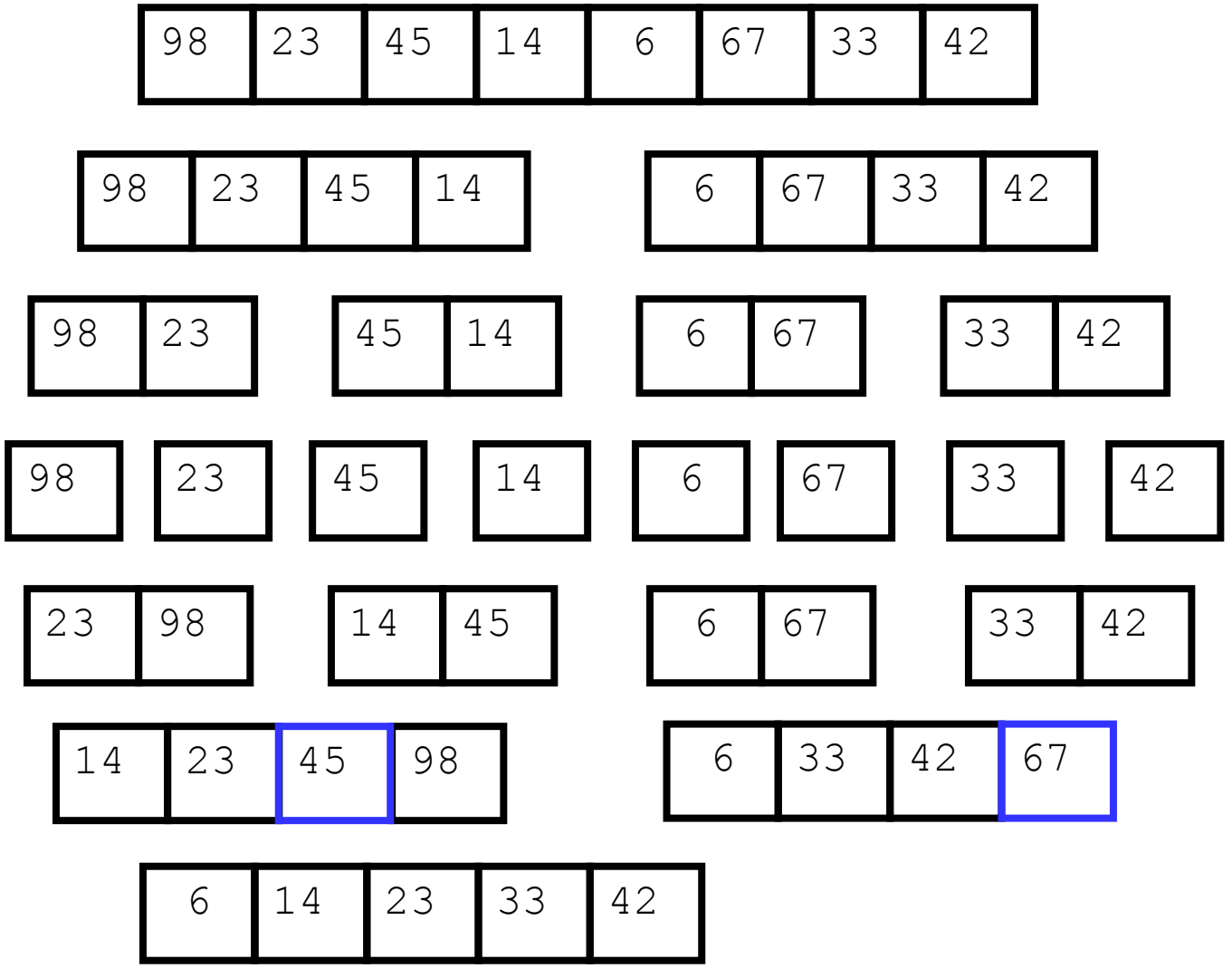
Merge

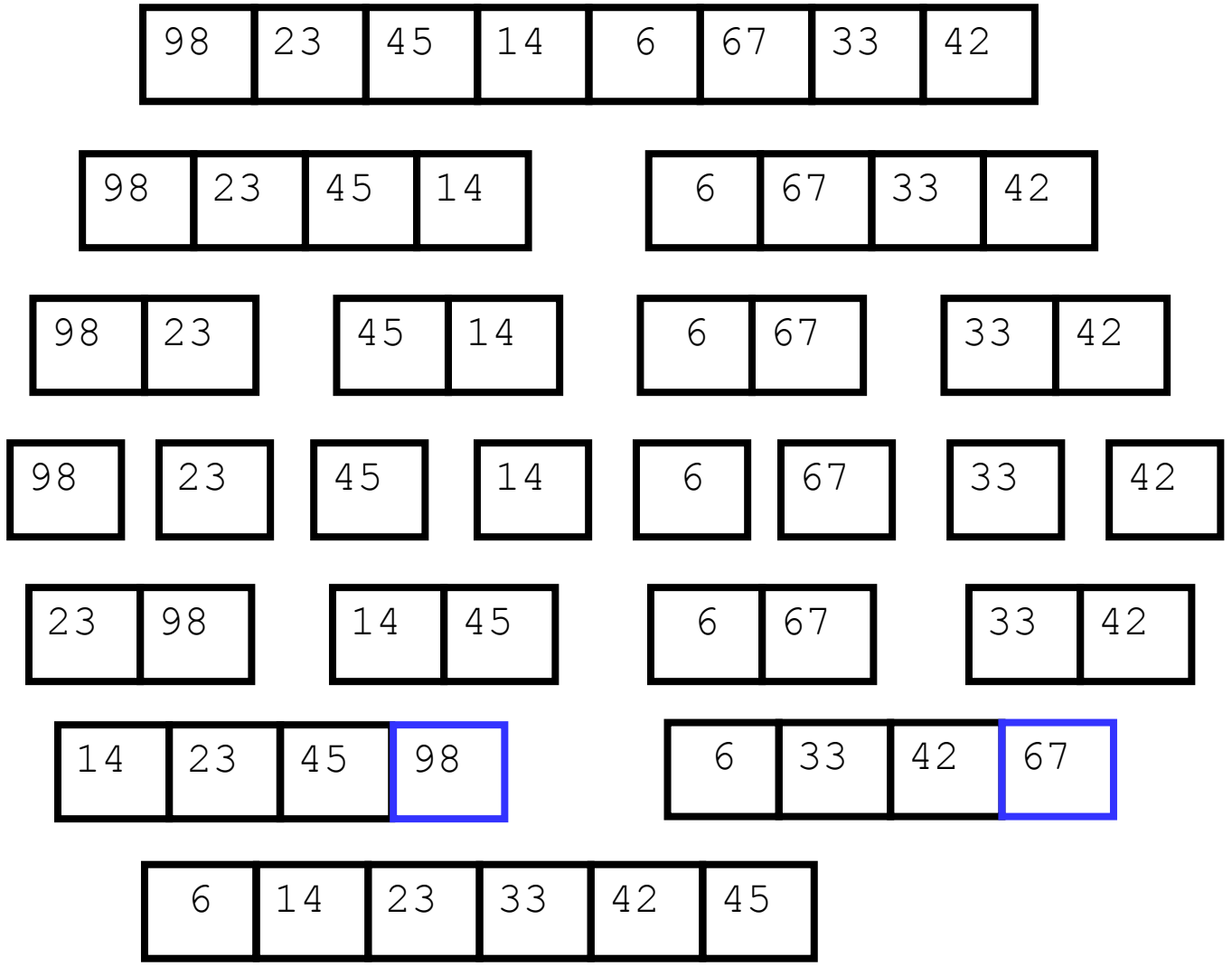

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6
Merge

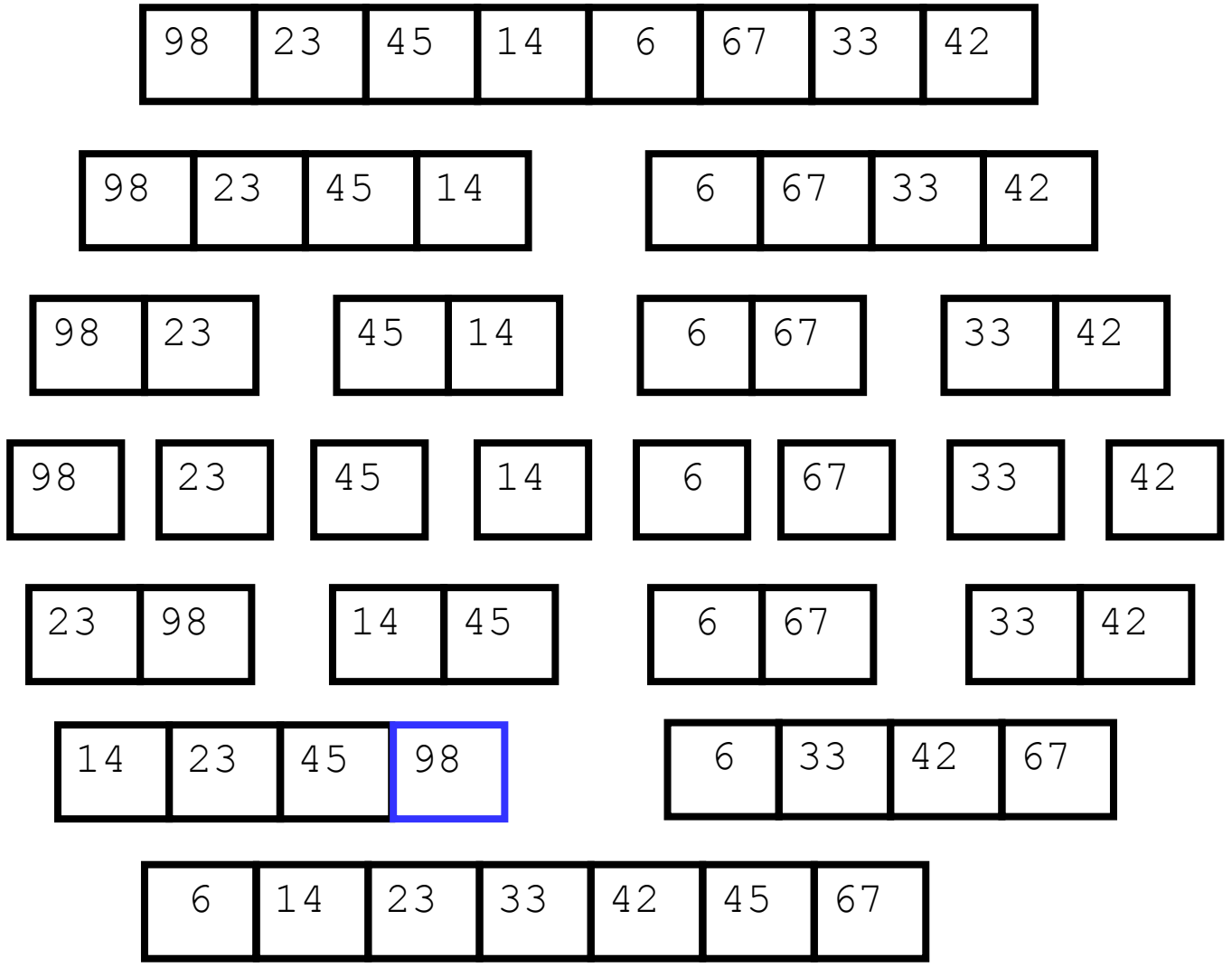

