DATA STRUCTURE AND ALGORITHMS

Definition

- Data structure is representation of the logical relationship existing between individual elements of data.
- In other words, a data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

Introduction

- Data structure affects the design of both structural & functional aspects of a program.

  Program=algorithm + Data Structure

- You know that a algorithm is a step by step procedure to solve a particular function.

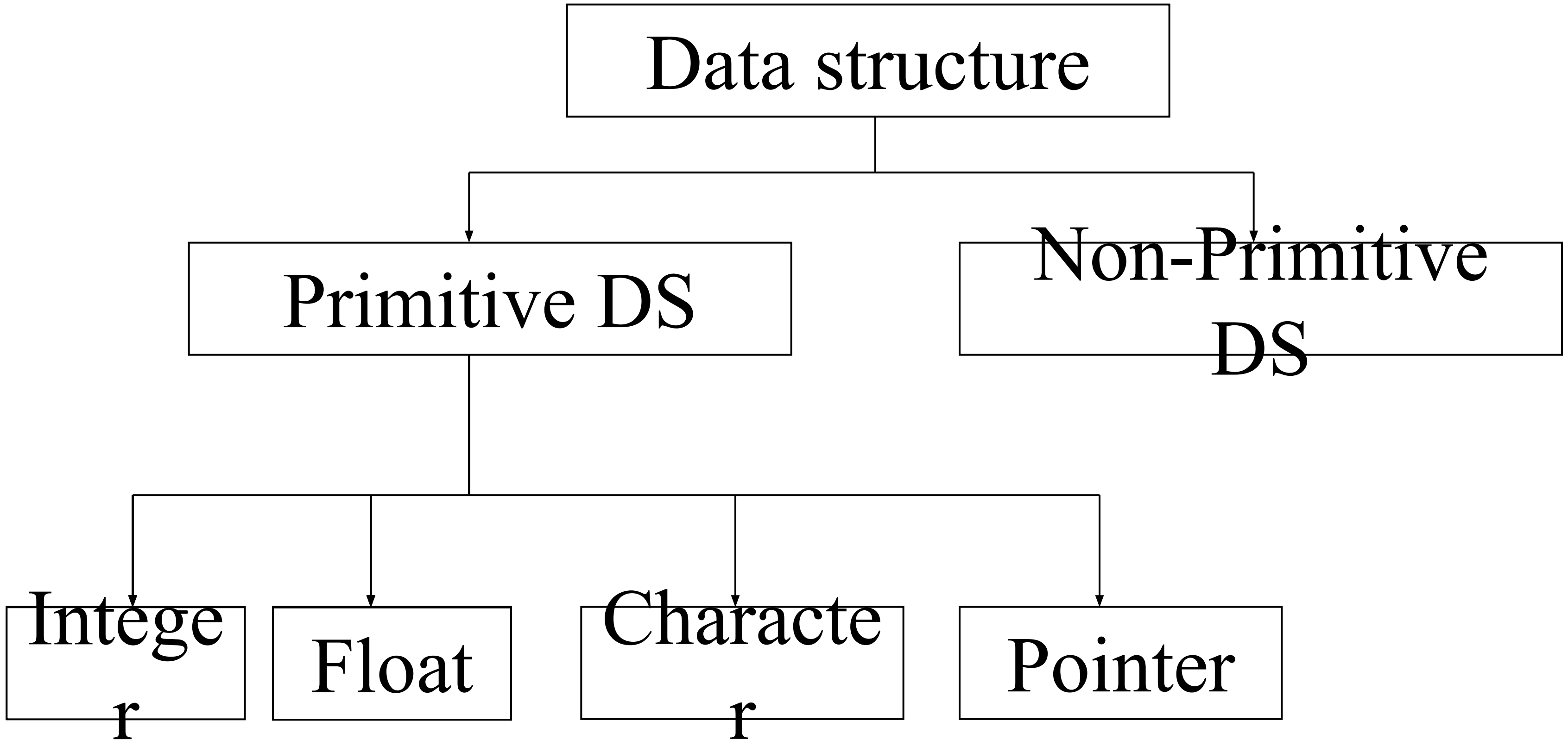
Introduction

- That means, algorithm is a set of instruction written to carry out certain tasks & the data structure is the way of organizing the data with their logical relationship retained.
- To develop a program of an algorithm, we should select an appropriate data structure for that algorithm.

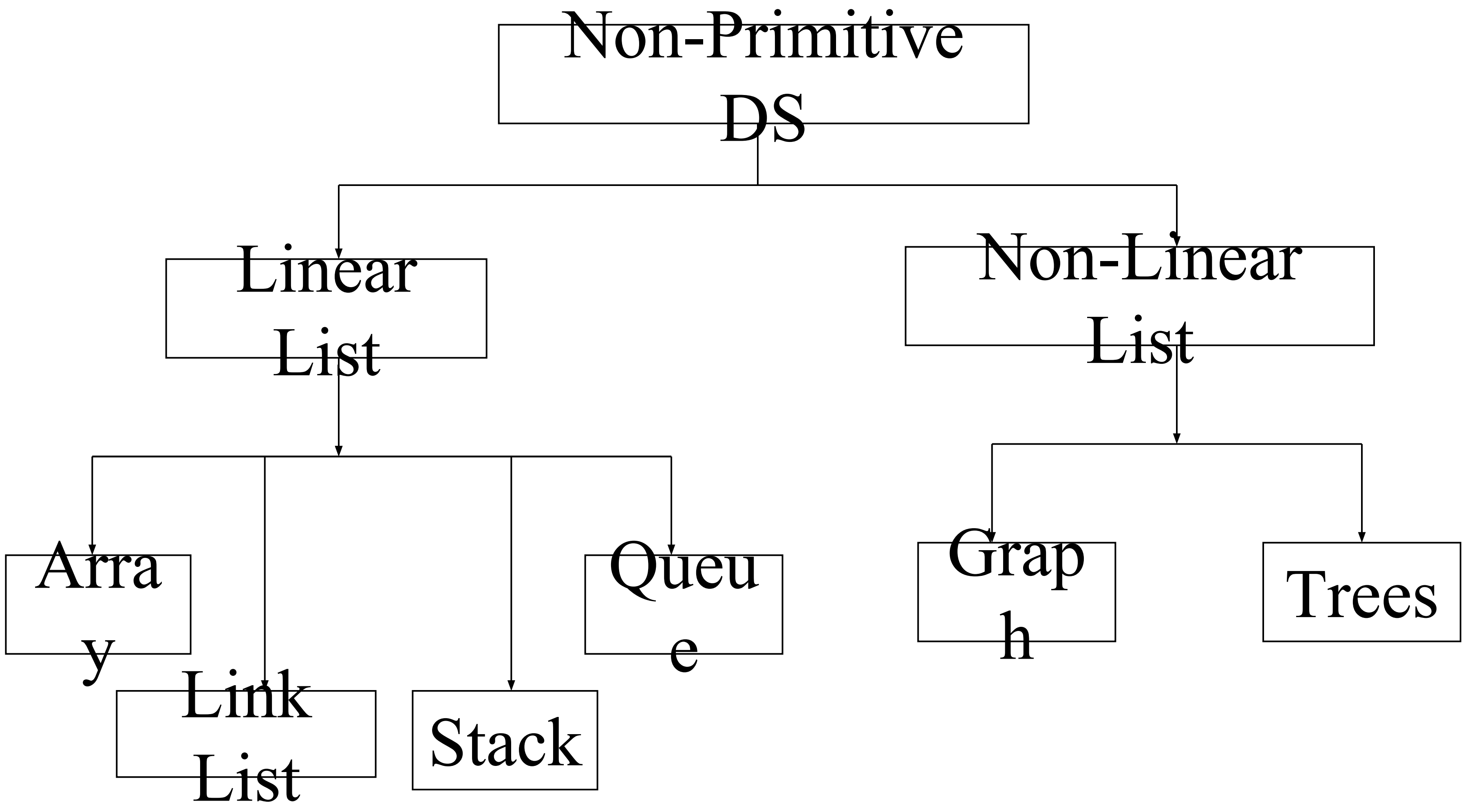
Classification of Data Structure

- Data structure are normally divided into two broad categories:
  - Primitive Data Structure
  - Non-Primitive Data Structure

Classification of Data Structure



Classification of Data Structure



Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.

Non-Primitive Data Structure

- Lists, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must take operations to be performed on the data structure.

Non-Primitive Data Structure

- The most commonly used operation on data structure are broadly categorized into following types:
  - Create
  - Selection
  - Updating
  - Searching
  - Sorting
  - Merging
  - Destroy or Delete

Different between them

- A primitive data structure is generally a basic structure that is usually built into the language, such as an integer, a float.
- A non-primitive data structure is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

Description of various Data Structures : Arrays

- An array is defined as a set of finite number of homogeneous elements or same data items.
- It means an array can contain one type of data only, either all integer, all float-point number or all character.

Arrays

- Simply, declaration of array is as follows:

```
int arr[10]
```

- Where int specifies the data type or type of elements arrays stores.
- "arr" is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

Arrays

- Following are some of the concepts to be remembered about arrays:
  - The individual element of an array can be accessed by specifying name of the array, following by index or subscript inside square brackets.
  - The first element of the array has index zero[0]. It means the first element and last element will be specified as:arr[0] & arr[9]
    Respectively.

Arrays

- The elements of array will always be stored in the consecutive (continues) memory location.
- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

 (Upperbound-lowerbound)+1

Arrays

- For the above array it would be (9-0)+1=10,where 0 is the lower bound of array and 9 is the upper bound of array.
- Array can always be read or written through loop. If we read a one-dimensional array it require one loop for reading and other for writing the array.

Arrays

- For example: Reading an array

```
For(i=0;i<=9;i++)
     scanf(“%d”,&arr[i]);
```

- For example: Writing an array

```
For(i=0;i<=9;i++)
     printf(“%d”,arr[i]);
```

Arrays

- If we are reading or writing two-dimensional array it would require two loops. And similarly the array of a N dimension would required N loops.
- Some common operation performed on array are:
  - Creation of an array
  - Traversing an array

Arrays

- Insertion of new element
- Deletion of required element
- Modification of an element
- Merging of arrays

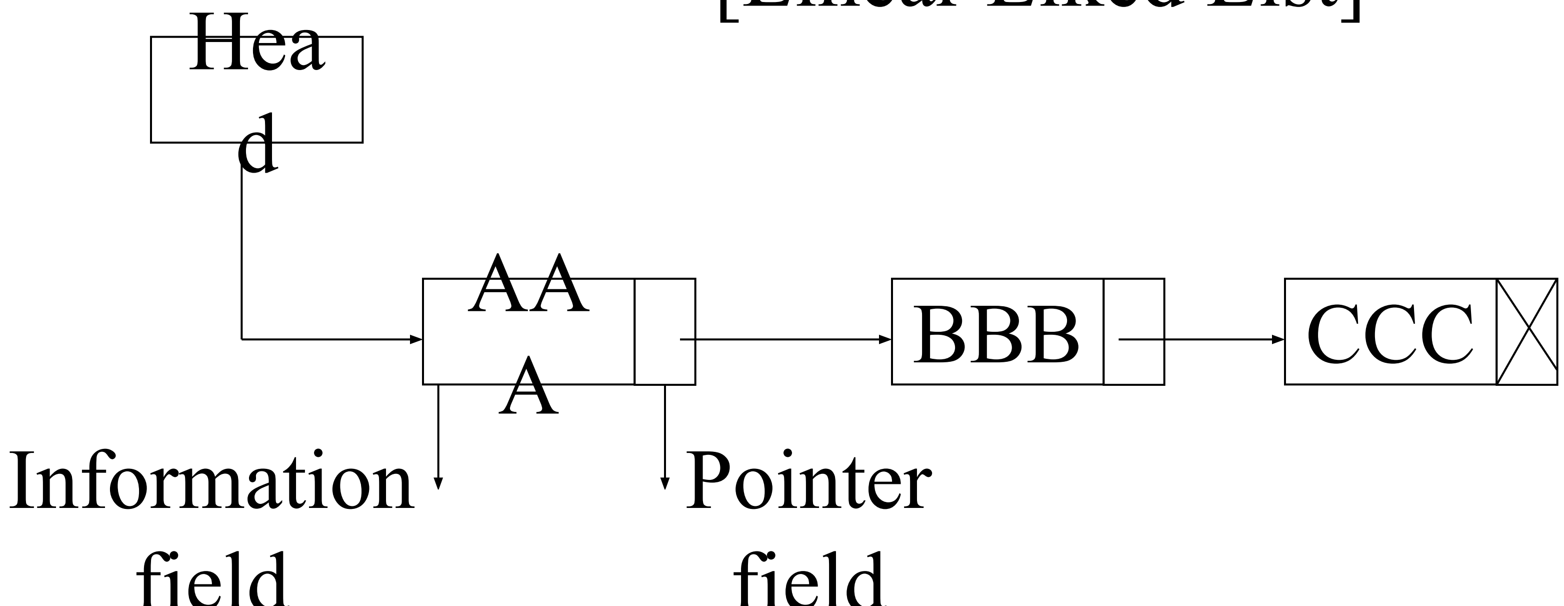
Lists

- A lists (Linear linked list) can be defined as a collection of variable number of data items.
- Lists are the most commonly used non-primitive data structures.
- An element of list must contain at least two fields, one for storing data or information and other for storing address of next element.
- As you know for storing address we have a special data structure of list the address must be pointer type.

Lists

- Technically each such element is referred to as a node, therefore a list can be defined as a collection of nodes as show bellow:

[Linear Liked List]



Lists

- Types of linked lists:
  - Single linked list
  - Doubly linked list
  - Single circular linked list
  - Doubly circular linked list

Stack

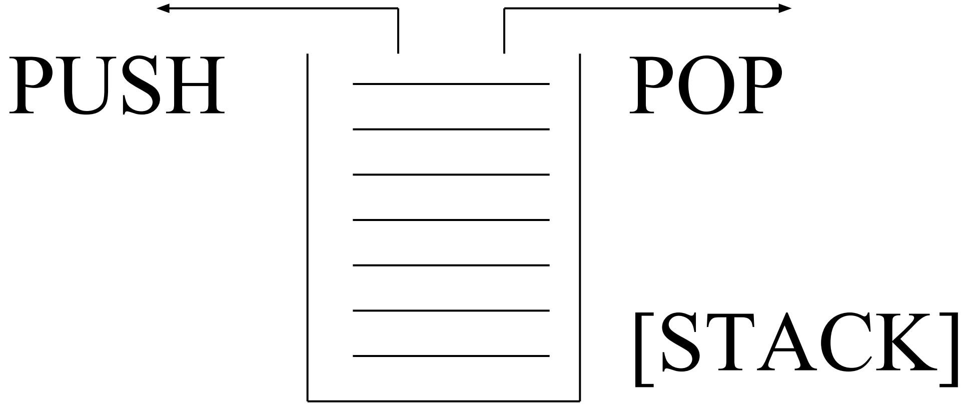
- A stack is also an ordered collection of elements like arrays, but it has a special feature that deletion and insertion of elements can be done only from one end called the top of the stack (TOP)
- Due to this property it is also called as last in first out type of data structure (LIFO).

Stack

- It could be through of just like a stack of plates placed on table in a party, a guest always takes off a fresh plate from the top and the new plates are placed on to the stack at the top.
- It is a non-primitive data structure.
- When an element is inserted into a stack or removed from the stack, its base remains fixed where the top of stack changes.

Stack

- Insertion of element into stack is called PUSH and deletion of element from stack is called POP.
- The bellow show figure how the operations take place on a stack:



Stack

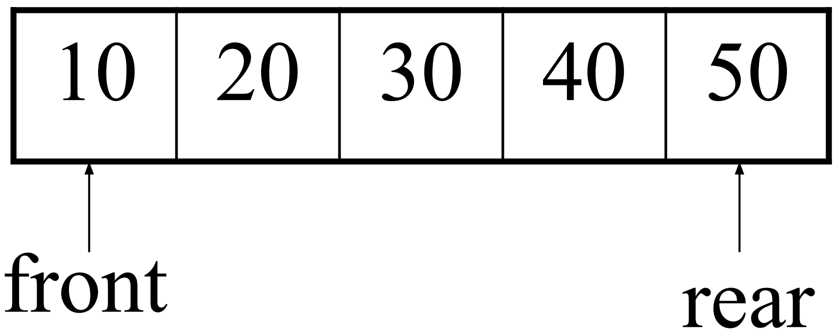
- The stack can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Queue

- Queue are first in first out type of data structure (i.e. FIFO)
- In a queue new elements are added to the queue from one end called REAR end and the element are always removed from other end called the FRONT end.
- The people standing in a railway reservation row are an example of queue.

Queue

- Each new person comes and stands at the end of the row and person getting their reservation confirmed get out of the row from the front end.
- The bellow show figure how the operations take place on a stack:



Queue

- The queue can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Trees

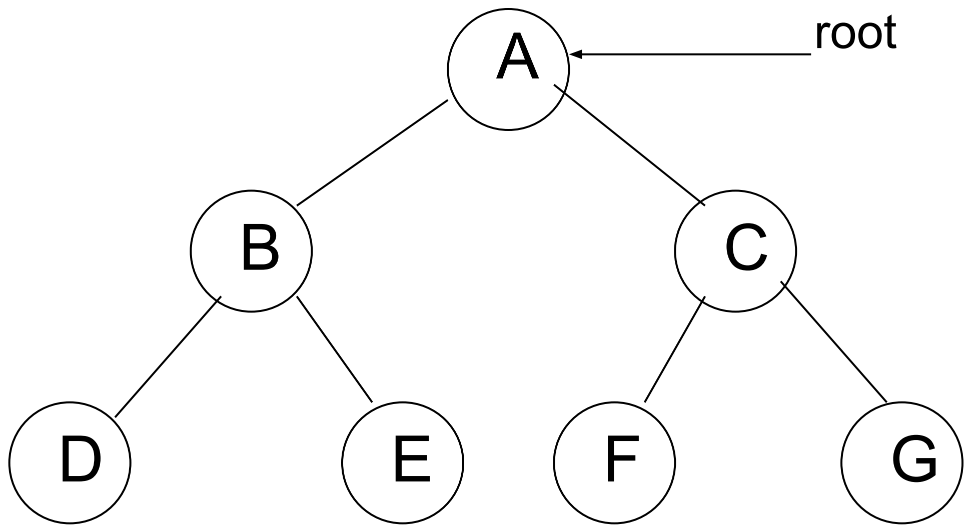
- A tree can be defined as finite set of data items (nodes).
- Tree is non-linear type of data structure in which data items are arranged or stored in a sorted sequence.
- Tree represent the hierarchical relationship between various elements.

Trees

- In trees:
- There is a special data item at the top of hierarchy called the Root of the tree.
- The remaining data items are partitioned into number of mutually exclusive subset, each of which is itself, a tree which is called the sub tree.
- The tree always grows in length towards bottom in data structures, unlike natural trees which grows upwards.

Trees

- The tree structure organizes the data into branches, which related the information.



Graph

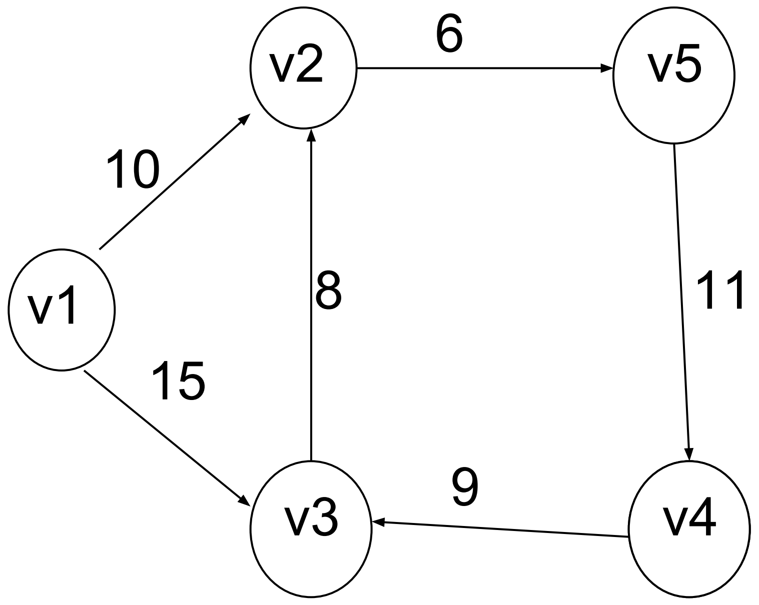
- Graph is a mathematical non-linear data structure capable of representing many kind of physical structures.
- It has found application in Geography, Chemistry and Engineering sciences.
- Definition: A graph G(V,E) is a set of vertices V and a set of edges E.

Graph

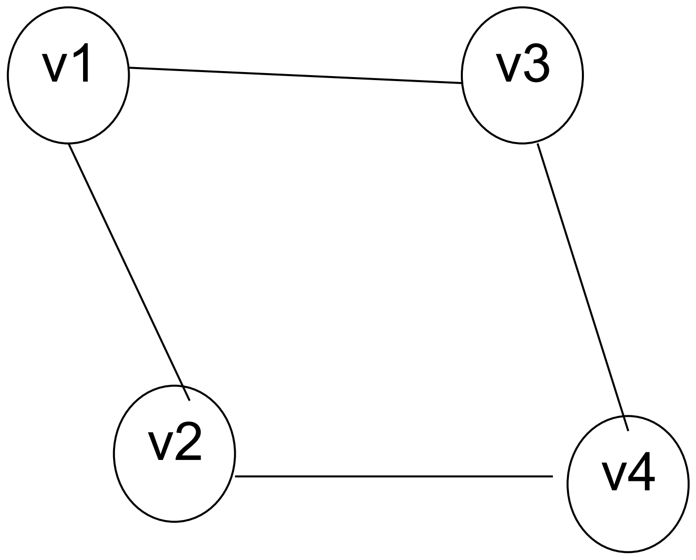
- An edge connects a pair of vertices and many have weight such as length, cost and another measuring instrument for according the graph.
- Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment.

Graph

- Example of graph:



[a] Directed & Weighted Graph



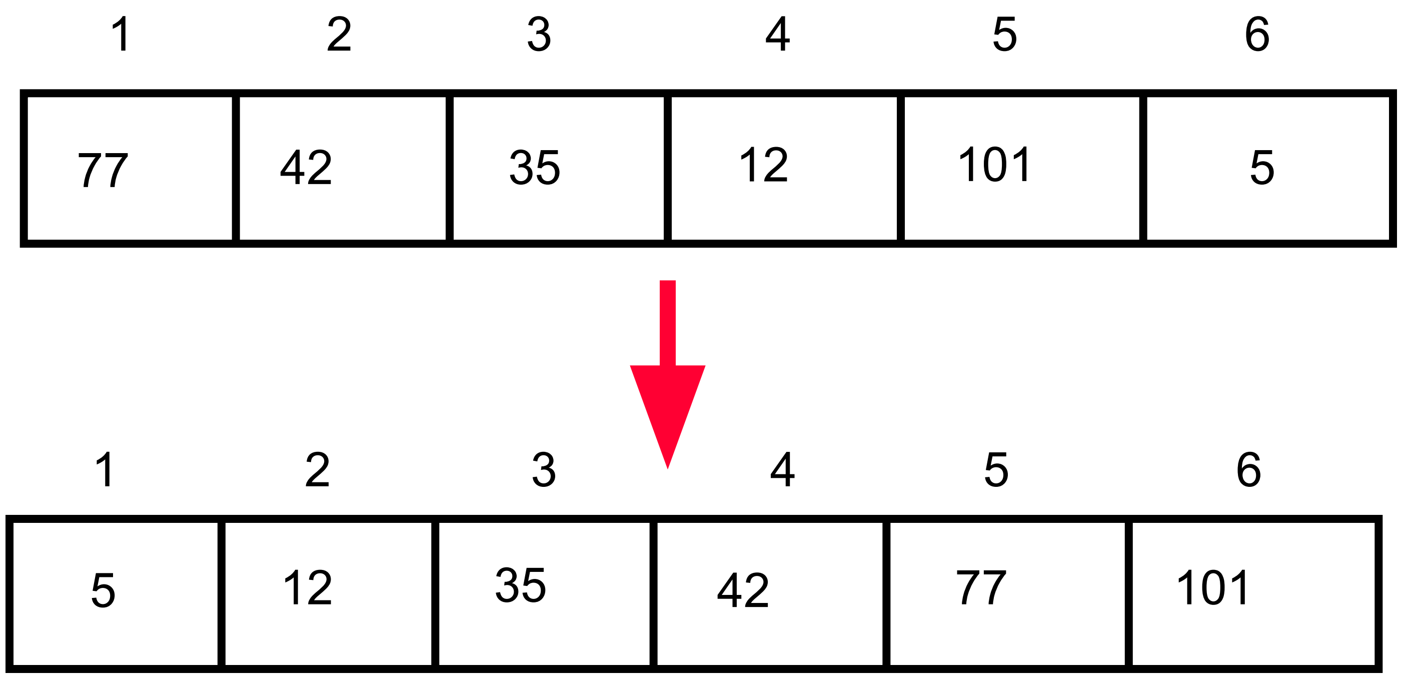
[b] Undirected Graph

Graph

- Types of Graphs:
  - Directed graph
  - Undirected graph
  - Simple graph
  - Weighted graph
  - Connected graph
  - Non-connected graph

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;
    }
}
```

Time Complexity of Selection Sort

- The total number of comparisons is :-

  $(N-1)+(N-2)+(N-3)+\ldots\ldots\ldots\ldots +1= N(N-1)/2$
- We can ignore the constant 1/2
- We can express N(N-1) as $N^2 – N$
- We can ignore N as well since N^2 grows more rapidly than N, making our algorithm $O(N^2)$

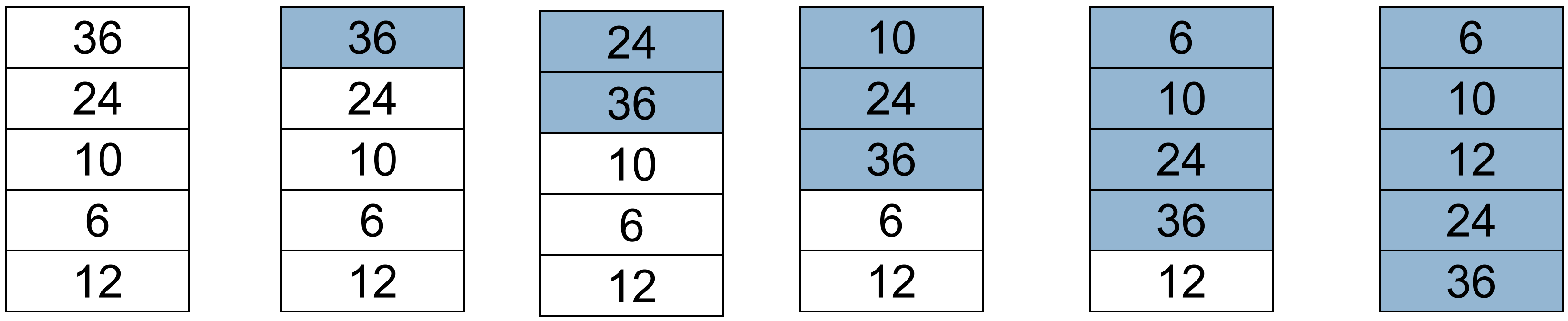
Insertion Sort

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 |

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.

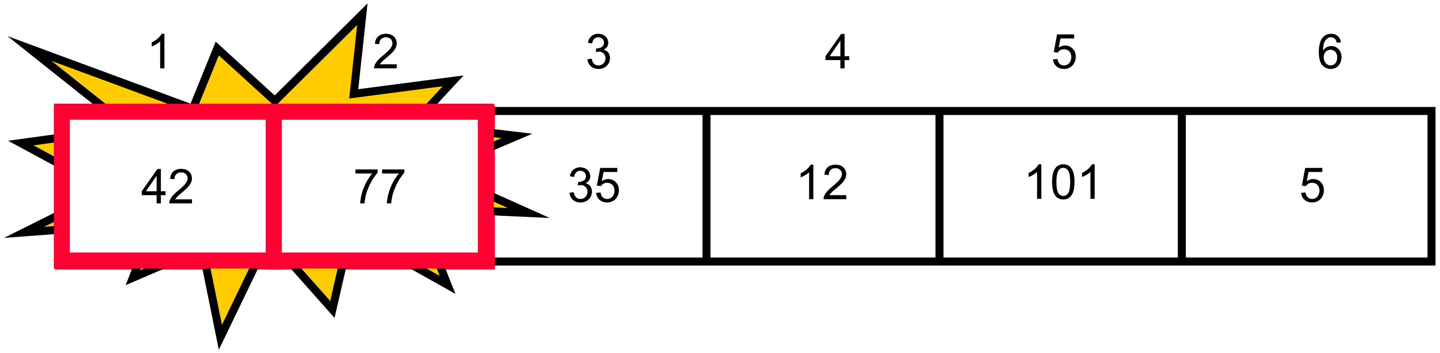
Bubble Sort

- 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

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

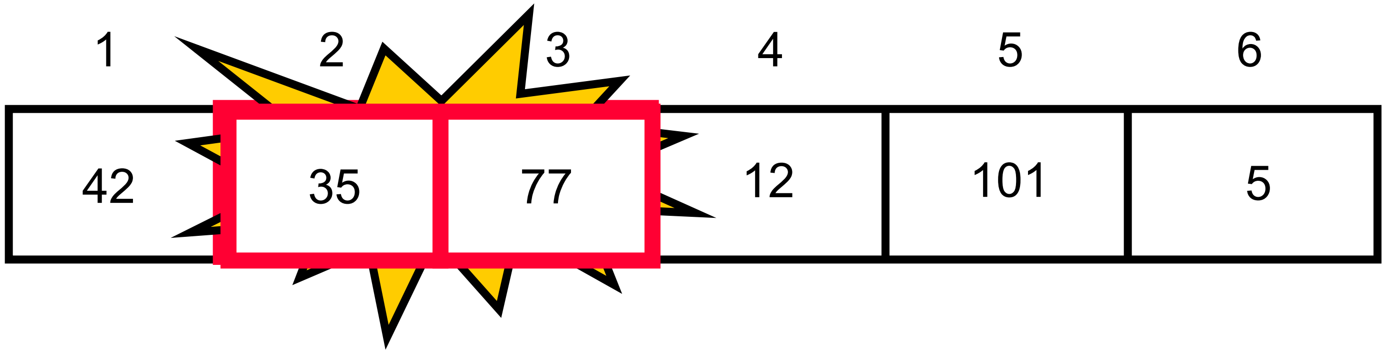
Bubble Sort

- 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



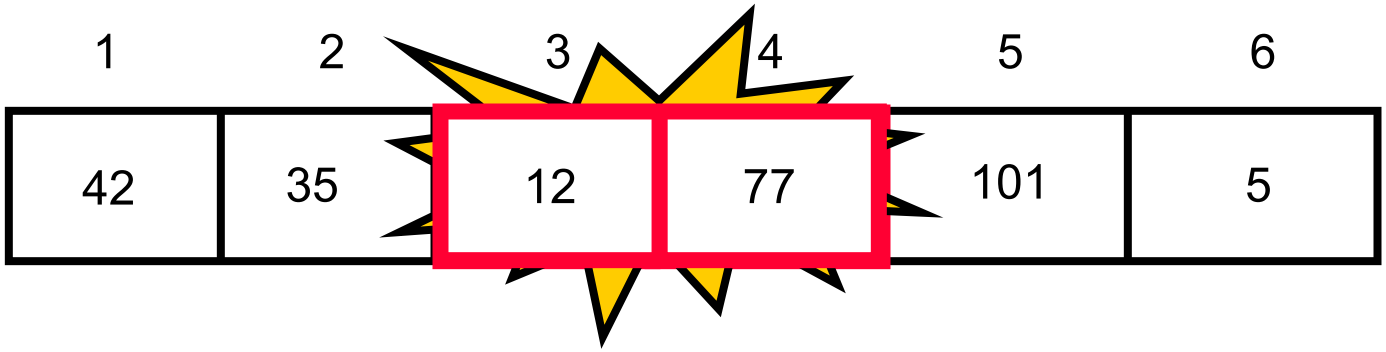
Bubble Sort

- 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



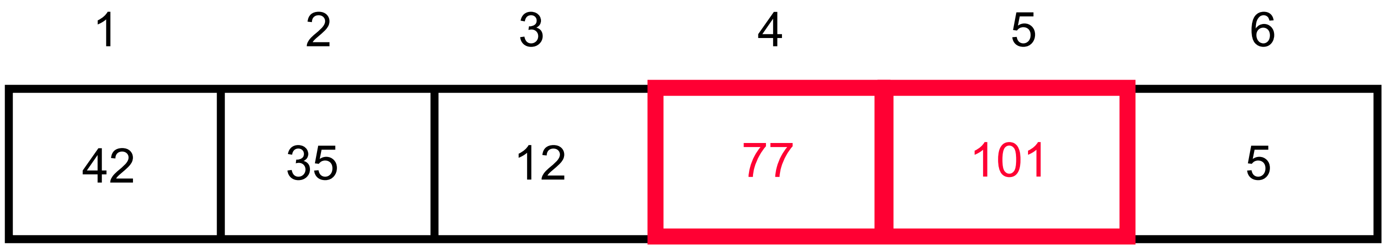
Bubble Sort

- 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

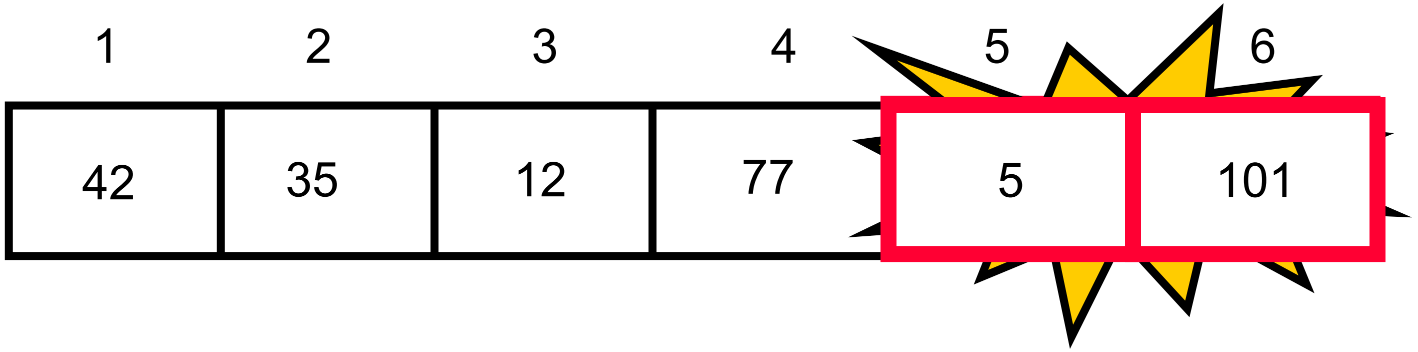
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



No need to swap

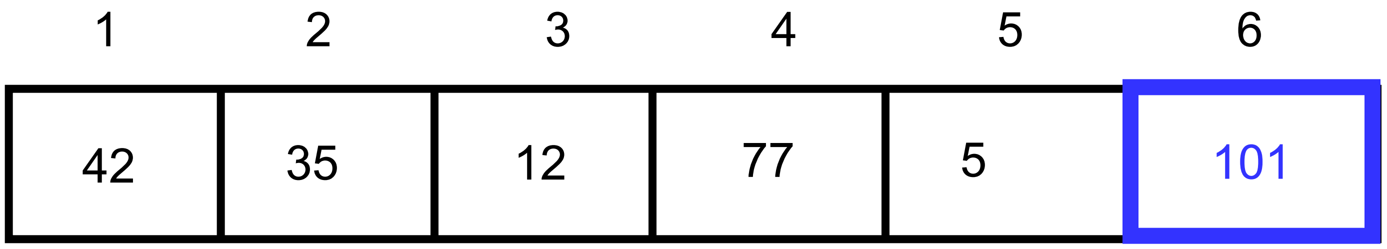
Bubble Sort

- 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

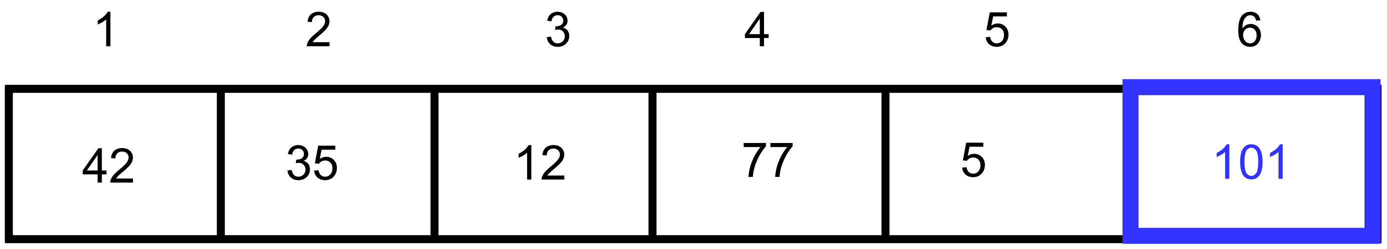
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



Largest value correctly placed

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

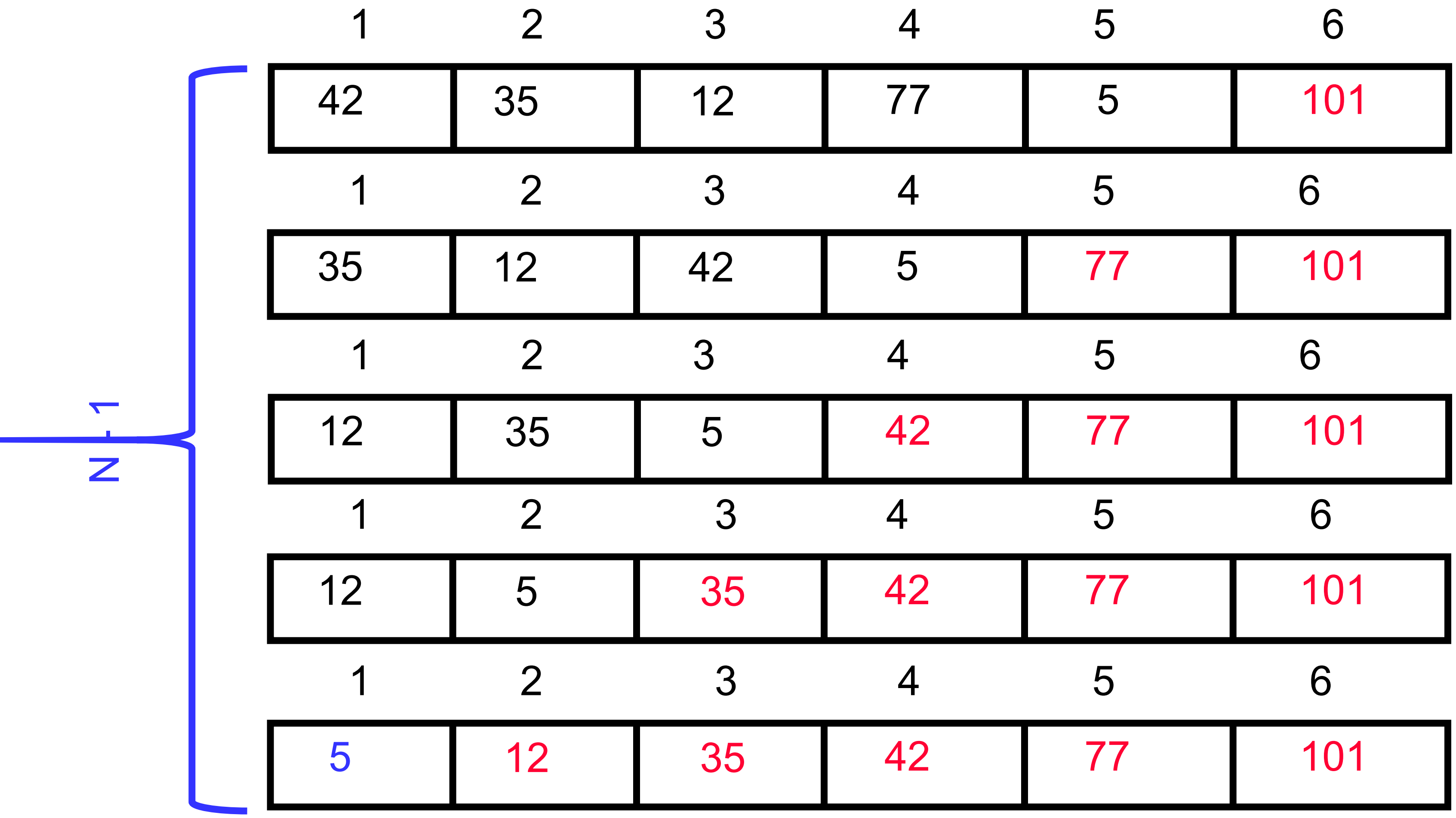


Largest value correctly placed

Repeat “Bubble Up” How Many Times?

- If we have N elements…
- And if each time we bubble an element, we place it in its correct location…
- Then we repeat the “bubble up” process N – 1 times.
- This guarantees we’ll correctly place all N elements.

“Bubbling” All the Elements



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

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

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

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

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

Reducing the Number of Comparisons

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

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

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

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

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

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"!

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

Using a Boolean “Flag”

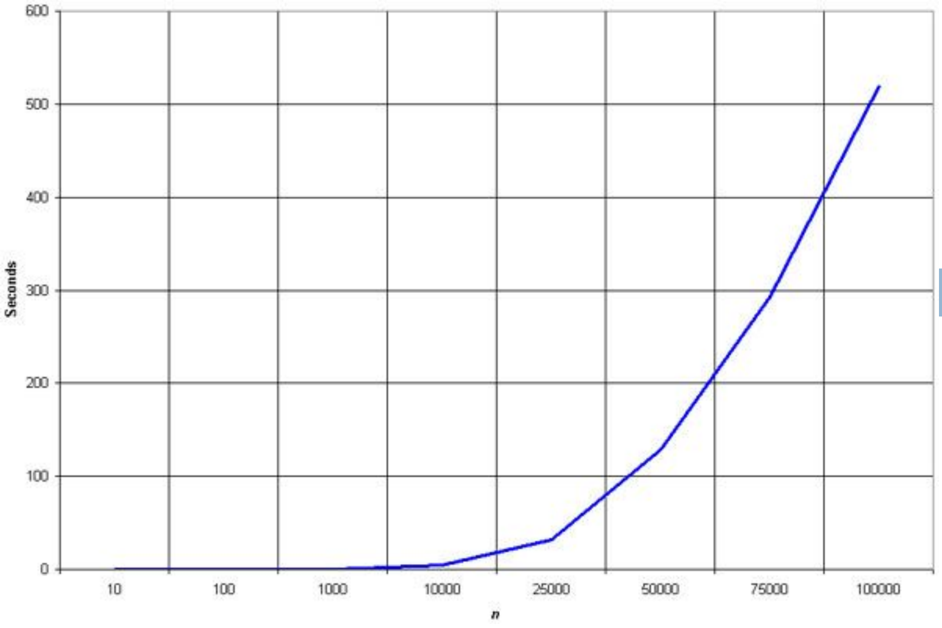
- We can use a boolean variable to determine if any swapping occurred during the “bubble up.”

- If no swapping occurred, then we know that the collection is already sorted!

- This boolean “flag” needs to be reset after each “bubble up.”

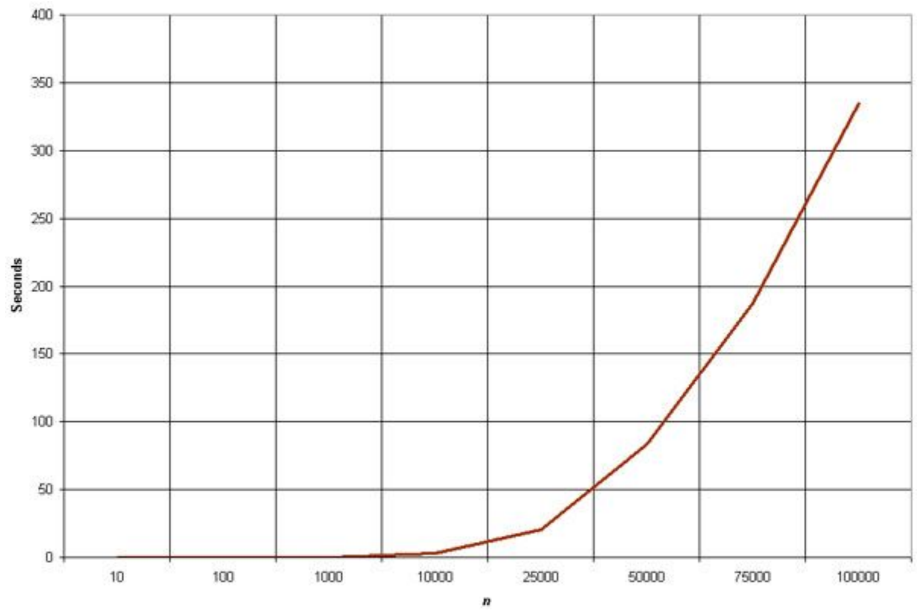
Bubble Sort Algorithm

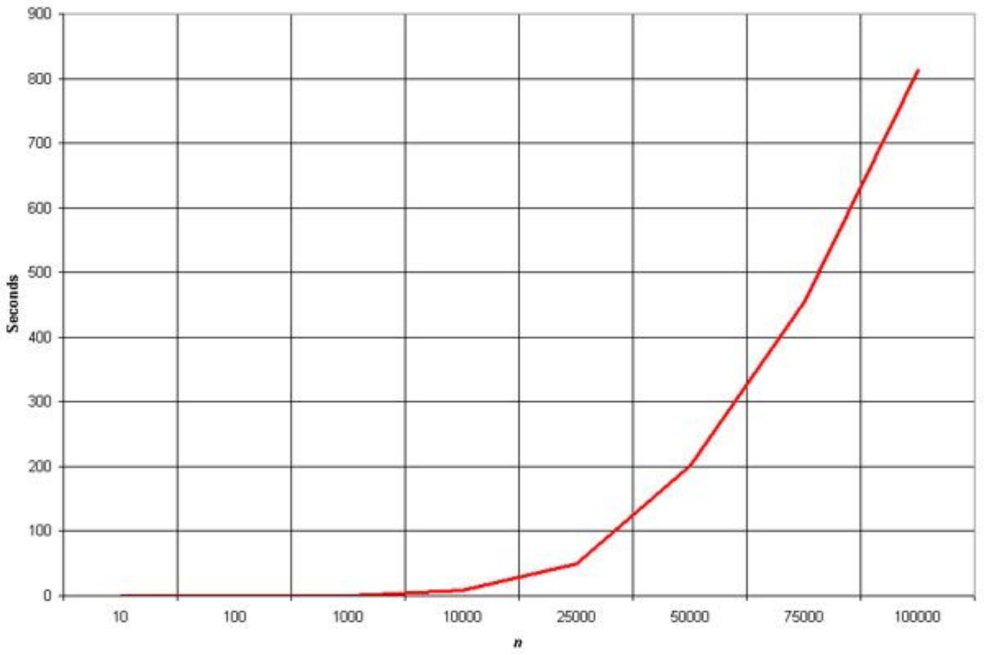
```
void bubbleSort (int S[ ],  int length) {
    bool isSorted = false;
    while(!isSorted)
    {
      isSorted = true;
      for(int i = 0; i<length; i++)
      {
          if(S[i] > S[i+1])
                {
             int temp = S[i];
             S[i] = S[i+1];
                    S[i+1] = temp;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