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14
Merge

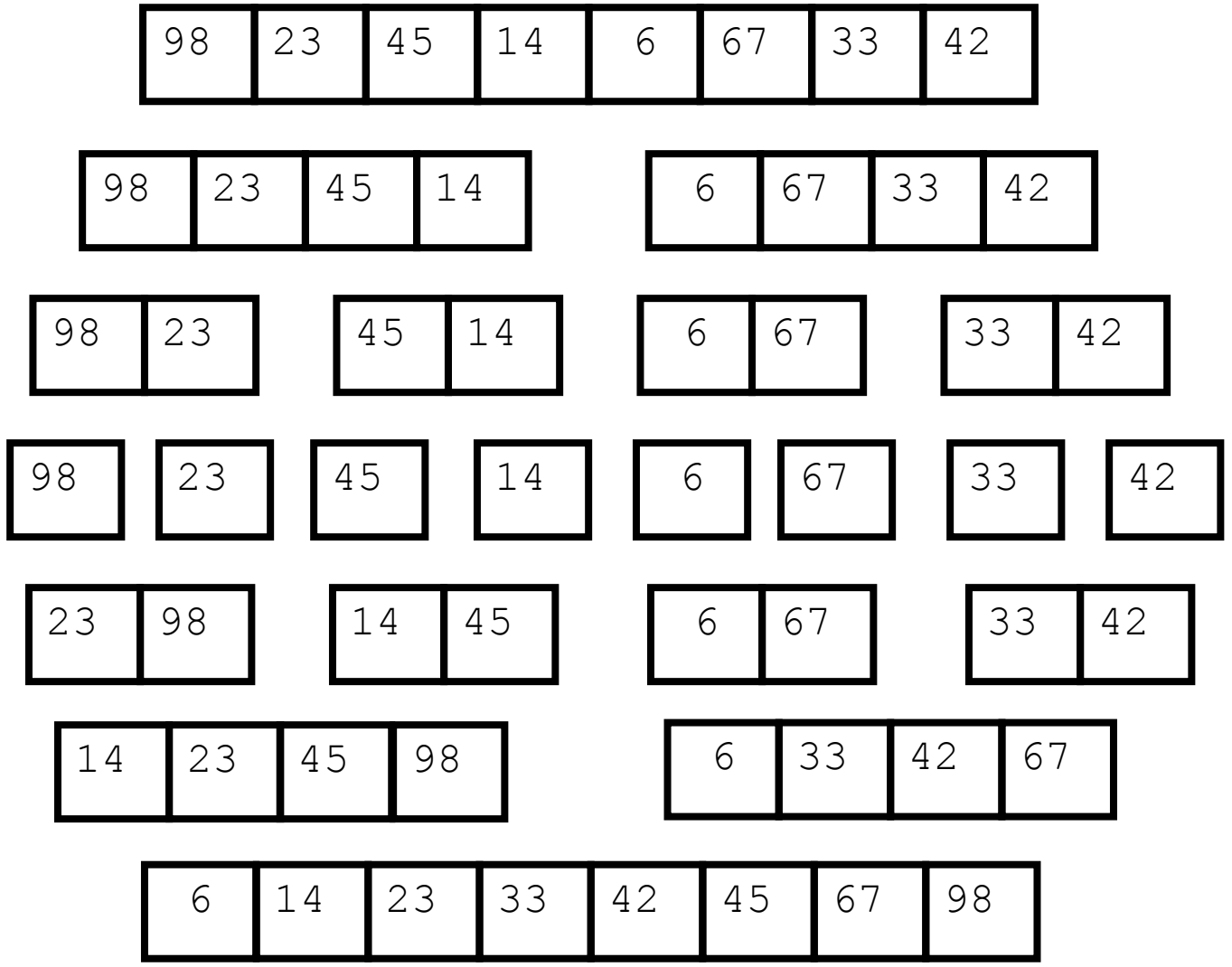

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23
Merge