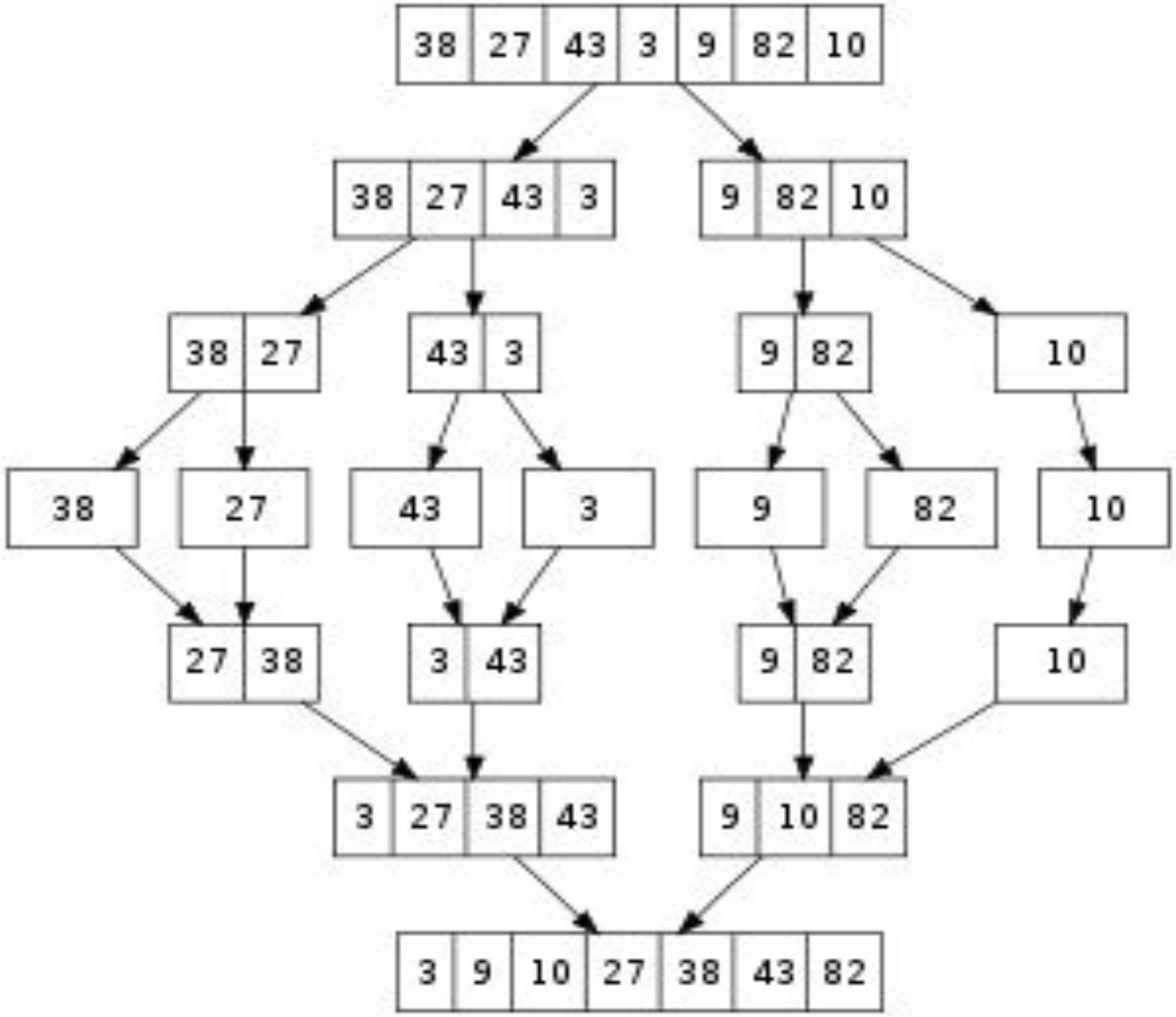
The insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items.

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



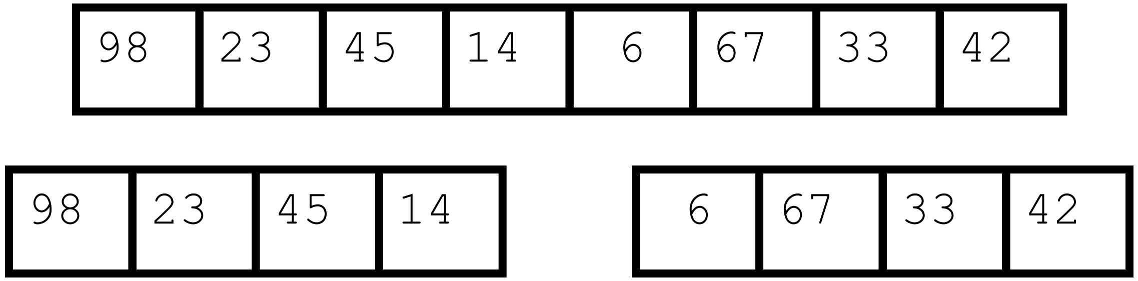
38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

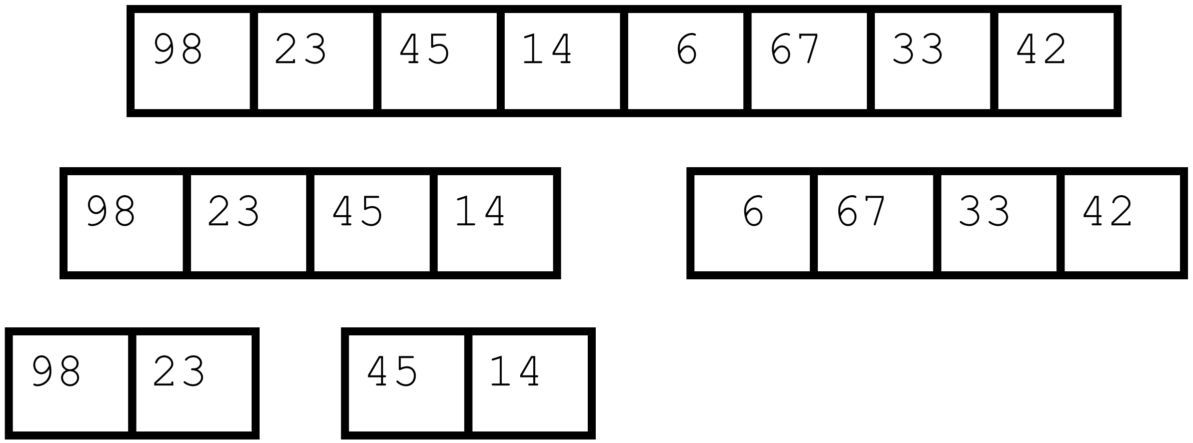
Mergesort

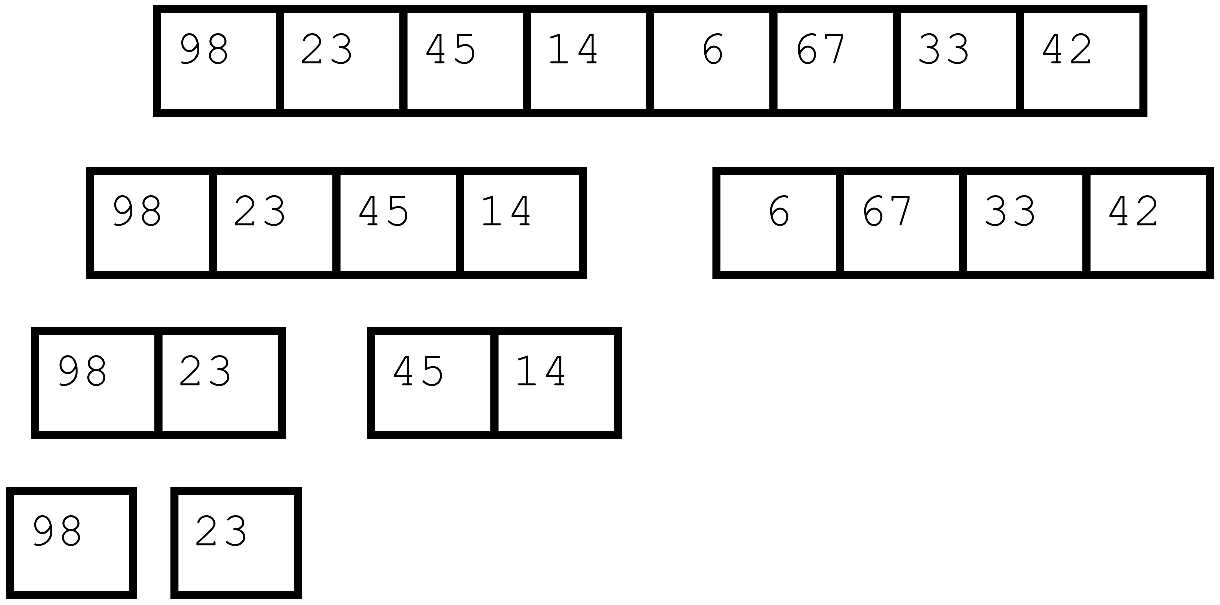
- A divide-and-conquer algorithm:
- Divide the unsorted array into 2 halves until the sub-arrays only contain one element
- Merge the sub-problem solutions together:
  - Compare the sub-array's first elements
  - Remove the smallest element and put it into the result array
  - Continue the process until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

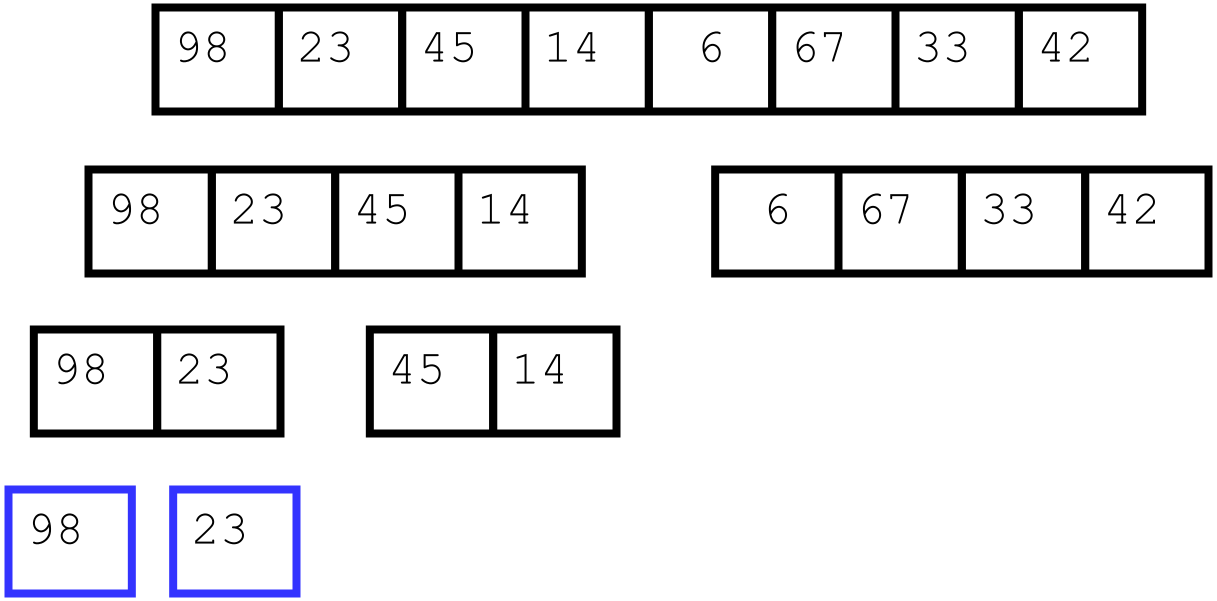
37	23	6	89	15	12	2	19

98	23	45	14	6	67	33	42

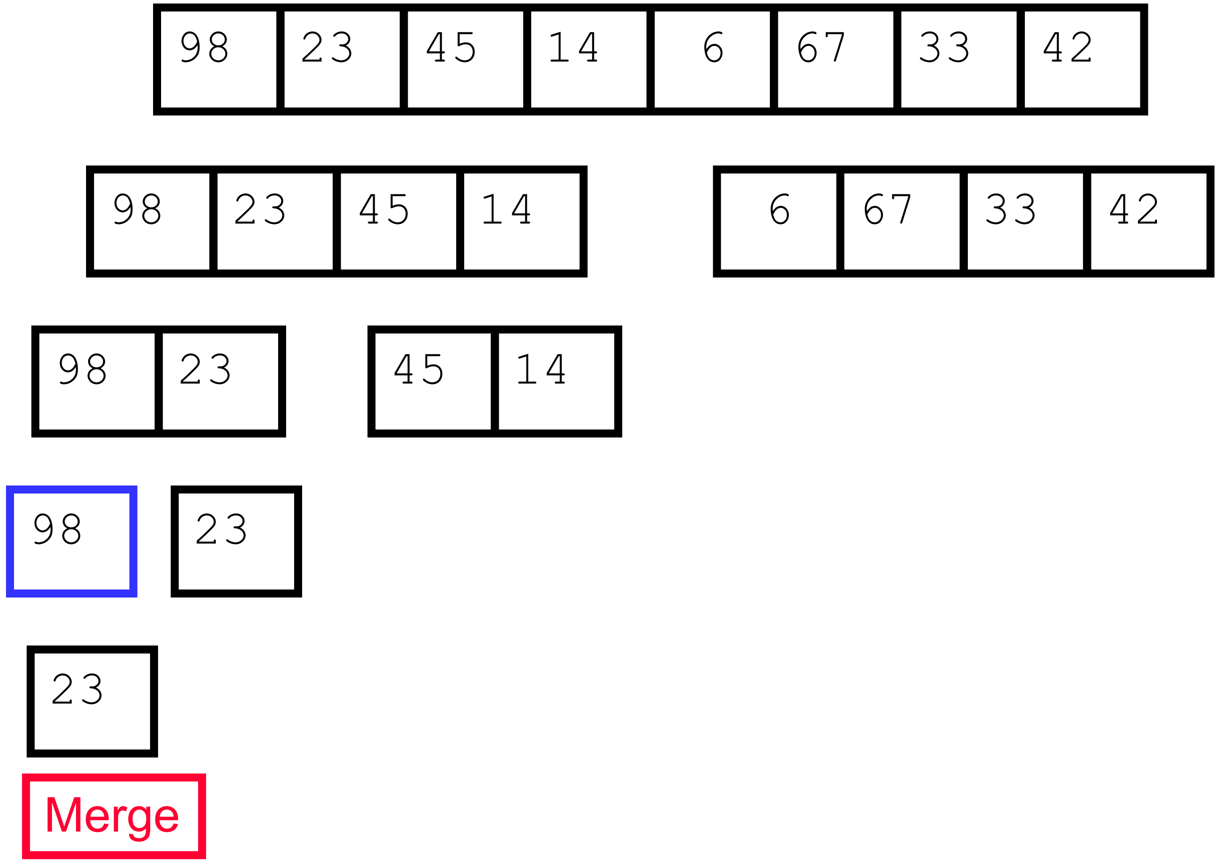

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42

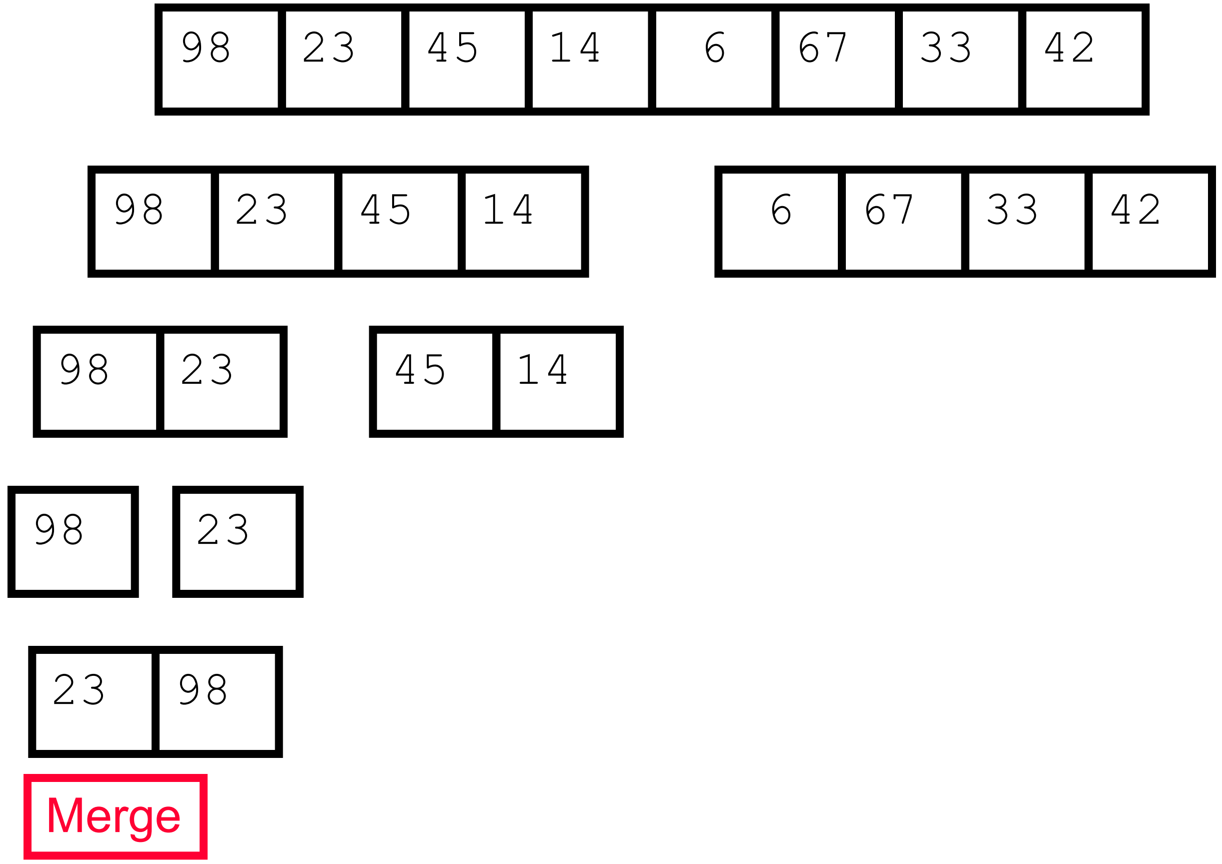

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

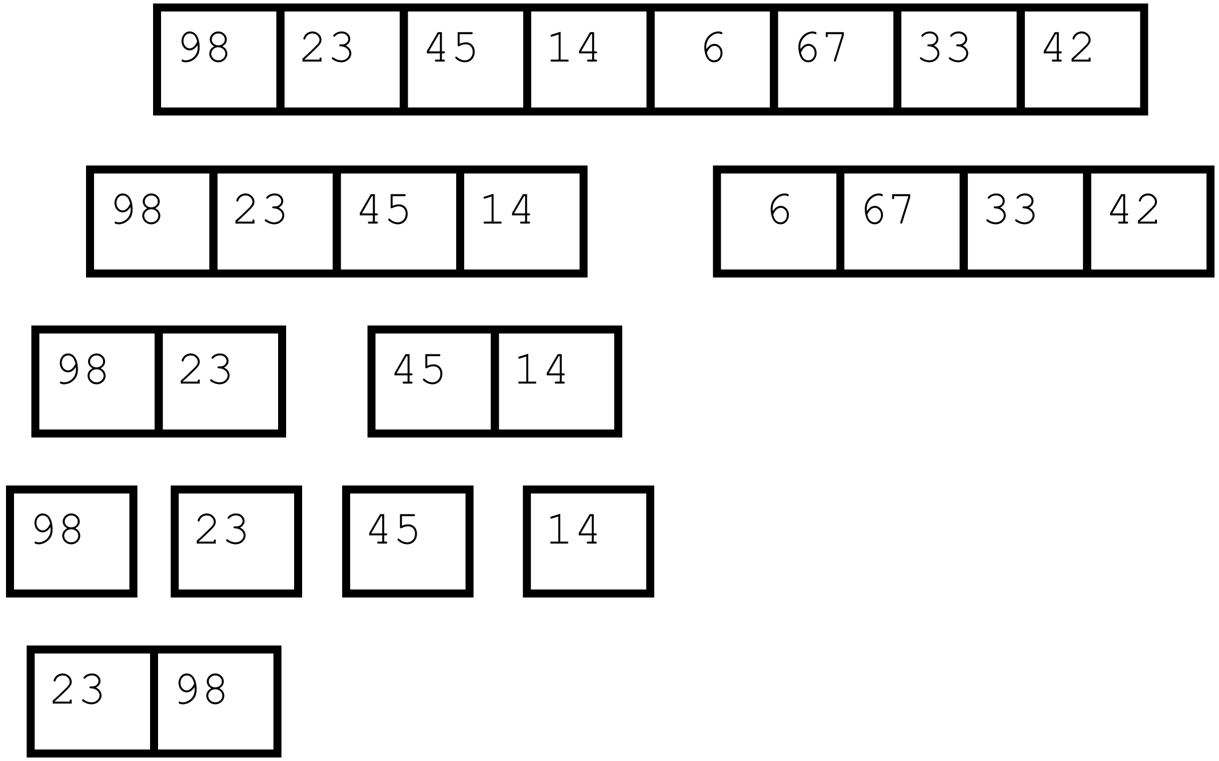

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

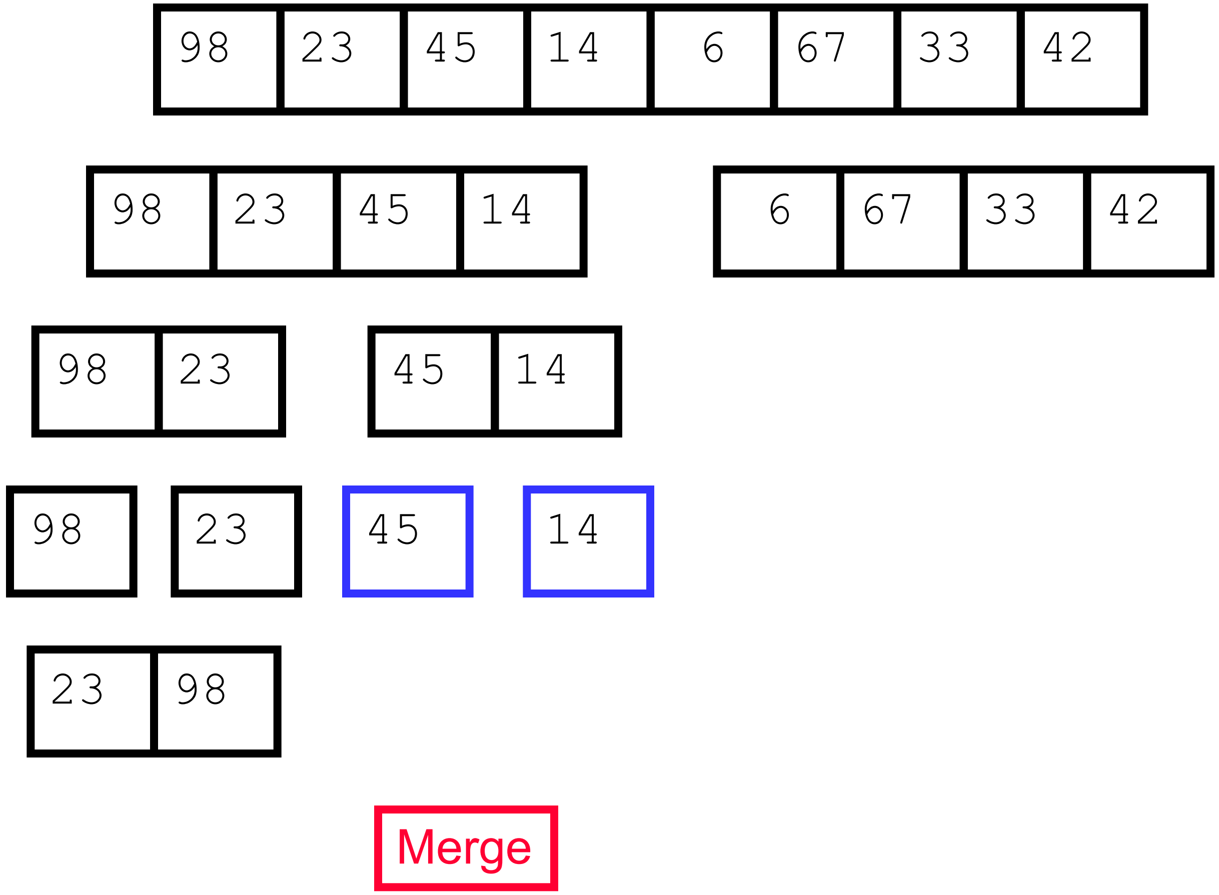


Merge

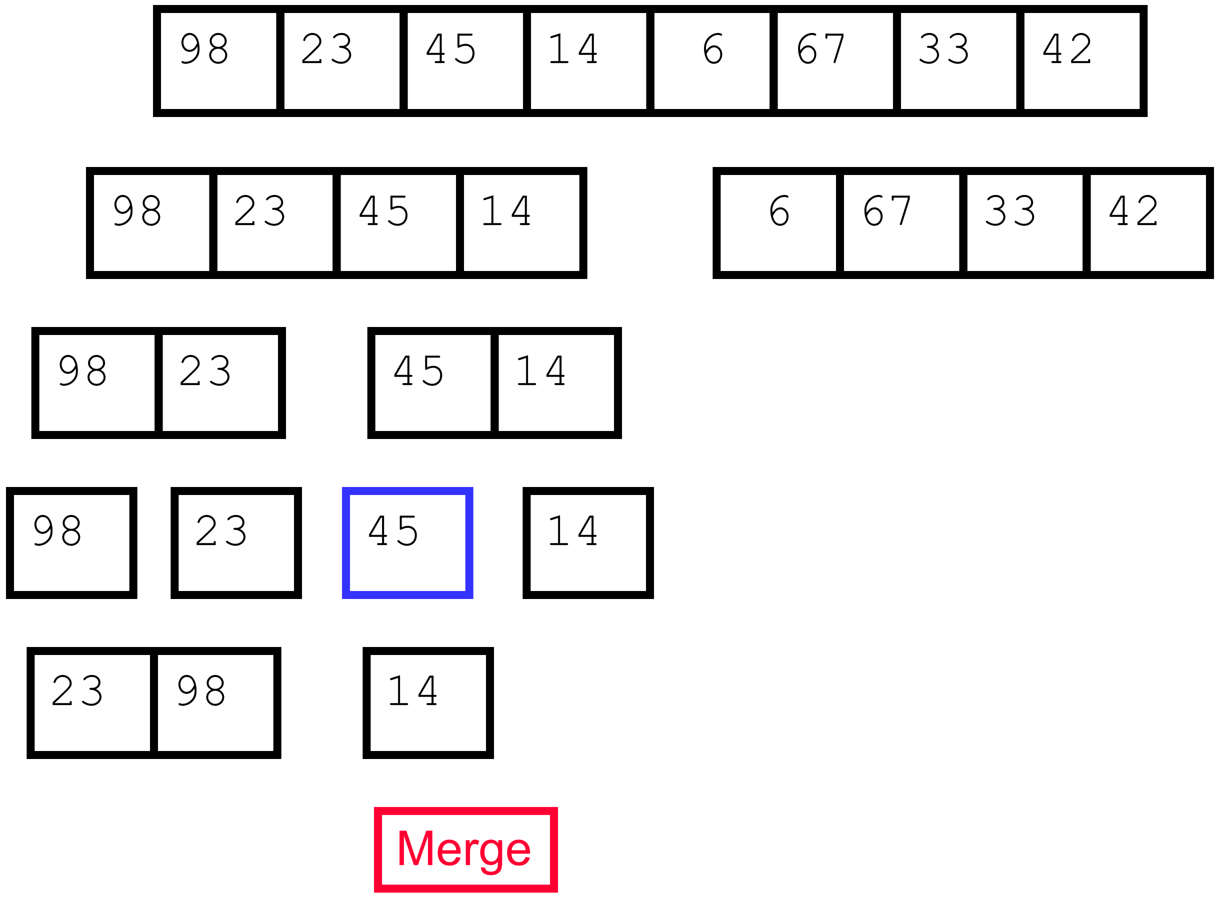

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

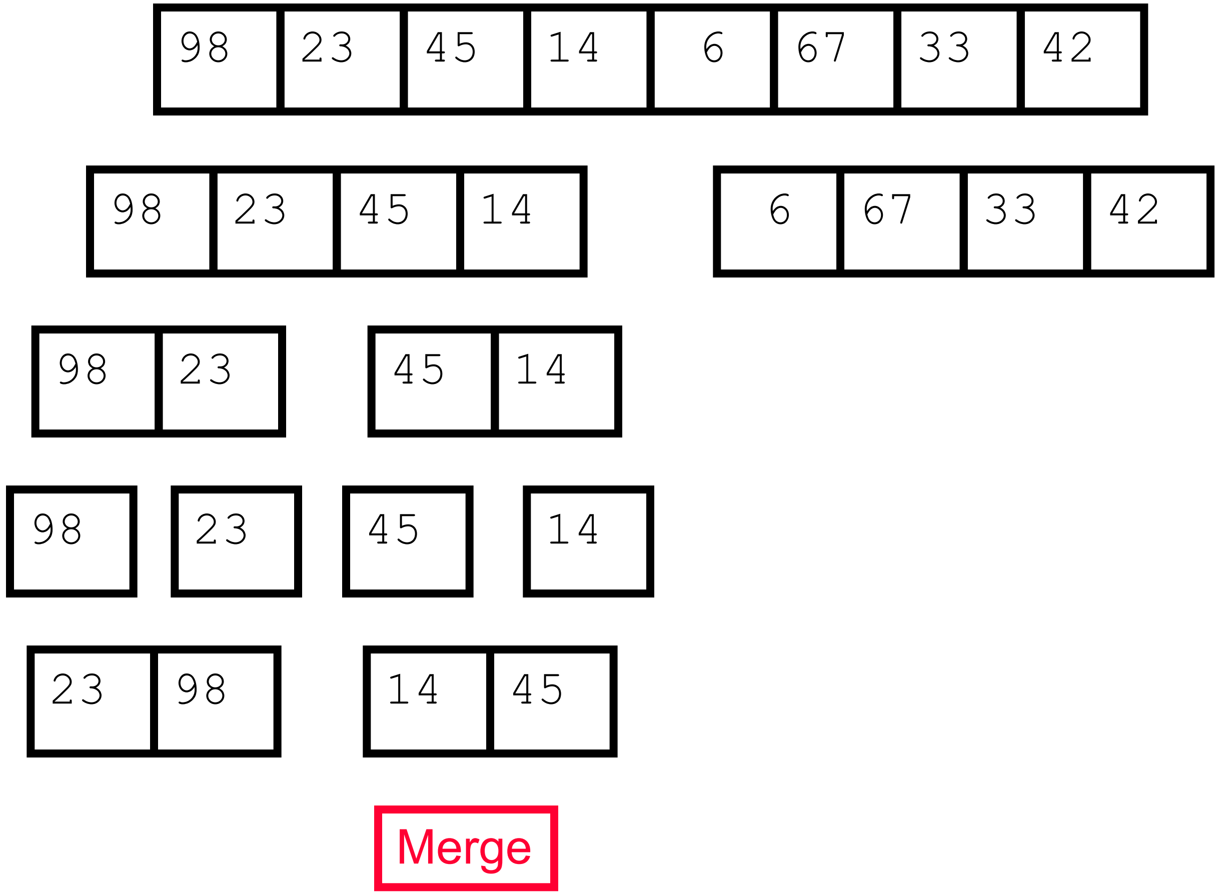

98
23
45
14
6
67
33
42
98
23
45
14
6
67
33
42
98
23
45
14
98
23
23
98
Merge

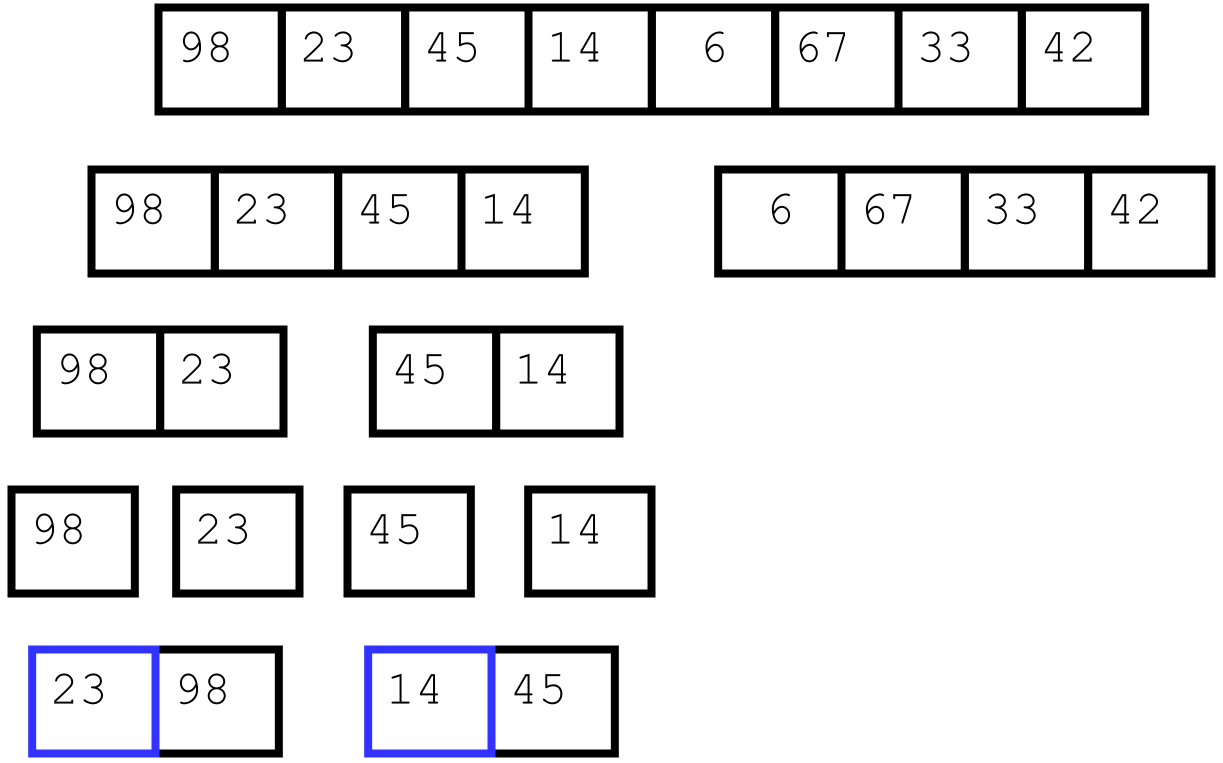

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98



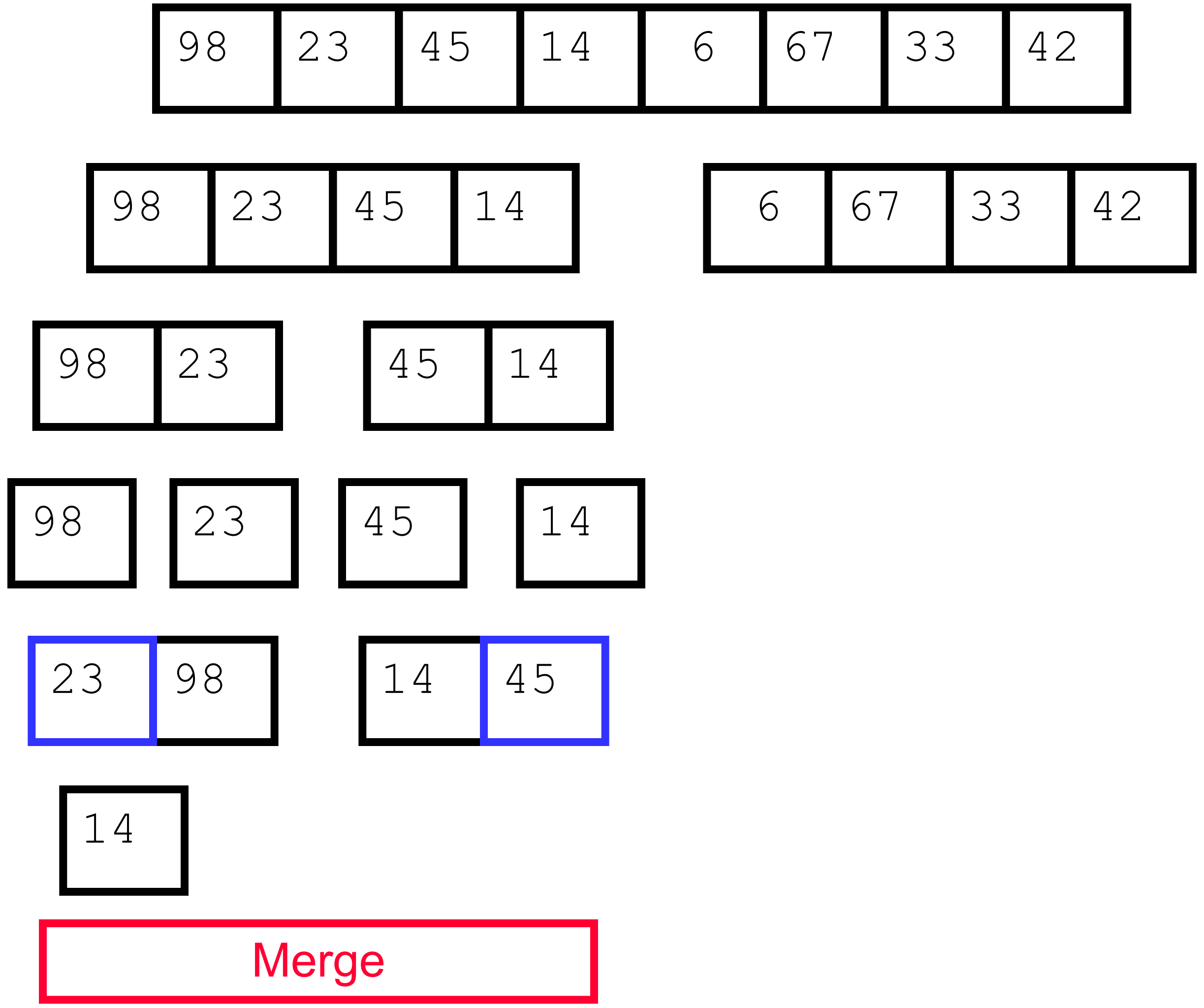
98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
Merge