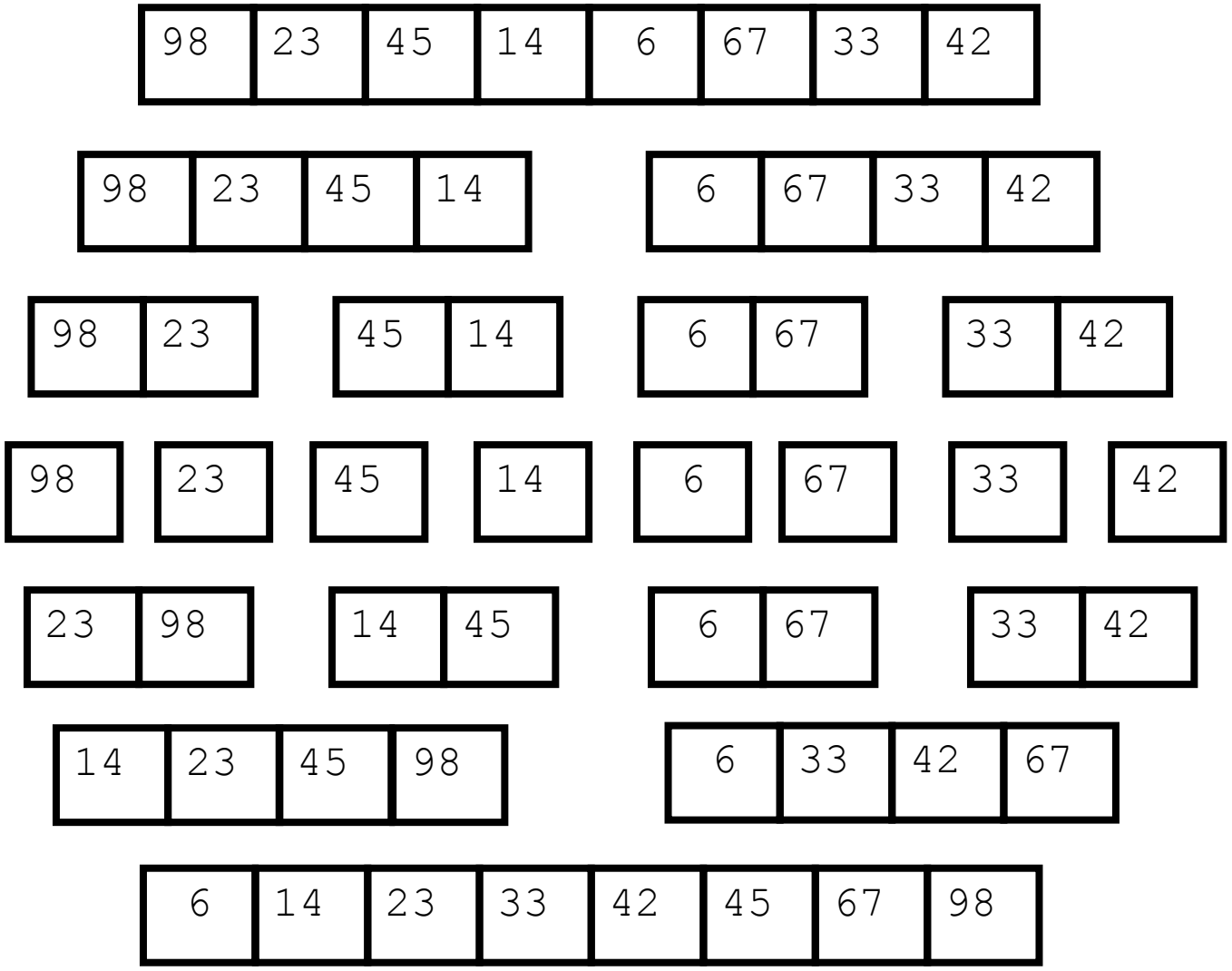

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33
Merge

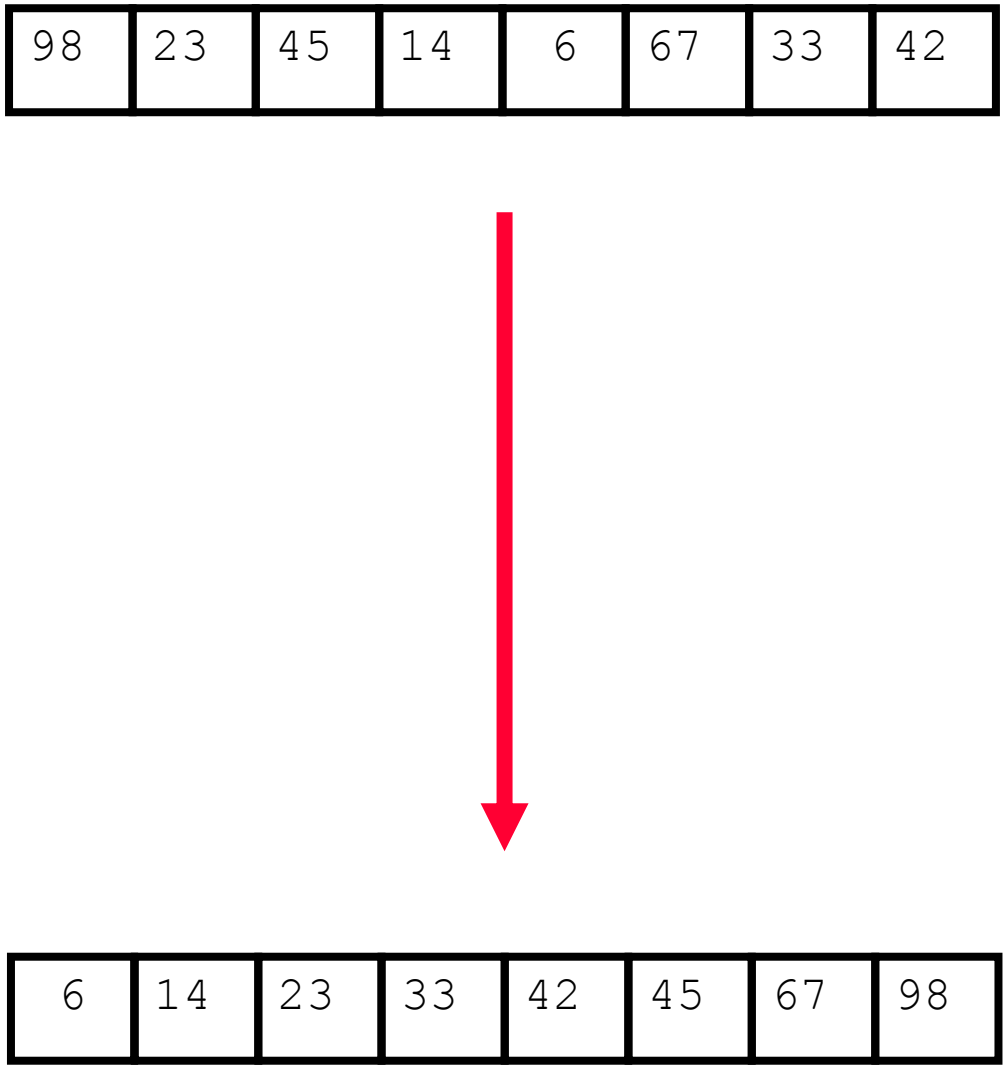

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42