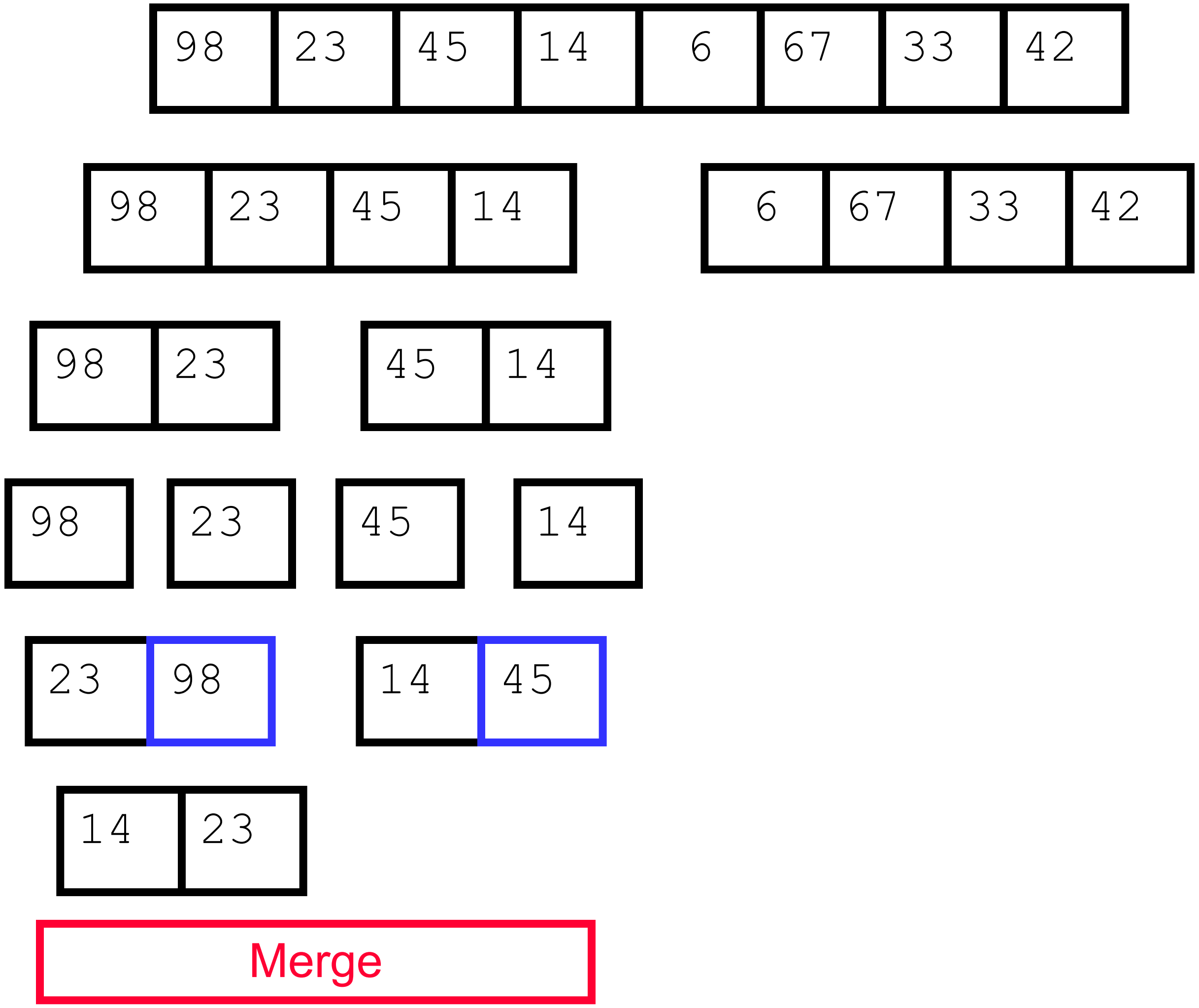

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14
Merge


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

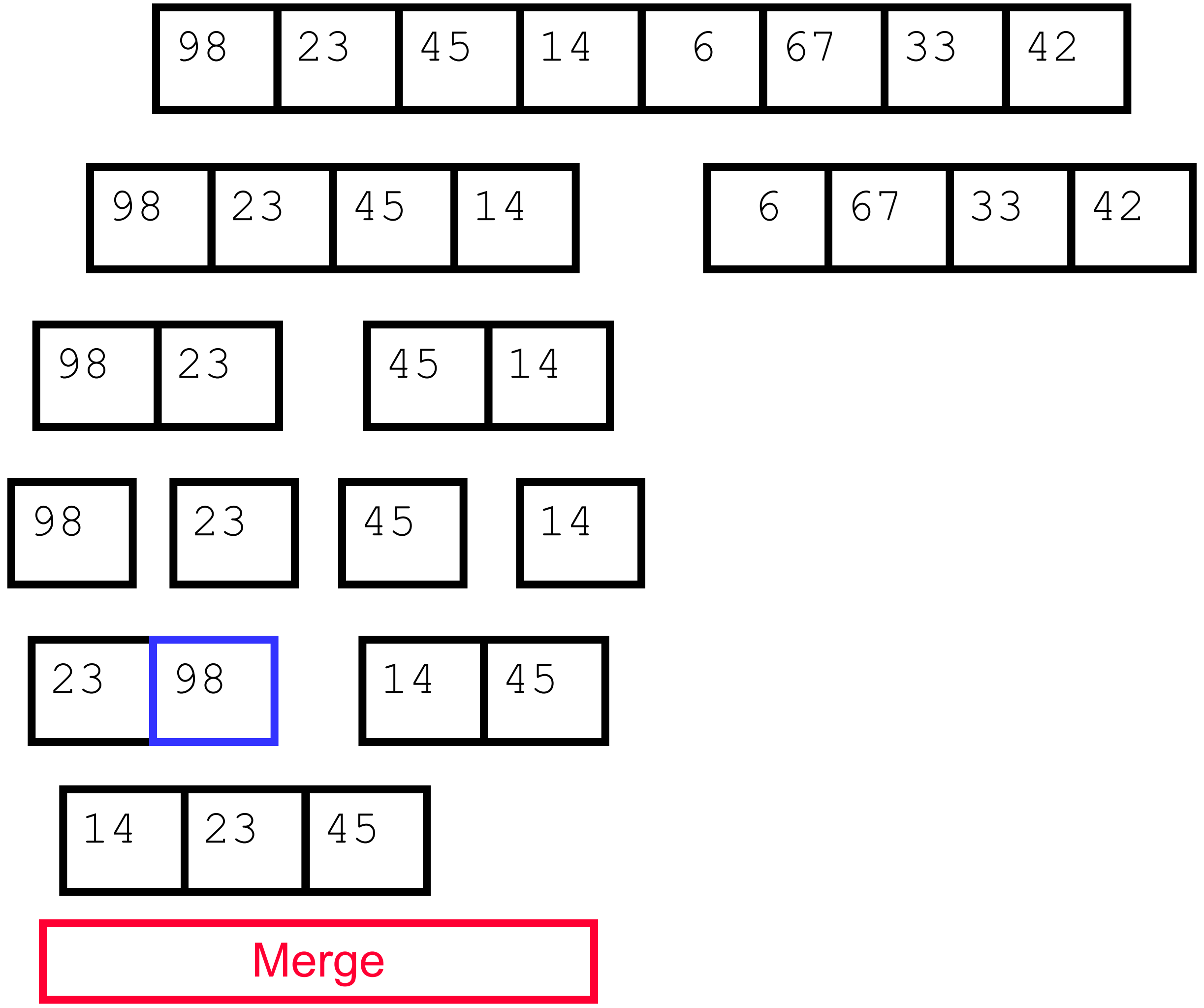

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45

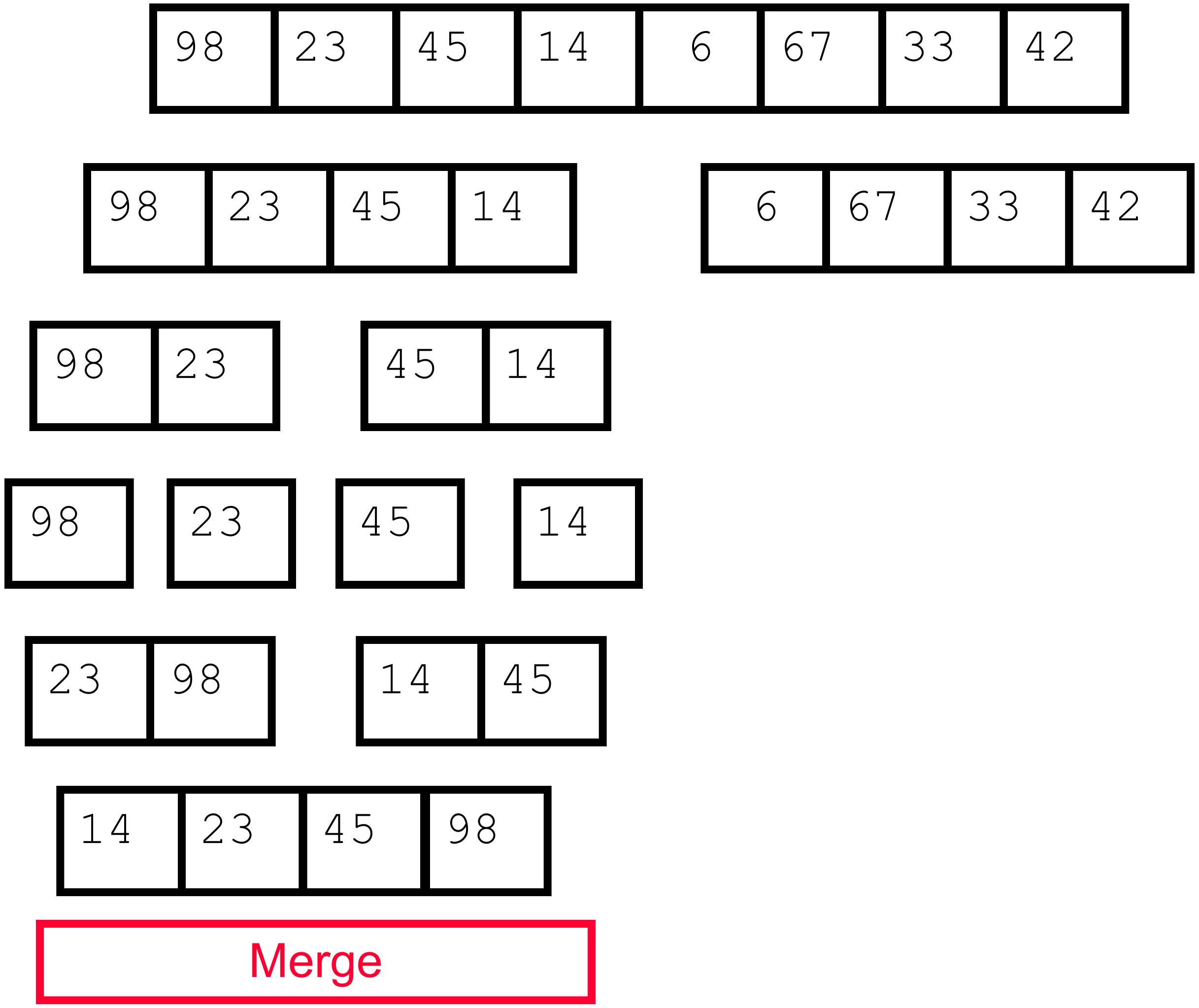
Merge

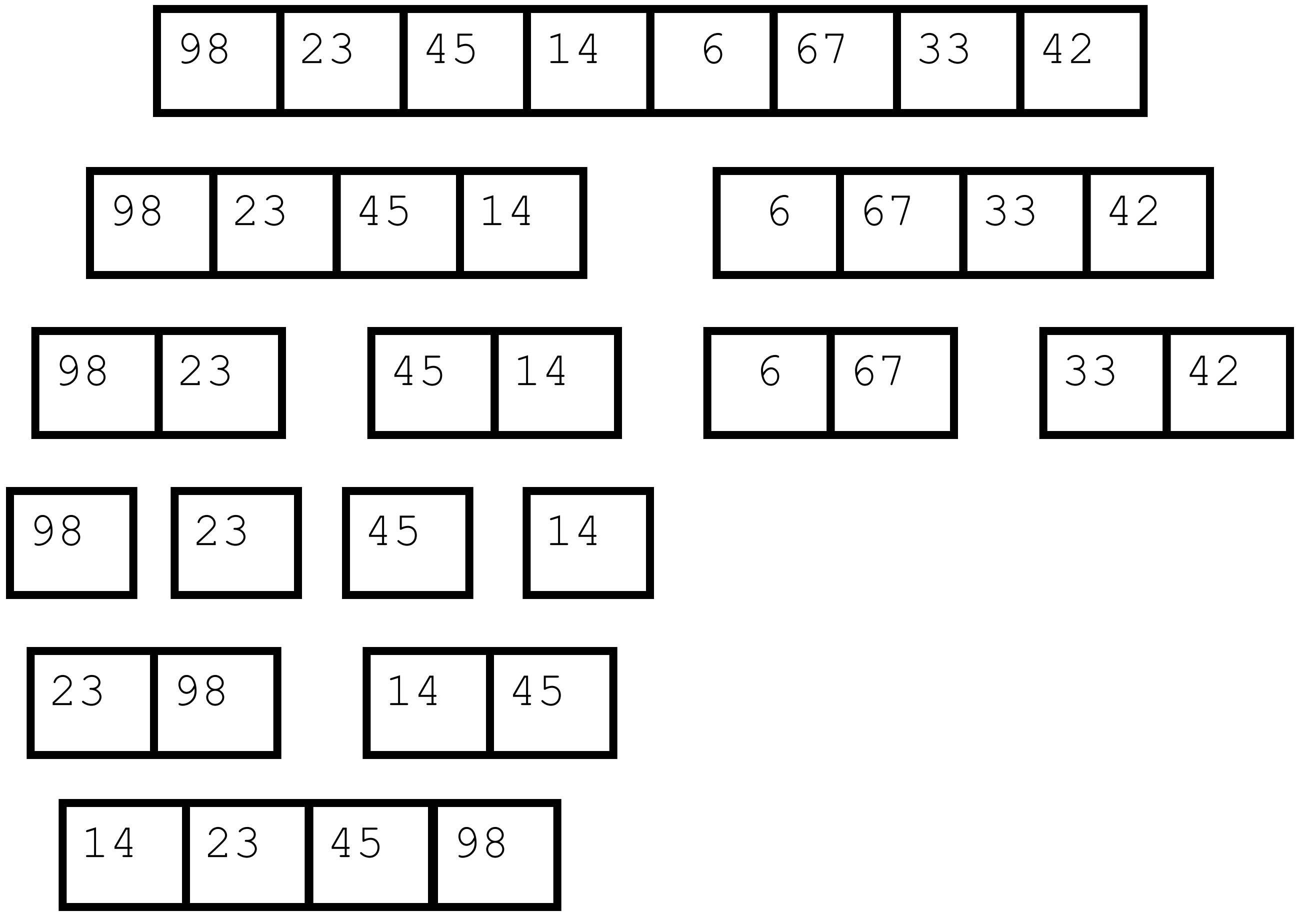

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

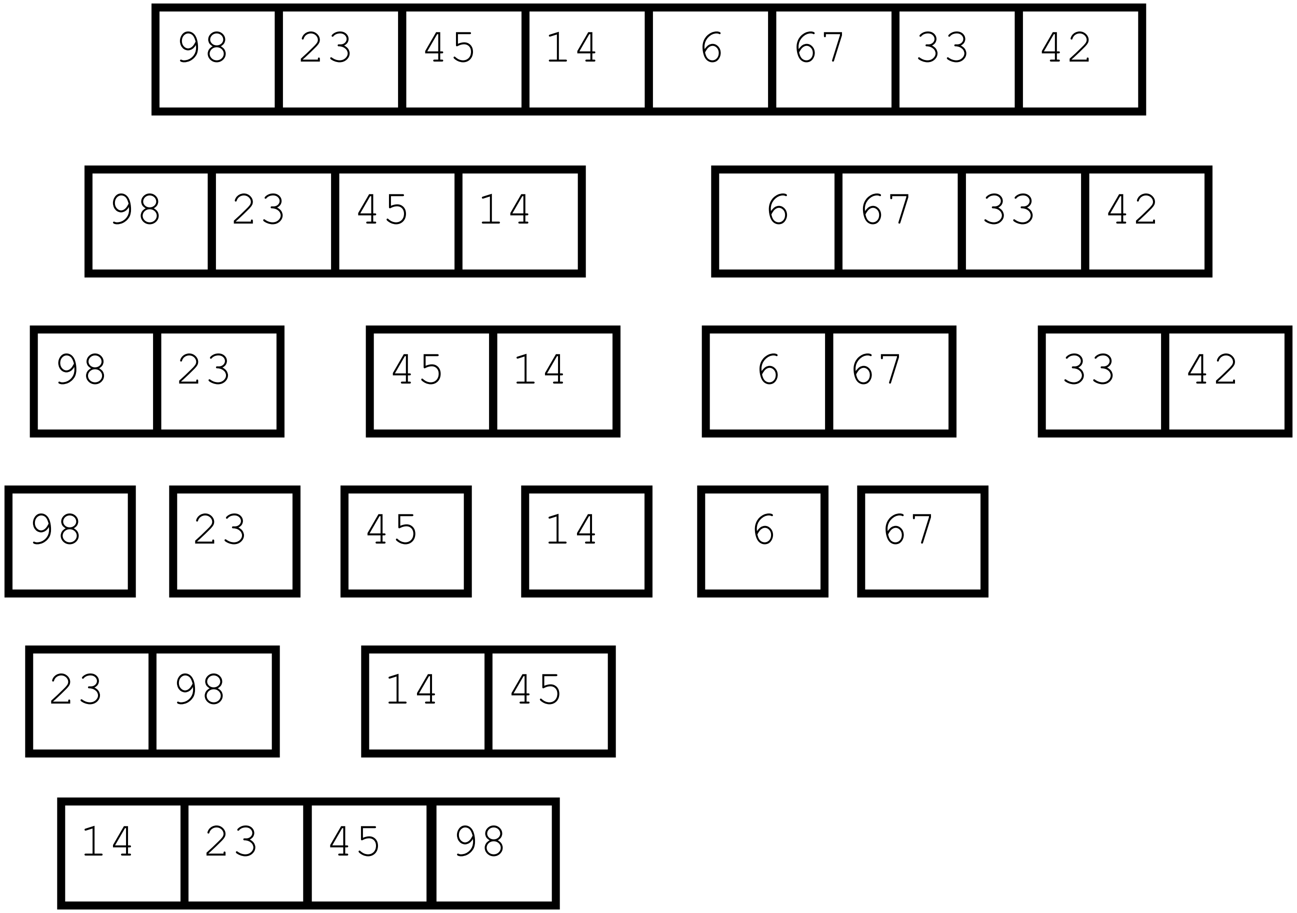


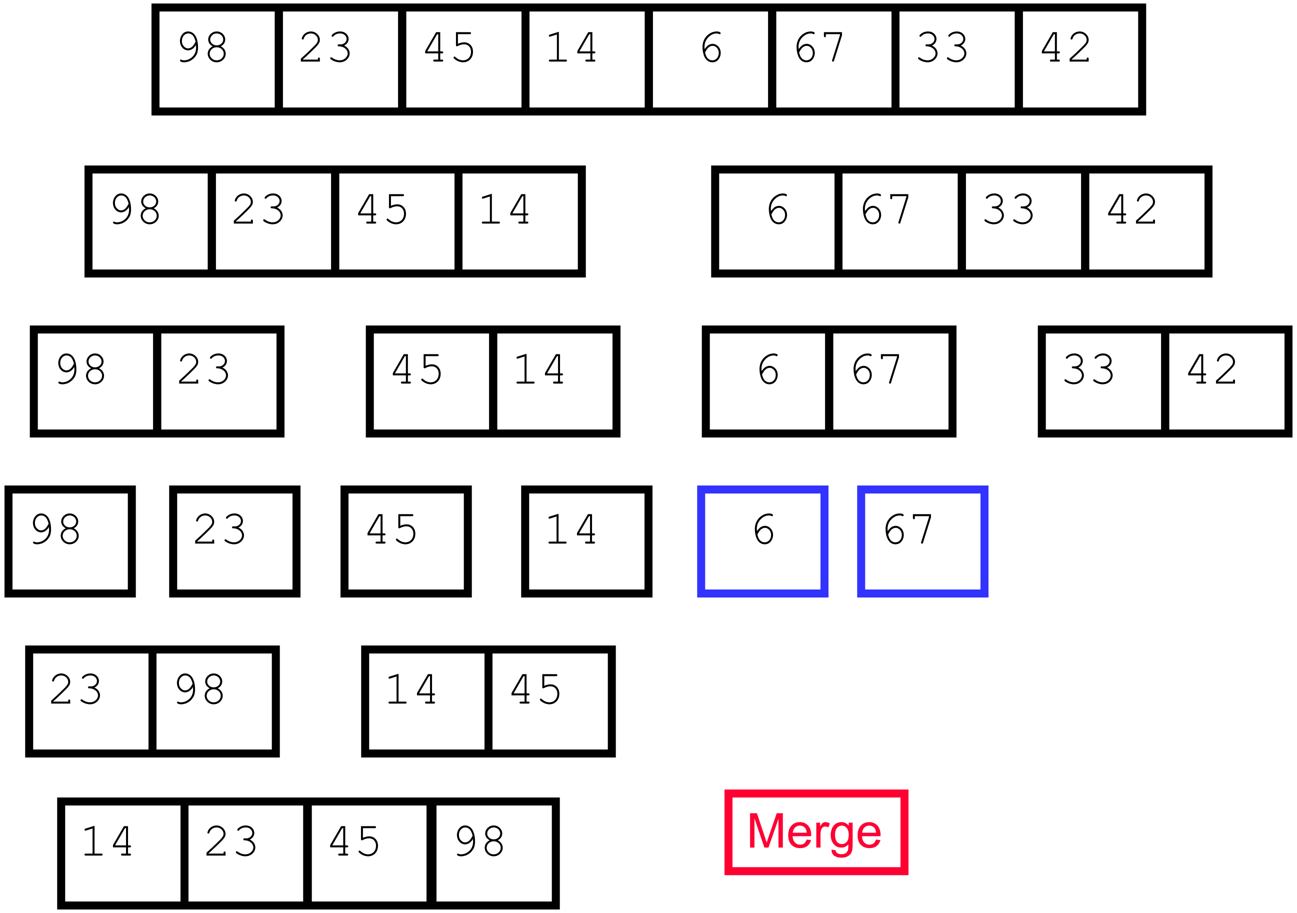
98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

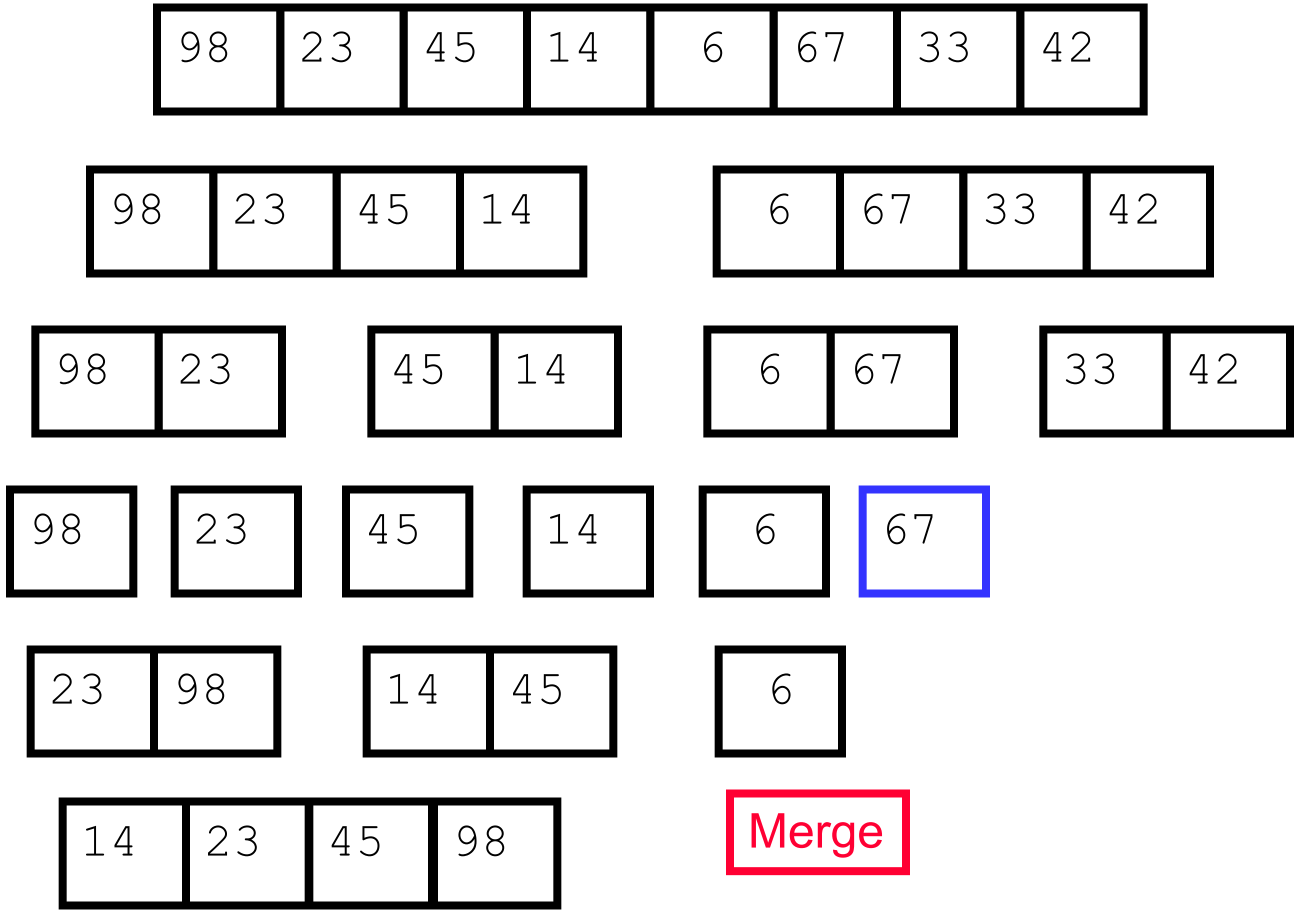

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

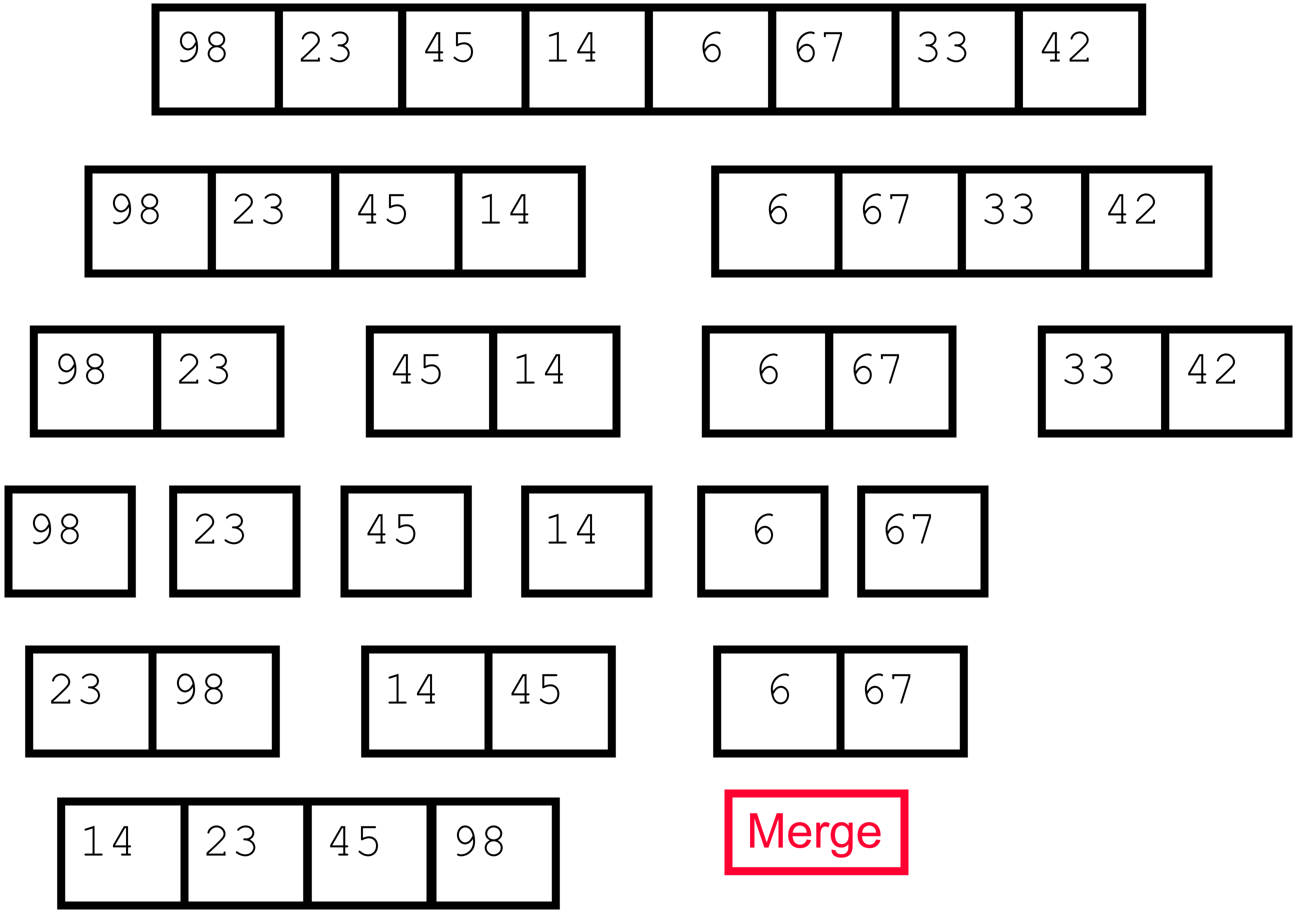

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

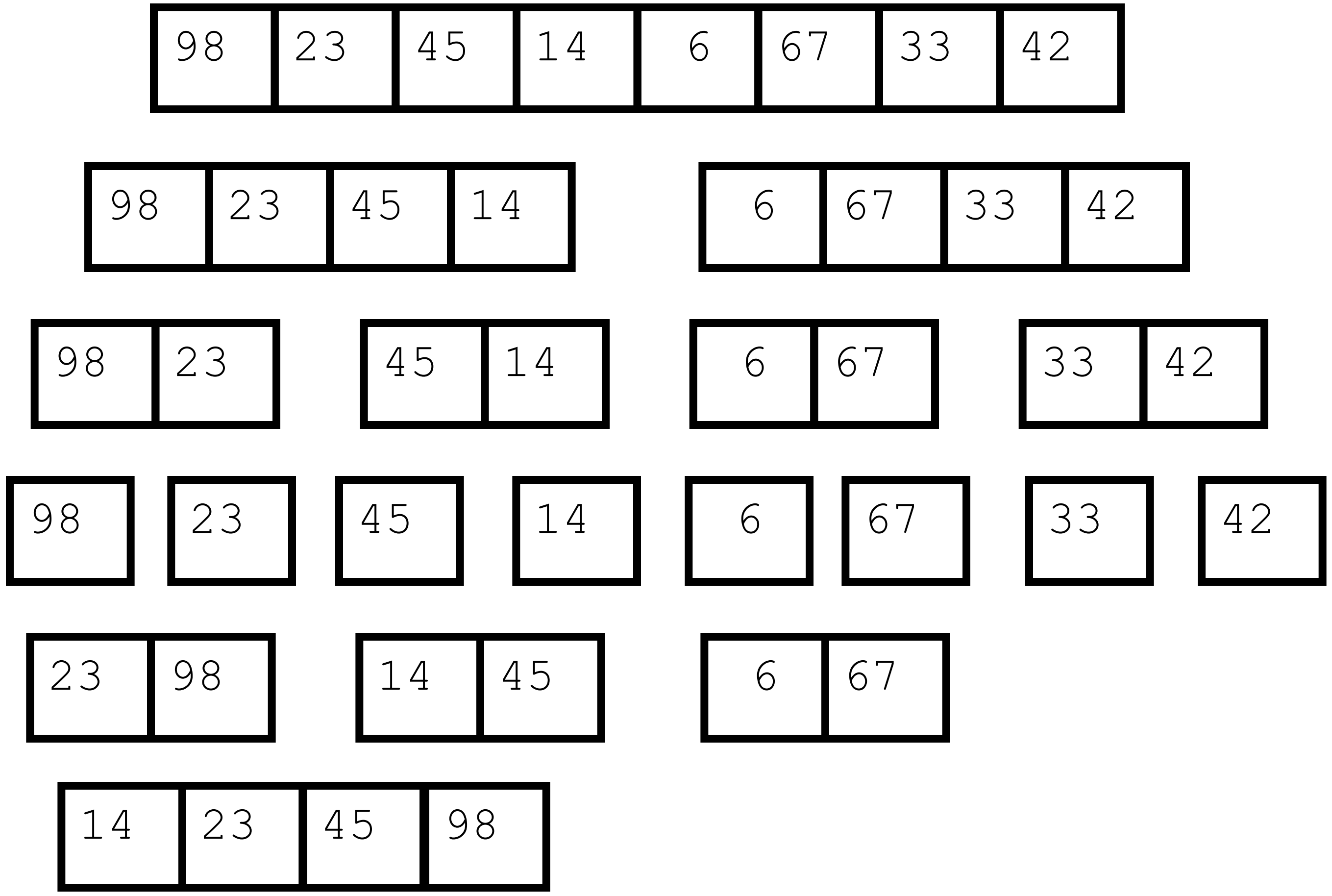

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

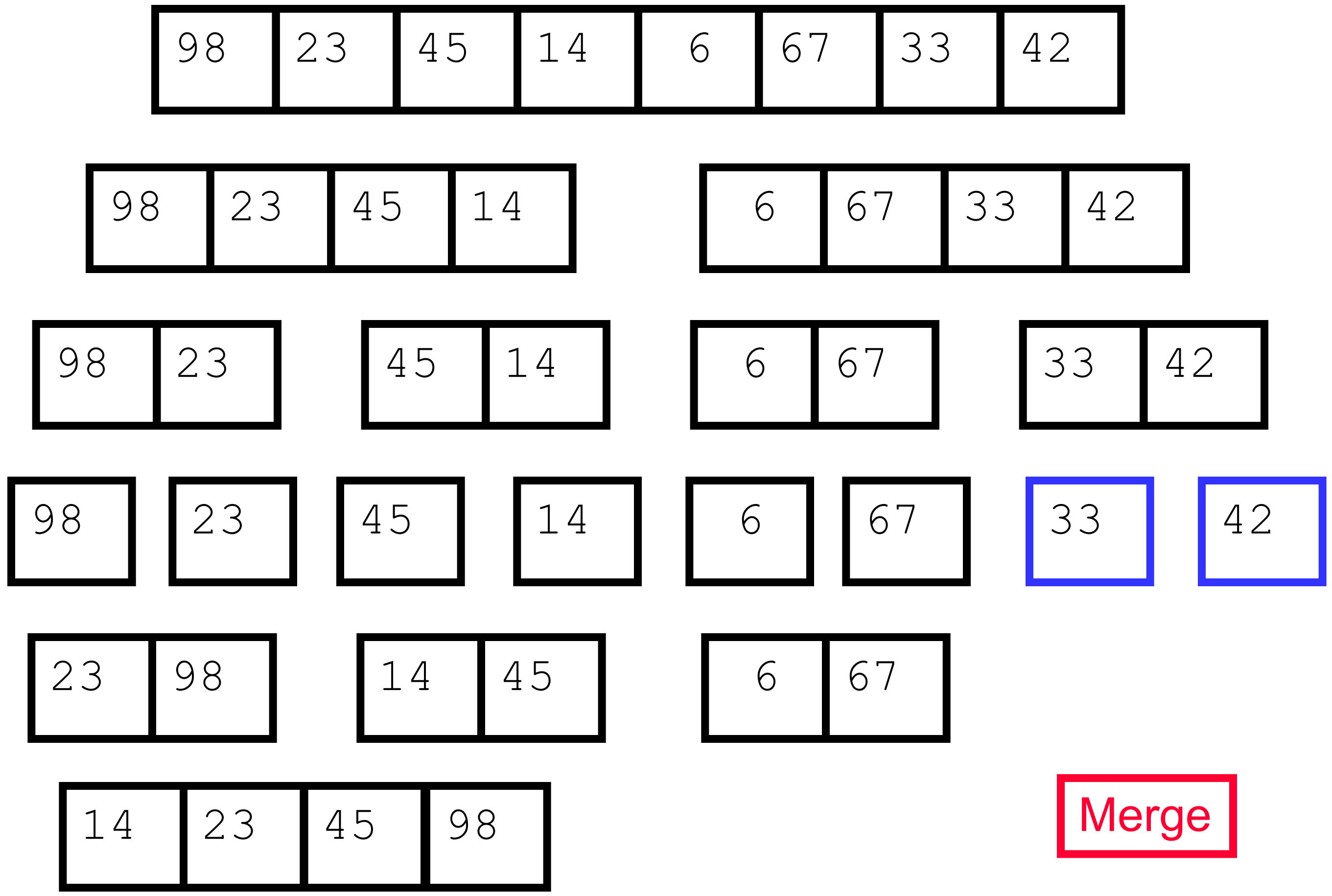

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

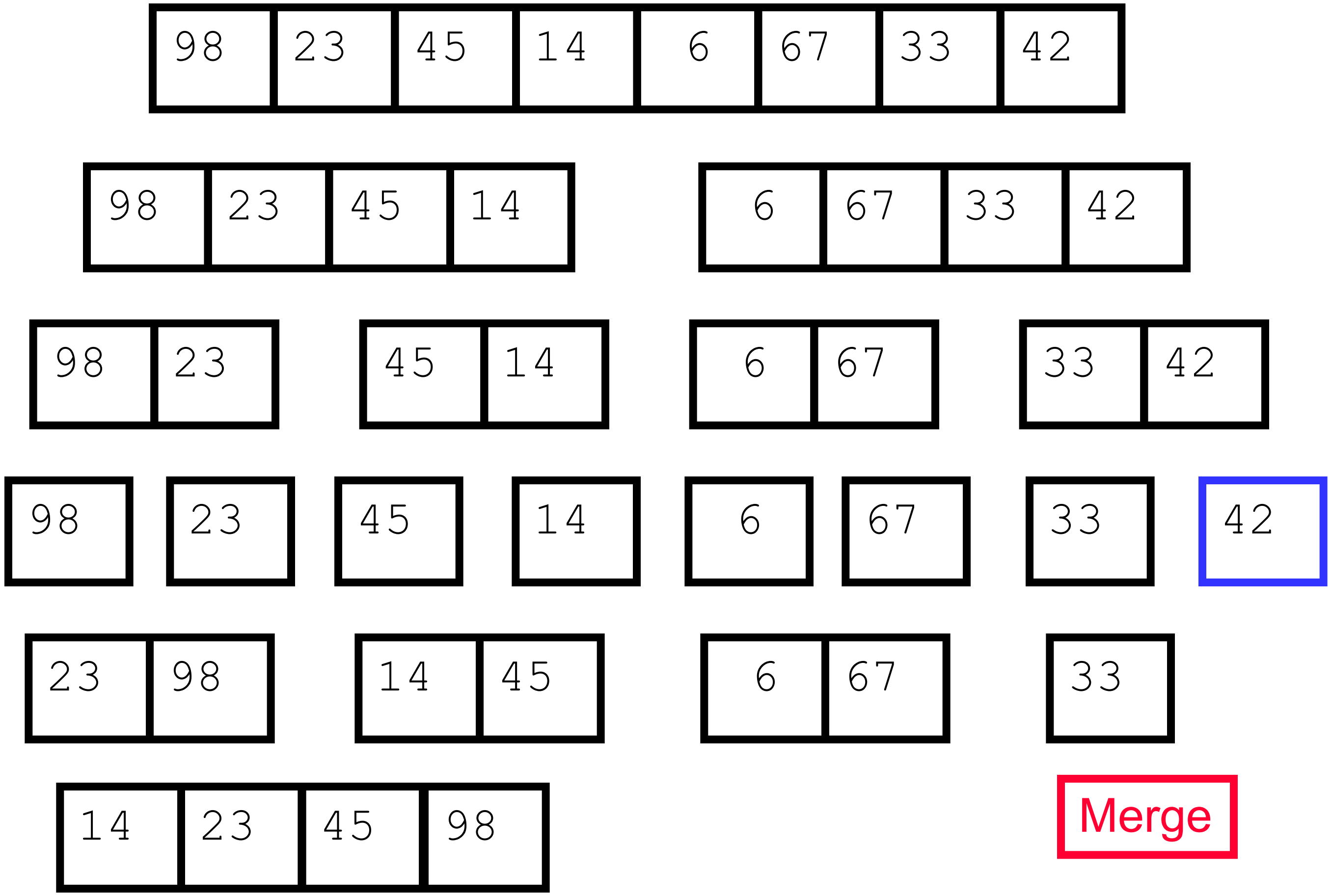

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

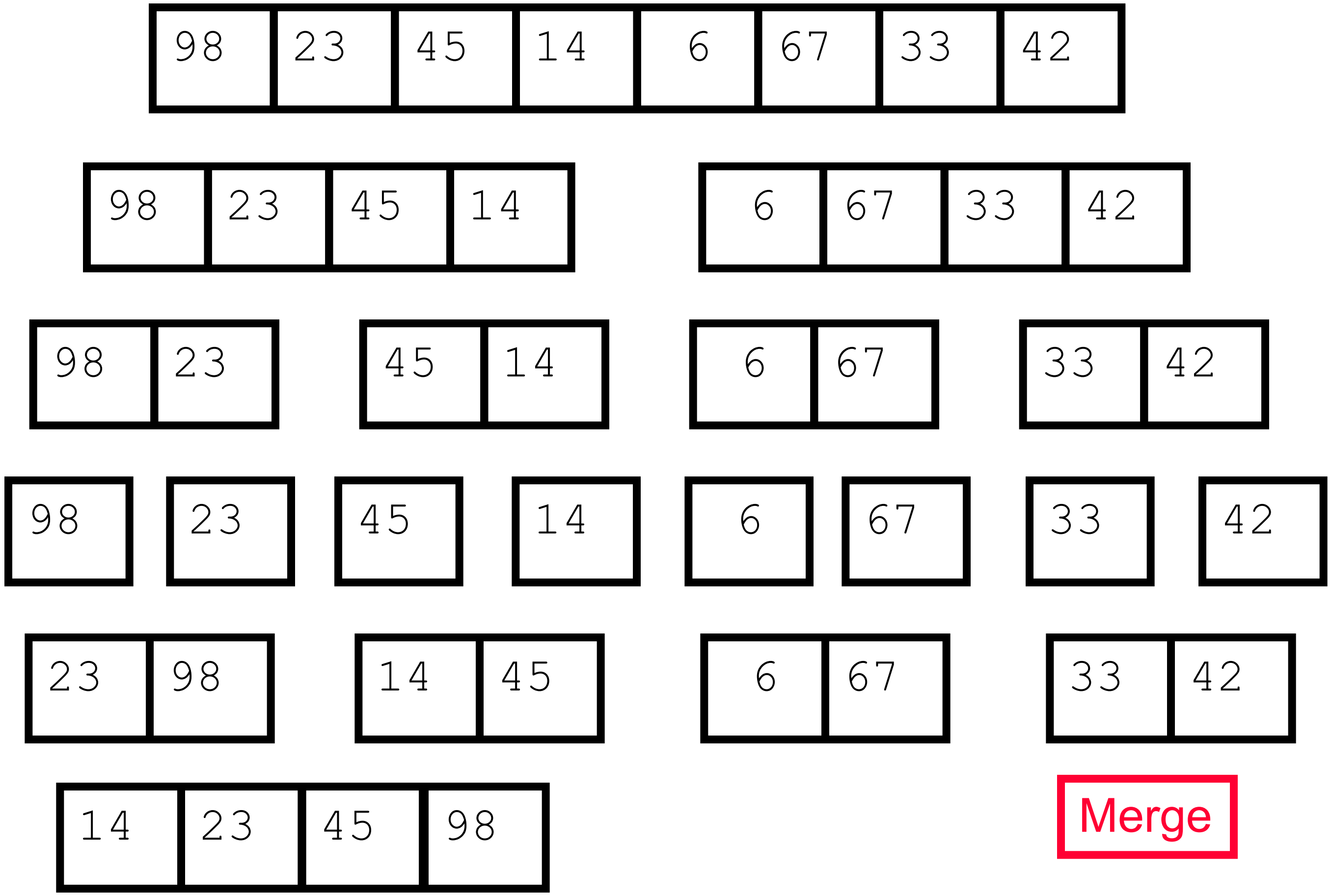

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge

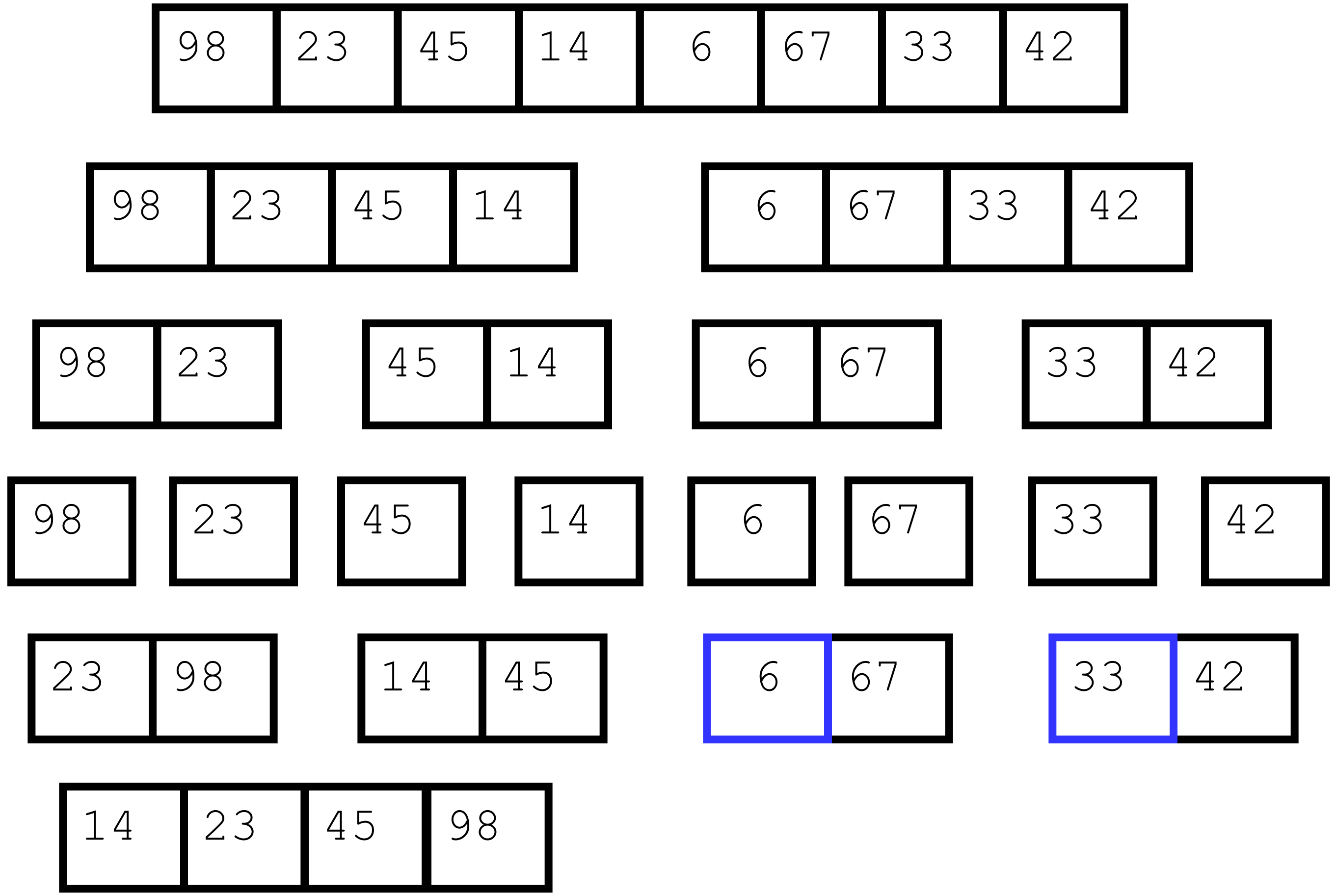

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6 67
14 23 45 98
Merge


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