Merge


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

Algorithm

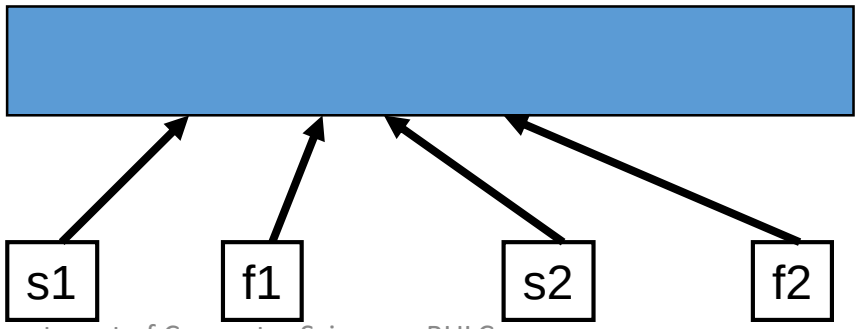
```
Mergesort(Passed an array)
  if array size > 1
    Divide array in half
    Call Mergesort on first half.
    Call Mergesort on second half.
    Merge two halves.

Merge(Passed two arrays)
  Compare leading element in each array
  Select lower and place in new array.
     (If one input array is empty then place
      remainder of other array in output array)
```

Merge Sort

- We don’t really pass in two arrays!