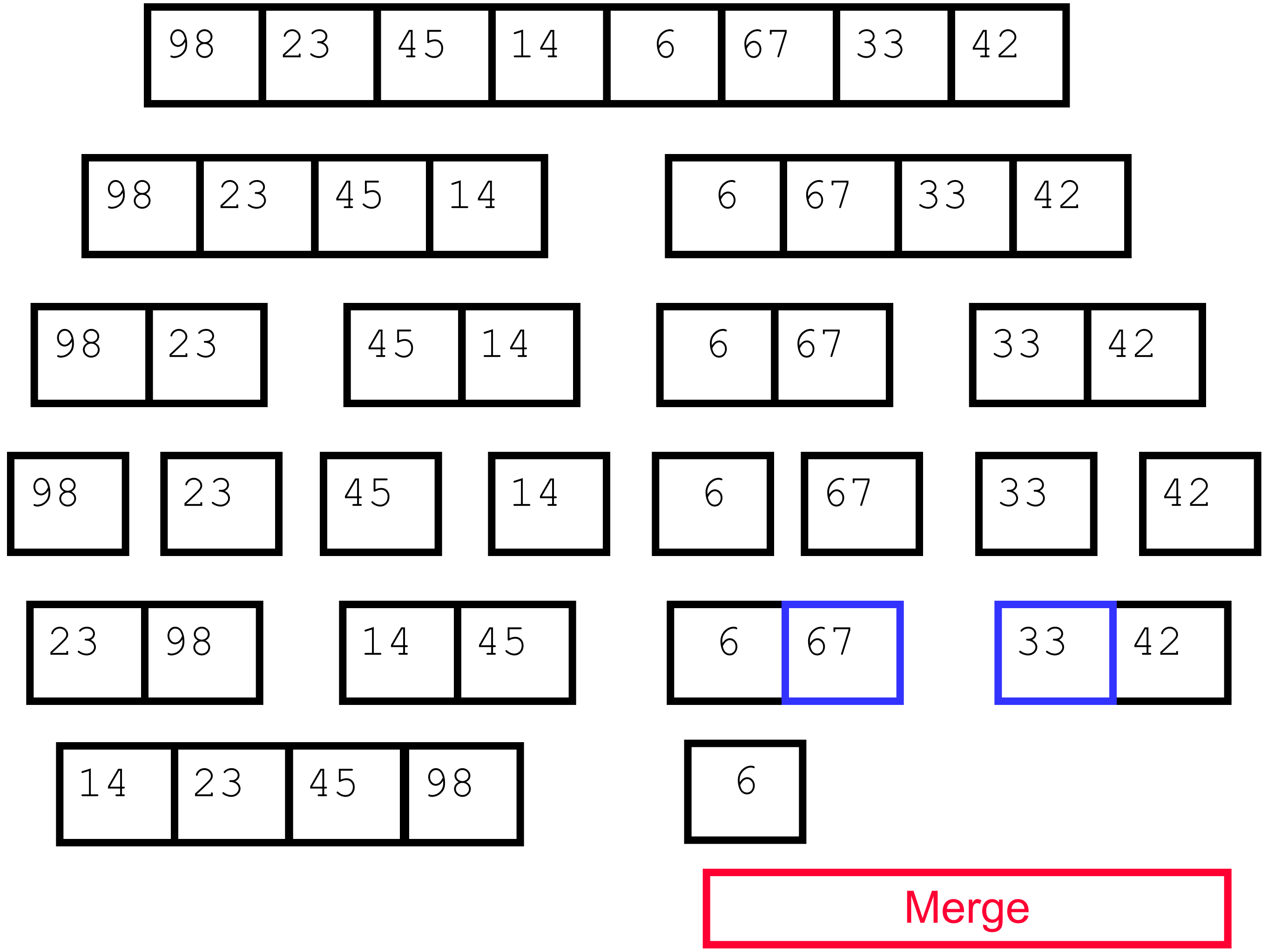

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98
Merge

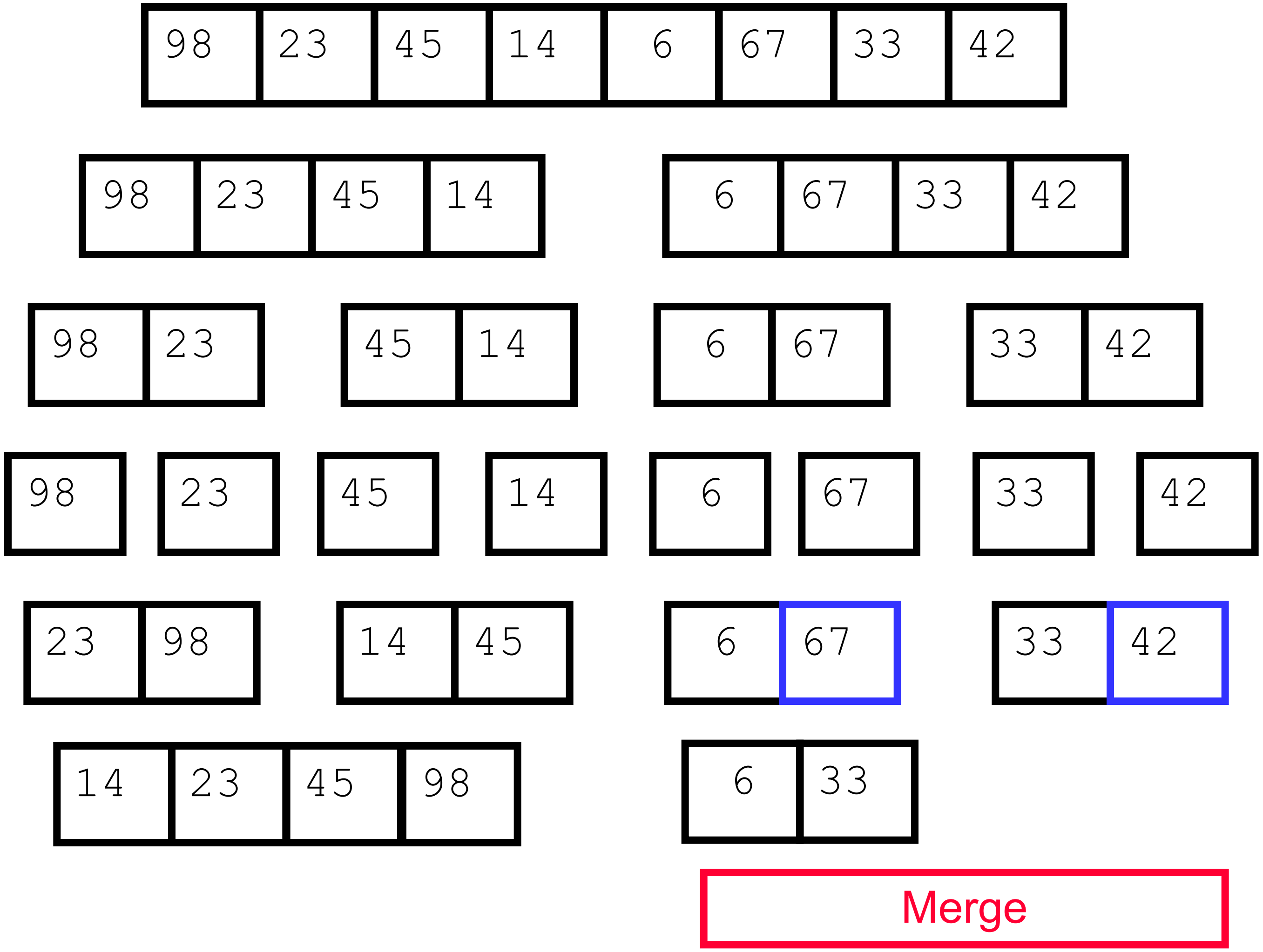

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge

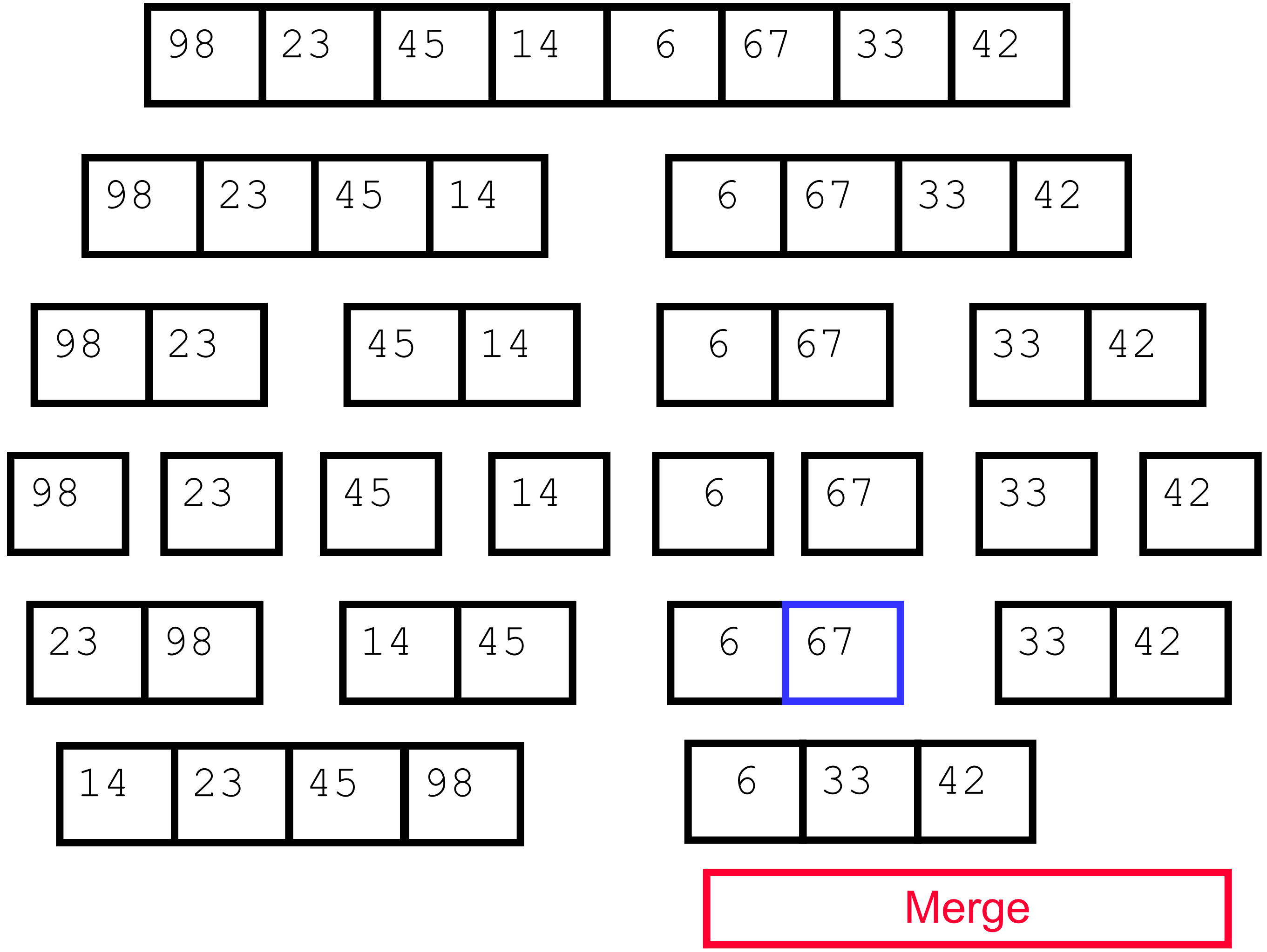

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge


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

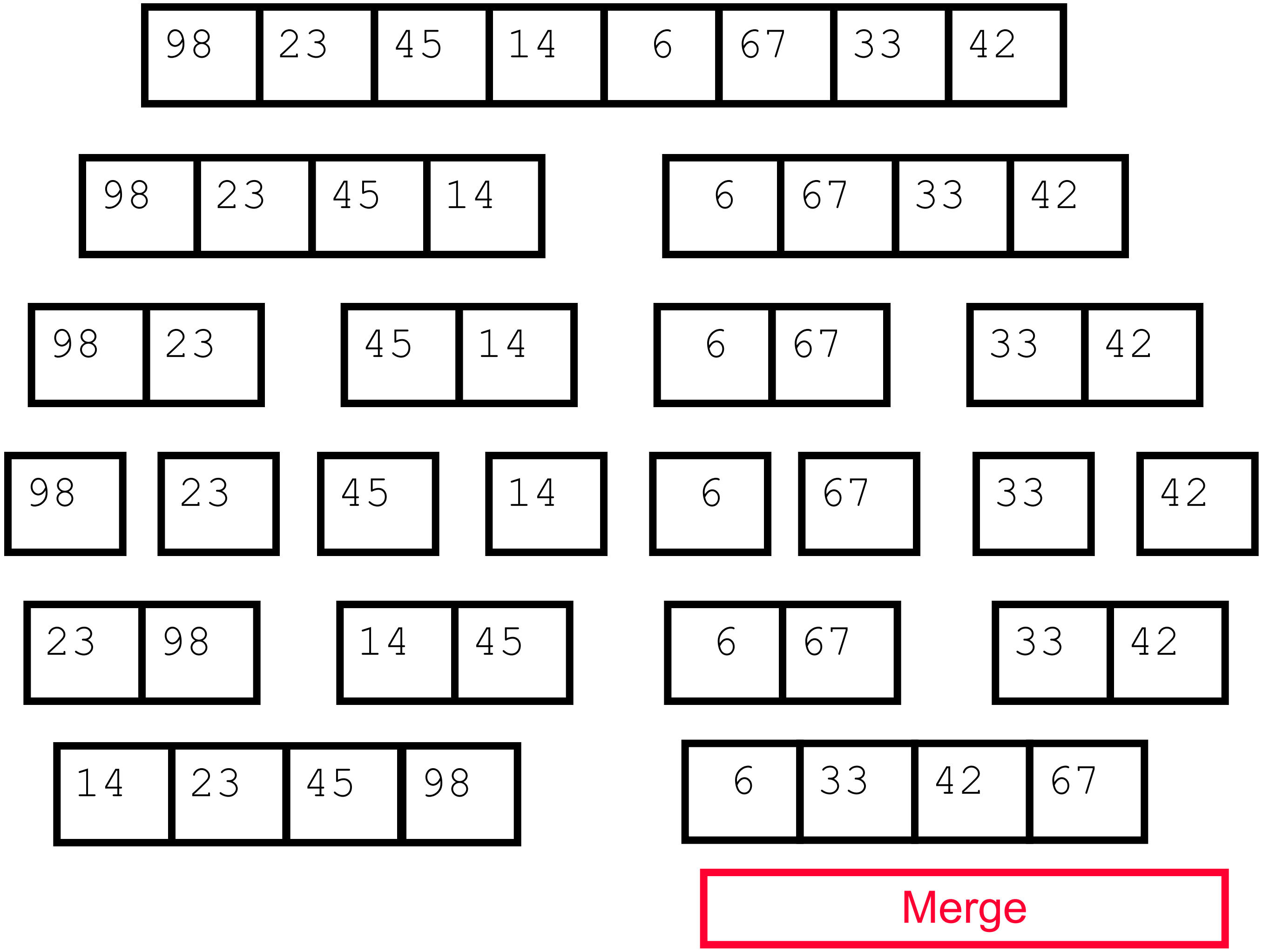
Merge

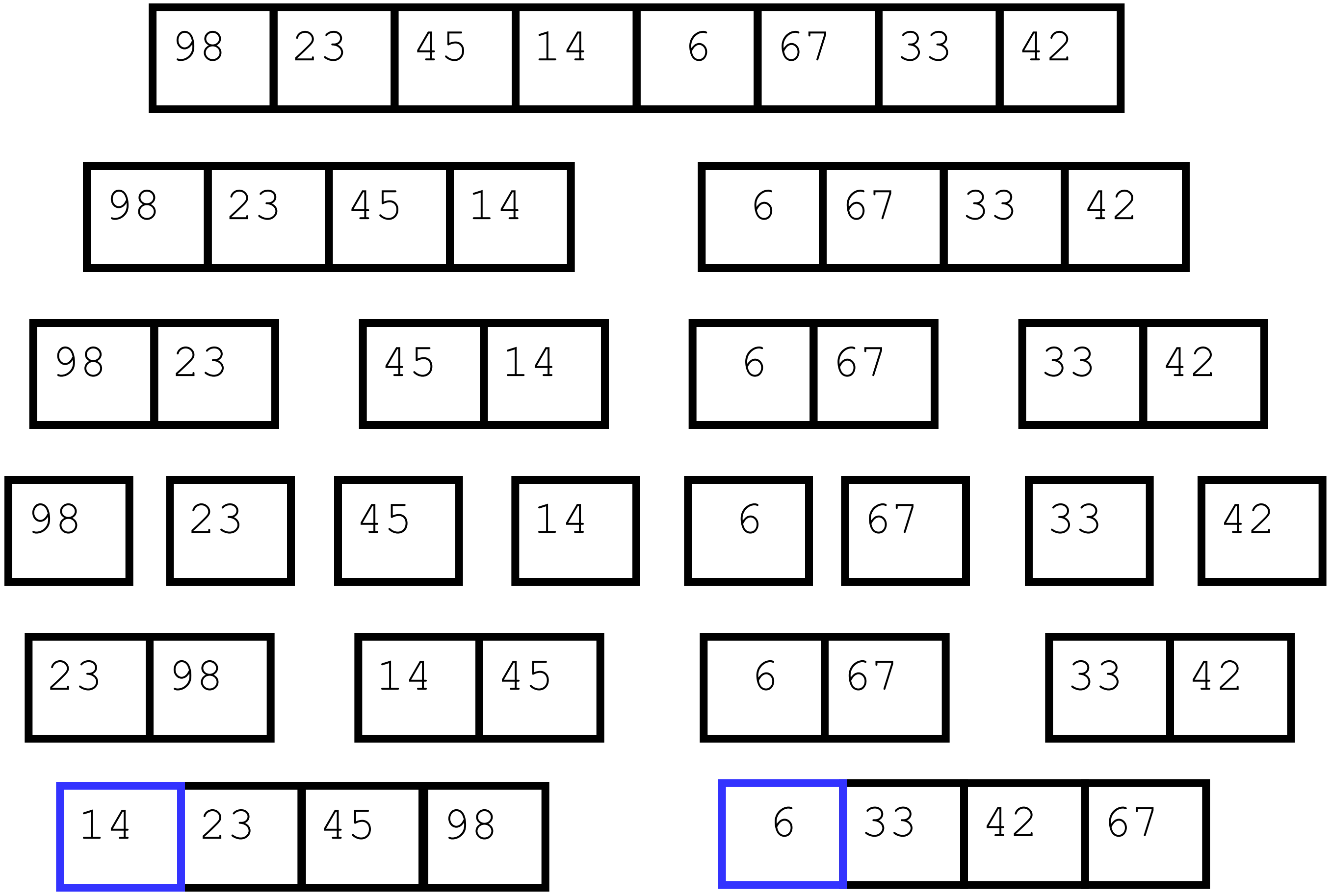

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6
Merge


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

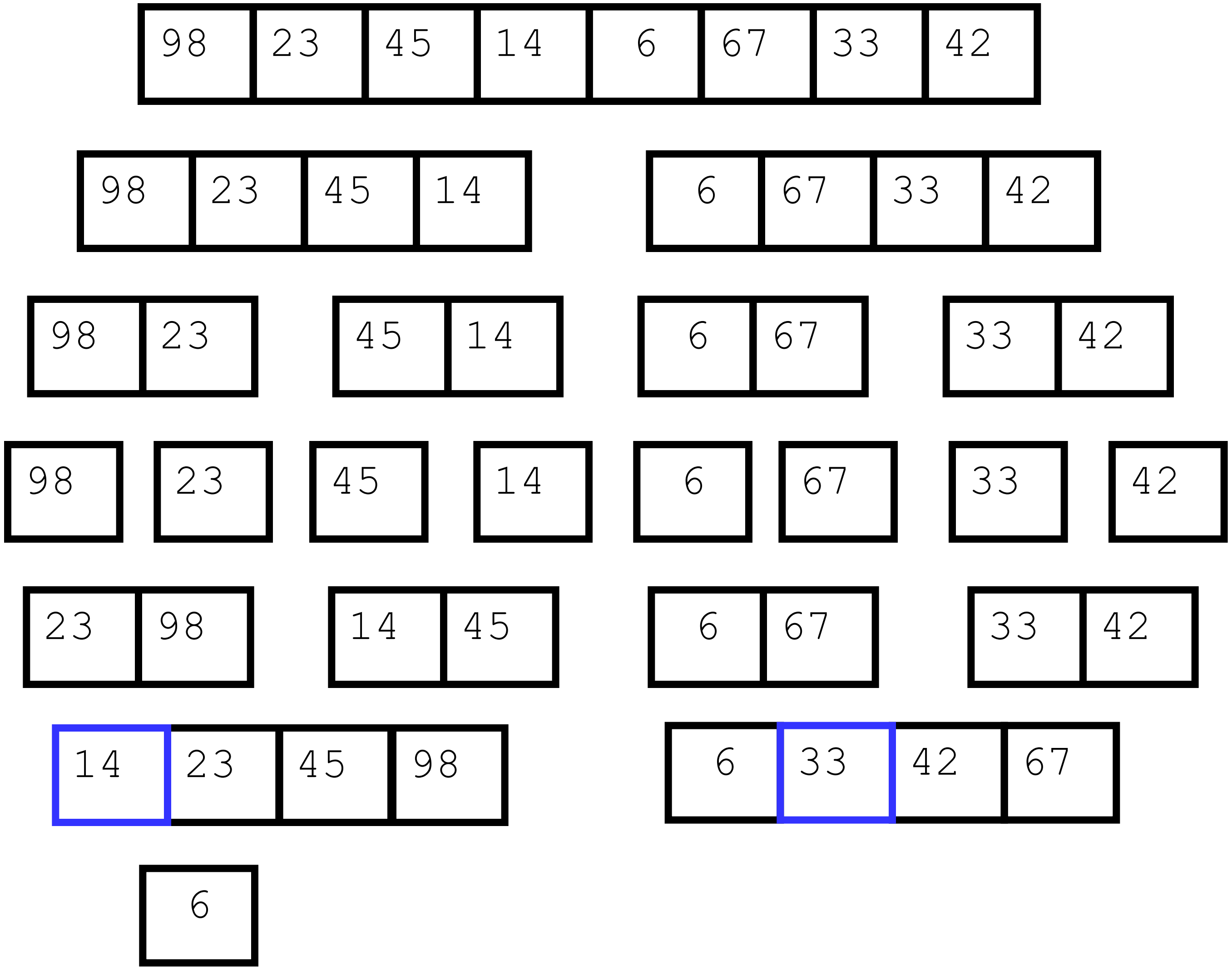


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42
Merge

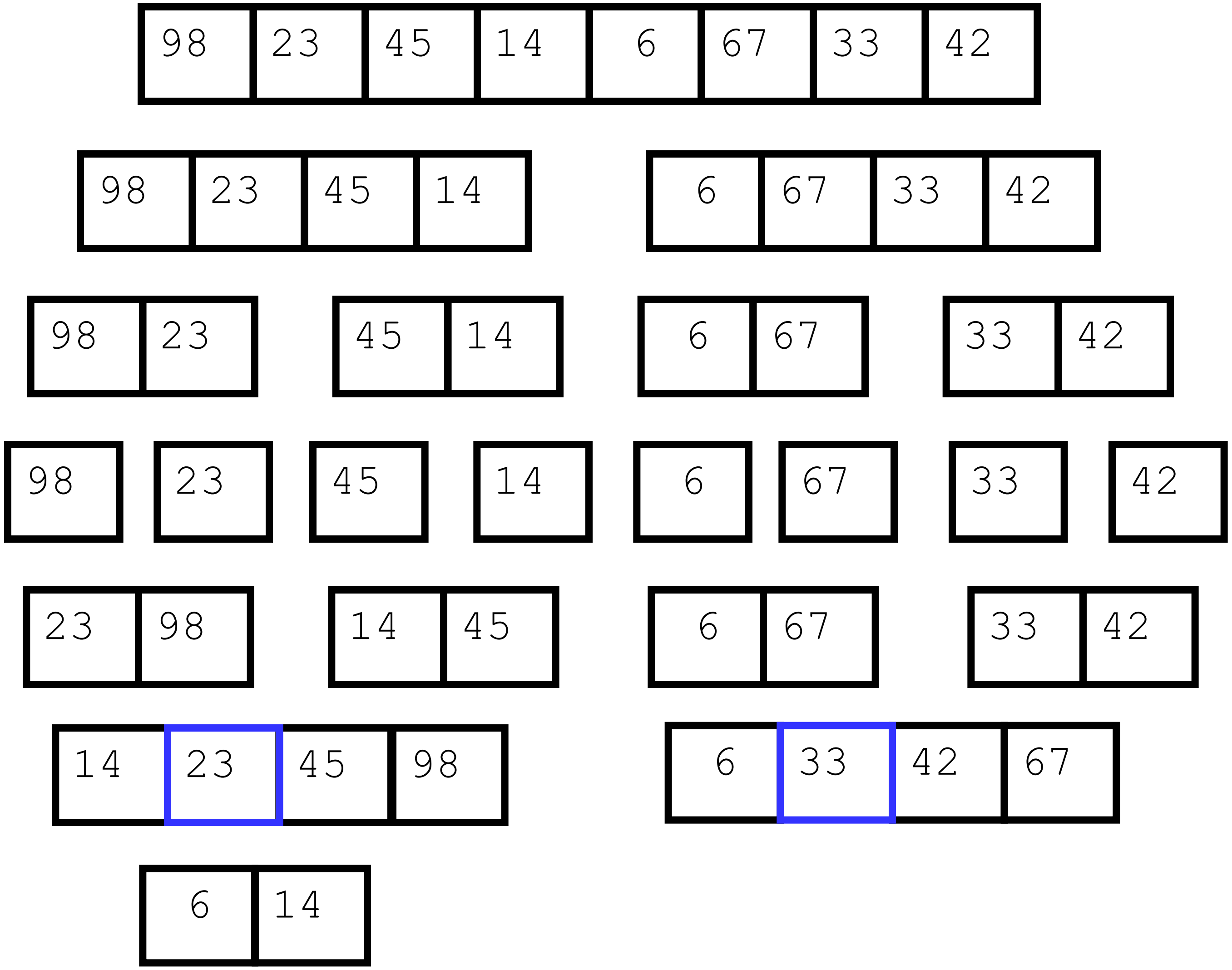