- We pass in one array with indicator variables which tell us where one set of data starts and finishes and where the other set of data starts and finishes.



Merge Sort

```
void merge(int*,int*,int,int,int);
void mergesort(int *a, int*b, int low, int high) {
  int pivot;
  if(low<high) {
     pivot=(low+high)/2;
     mergesort(a,b,low,pivot);
     mergesort(a,b,pivot+1,high);
     merge(a,b,low,pivot,high);
   }
}
  int main() {
  int a[] = {12,10,43,23,78,45,56,98,41,90,24};
  int num; num = sizeof(a)/sizeof(int);
  int b[num];
  mergesort(a,b,0,num-1);
  for(int i=0; i<num; i++) cout<<a[i]<<" "; cout<<endl; }
```

```
void merge(int *a, int *b, int low, int pivot,
  int high) {
   int h,i,j,k; h=low; i=low; j=pivot+1;
   while((h<=pivot)&&(j<=high)) {
     if(a[h]<=a[j]) { b[i]=a[h]; h++; }
     else { b[i]=a[j]; j++; }
     i++; }
     if(h>pivot) {
       for(k=j; k<=high; k++) {
         b[i]=a[k]; i++; } }
     else {
       for(k=h; k<=pivot; k++) {
         b[i]=a[k]; i++; } }
     for(k=low; k<=high; k++) a[k]=b[k]; }
```

L=0; h=10; p= 5 L
L=0; h=5; p=2 L
~~L=0; h=2 p=1 L~~
~~L=0; h=1 p=0 L~~
~~L=0 h=0~~
~~L=0; h=1 p=0 R~~
~~L=1 h=1~~

Merge (a, b, 0, 0, 1);
a[] = (h & j are used as indexes for left and right sub array to be sorted)

12	**10**	43	23	78	45	56	98	41	90	24

b[] = (i is the index with b array of sorted merged items)

~~L=0; h=2 p=1 R~~
~~Merge (a, b, 0, 1, 2);~~
L=0; h=5; p=2 R

Quicksort / partition-exchange sort

Quick sort is a divide and conquer algorithm.

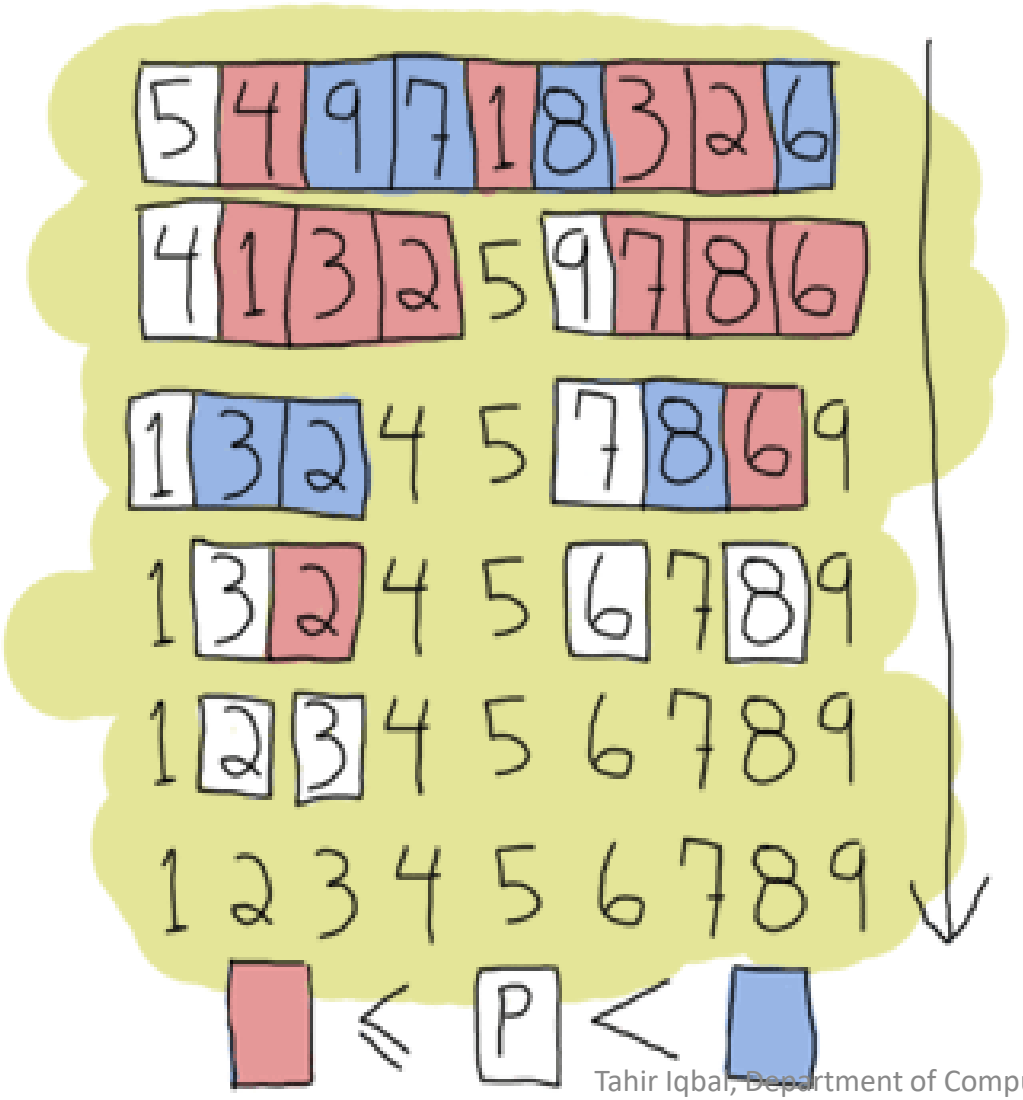
Quick sort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

The steps are:

1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

The base case of the recursion are lists of size zero or one, which never need to be sorted.

Quick Sort


5 4 9 7 1 8 3 2 6
4 1 3 2 5 9 7 8 6
1 3 2 4 5 7 8 6 9
1 3 2 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
< P <


Quicksort
24 5 3 35 14 23 19 43 2
2 5 3 14 19 23 35 43 24
2 3 5 14 19 23 35 43 24
2 3 5 14 19 23 24 35 43

Quick Sort

```
void quickSort(int a[], int first, int last);
int pivot(int a[], int first, int last);
void swap(int& a, int& b);
Void main()
{
  int test[] = { 7, 13, 1, 3, 10, 5, 2, 4 };
  int N = sizeof(test)/sizeof(int);
  quickSort(test, 0, N-1);
}
void swap(int& a, int& b) {
  int temp = a;
  a = b;
  b = temp;
}
```

QuickSort: first=0; last = 7;

Pivot: first=0; last = 7; pivotElement=7; p =0

7	13	1	3	10	5	2	4

7	1	13	3	10	5	2	4

7	1	3	13	10	5	2	4

7	1	3	5	10	13	2	4

7	1	3	5	2	4	10	13

4	1	3	5	2	**7**	10	13

```
void quickSort( int a[], int first, int last) {
  int pivotElement;
  if(first < last)  {
    pivotElement = pivot(a, first, last);
    quickSort(a, first, pivotElement-1);
    quickSort(a, pivotElement+1, last); }
}
int pivot(int a[], int first, int last){
  int p = first;
  int pivotElement = a[first];
  for(int i = first+1 ; i <= last ; i++){
    if(a[i] <= pivotElement){
      p++;
      swap(a[i], a[p]);
    }
  }
  swap(a[p], a[first]);
  return p;
}
```

Useful Links

http://www.csanimated.com/animation.php?t=Quicksort

http://www.hakansozer.com/category/c/

http://www.nczonline.net/blog/2012/11/27/computer-science-in-javascript-quicksort/

http://www.sorting-algorithms.com/shell-sort