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge



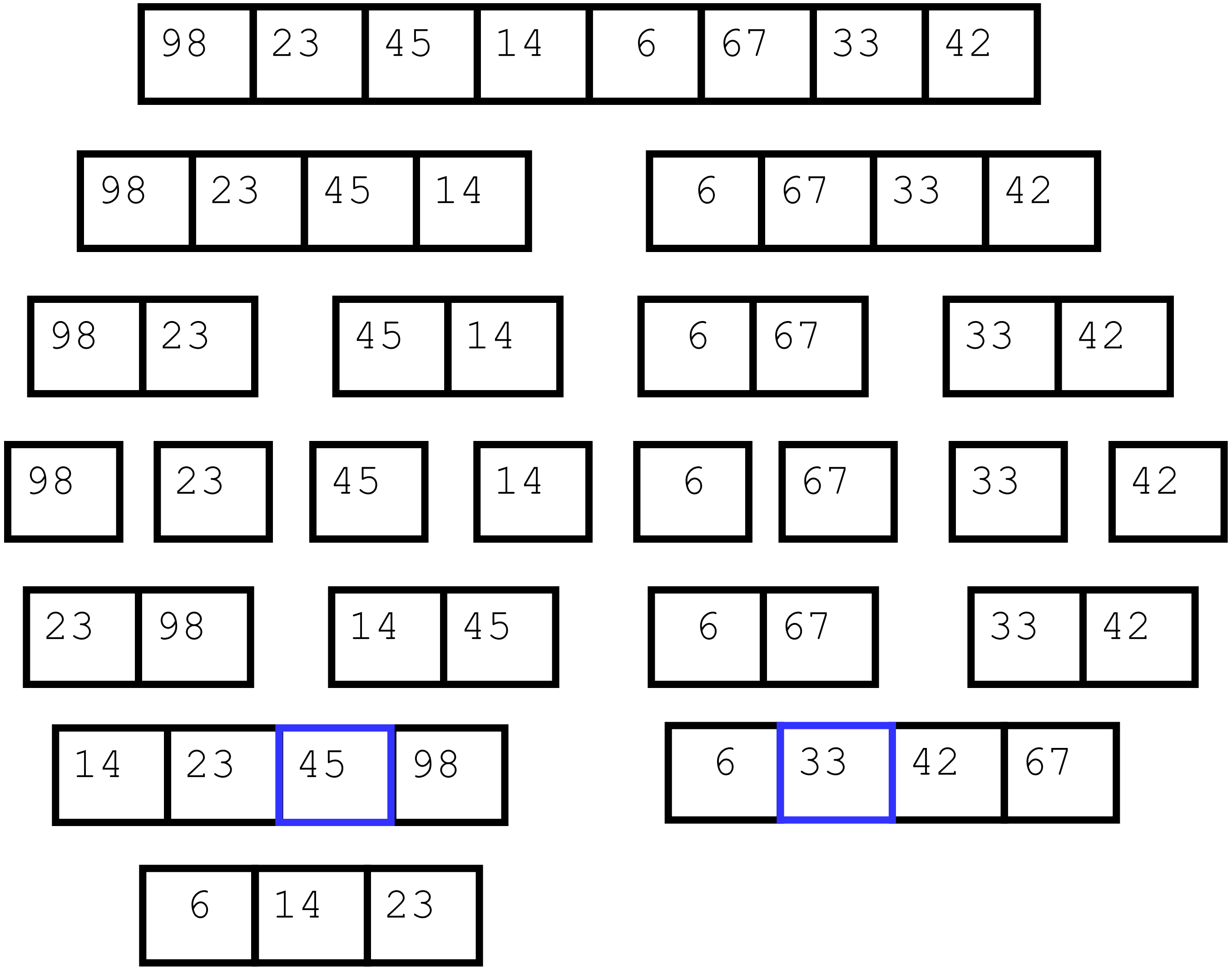
Merge


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

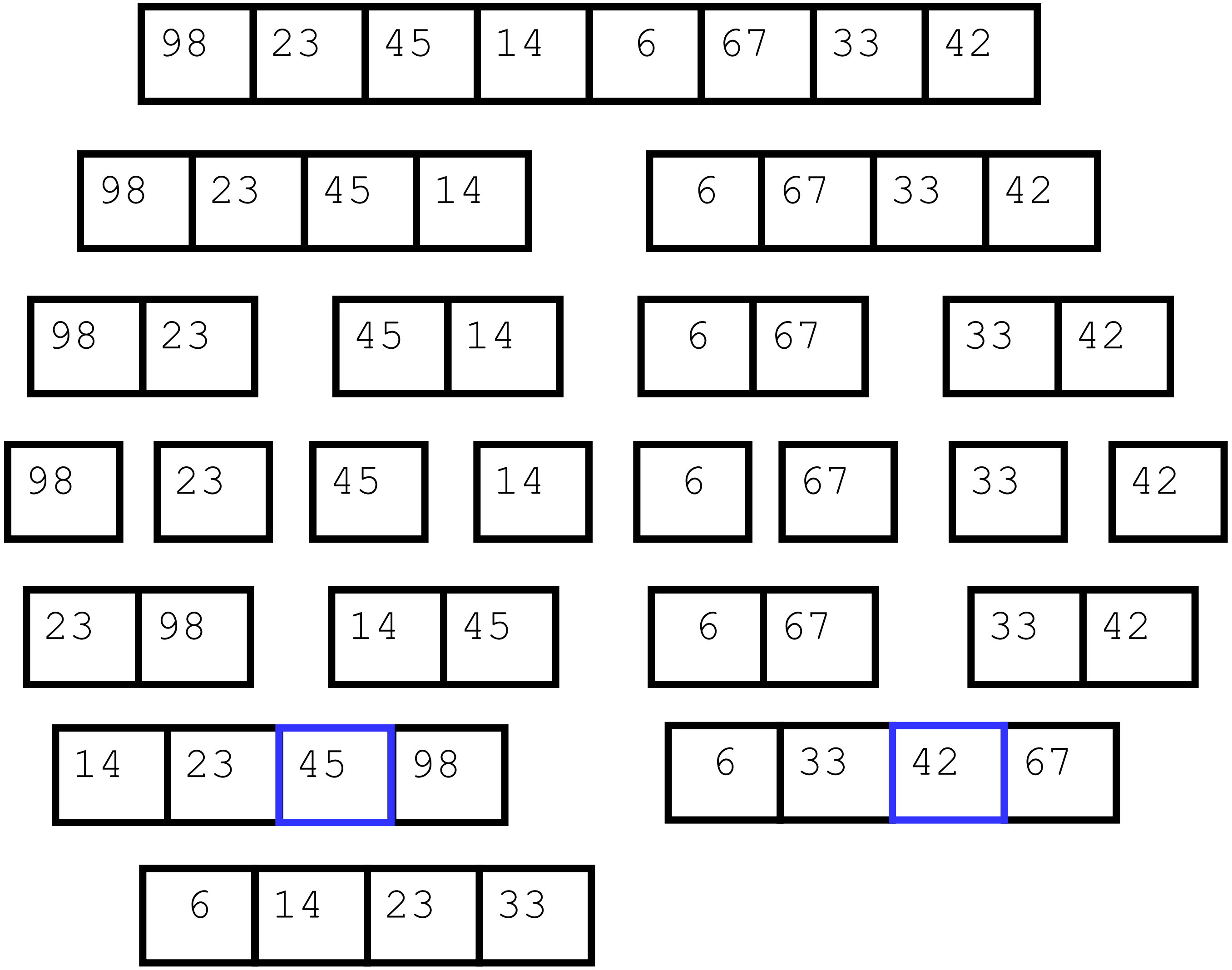
Merge


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

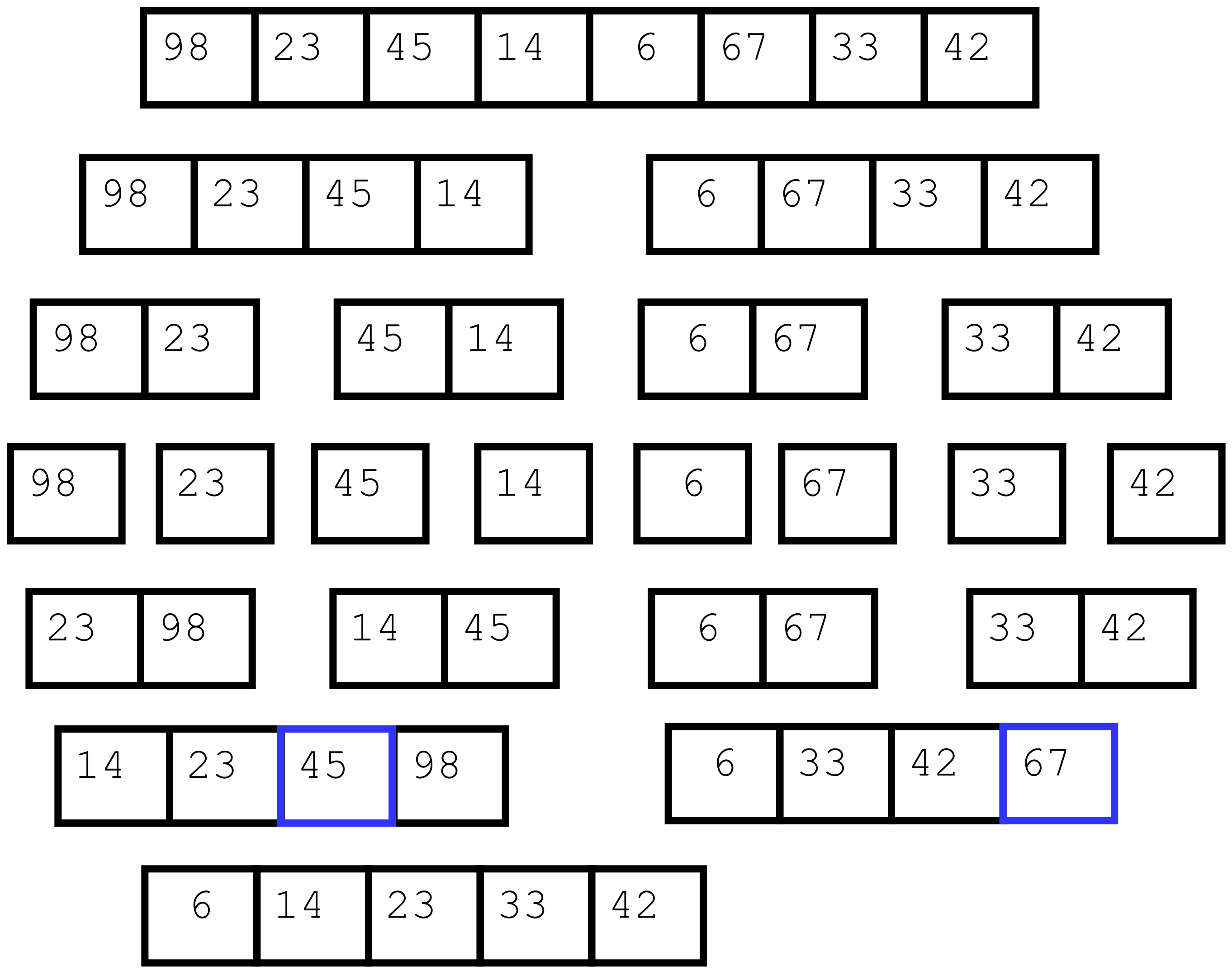
Merge


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

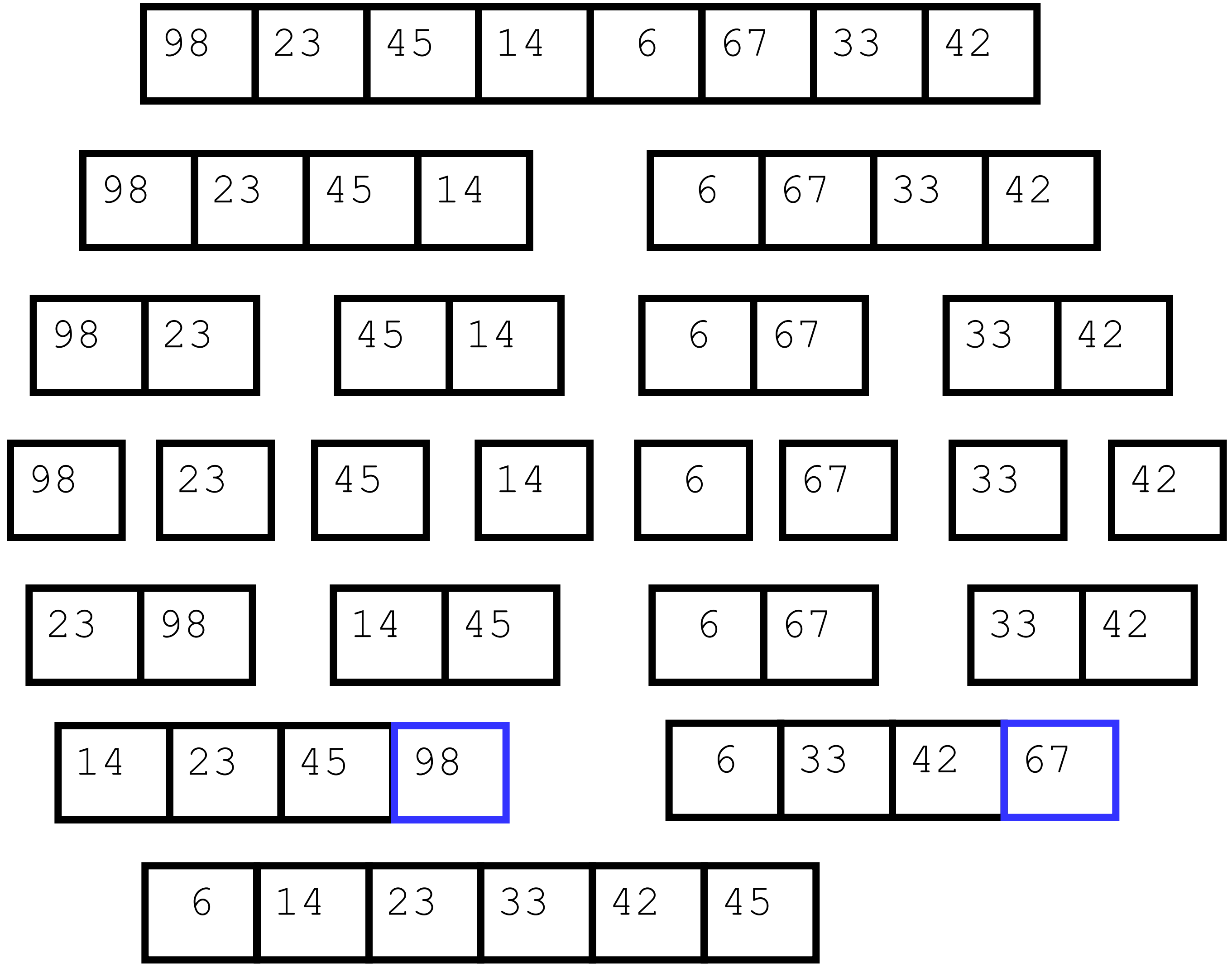
Merge


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

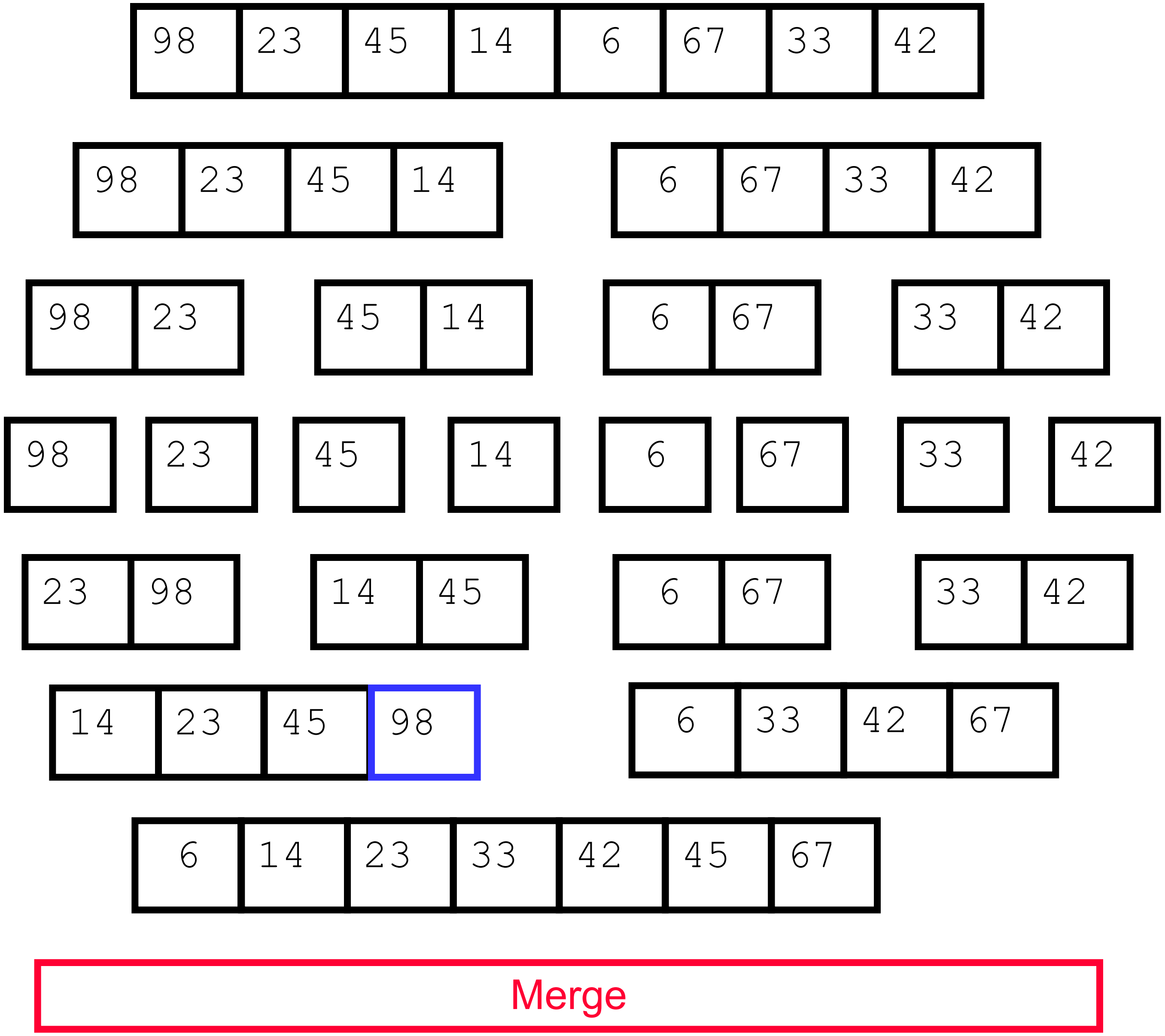
Merge

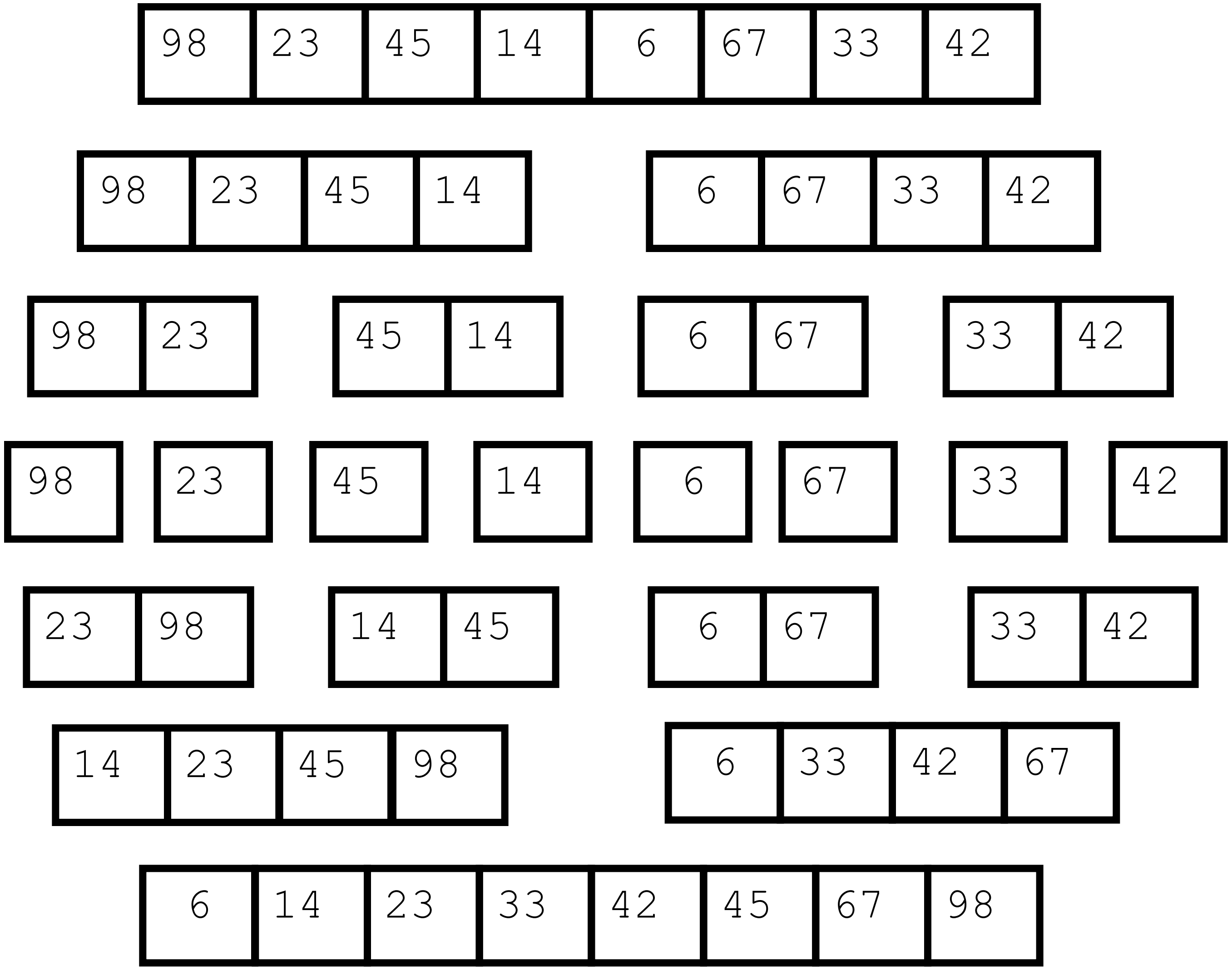

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42

Merge

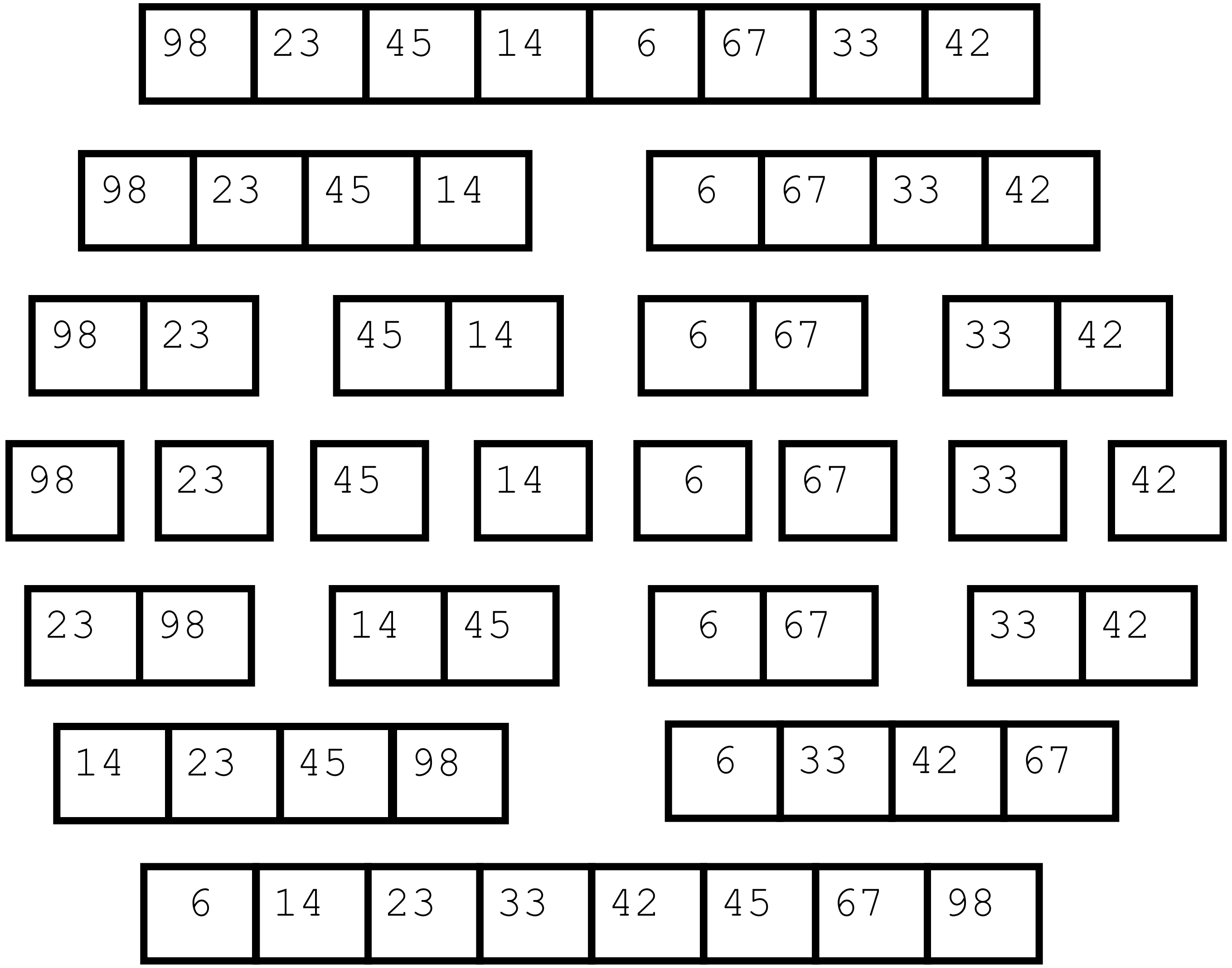

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45

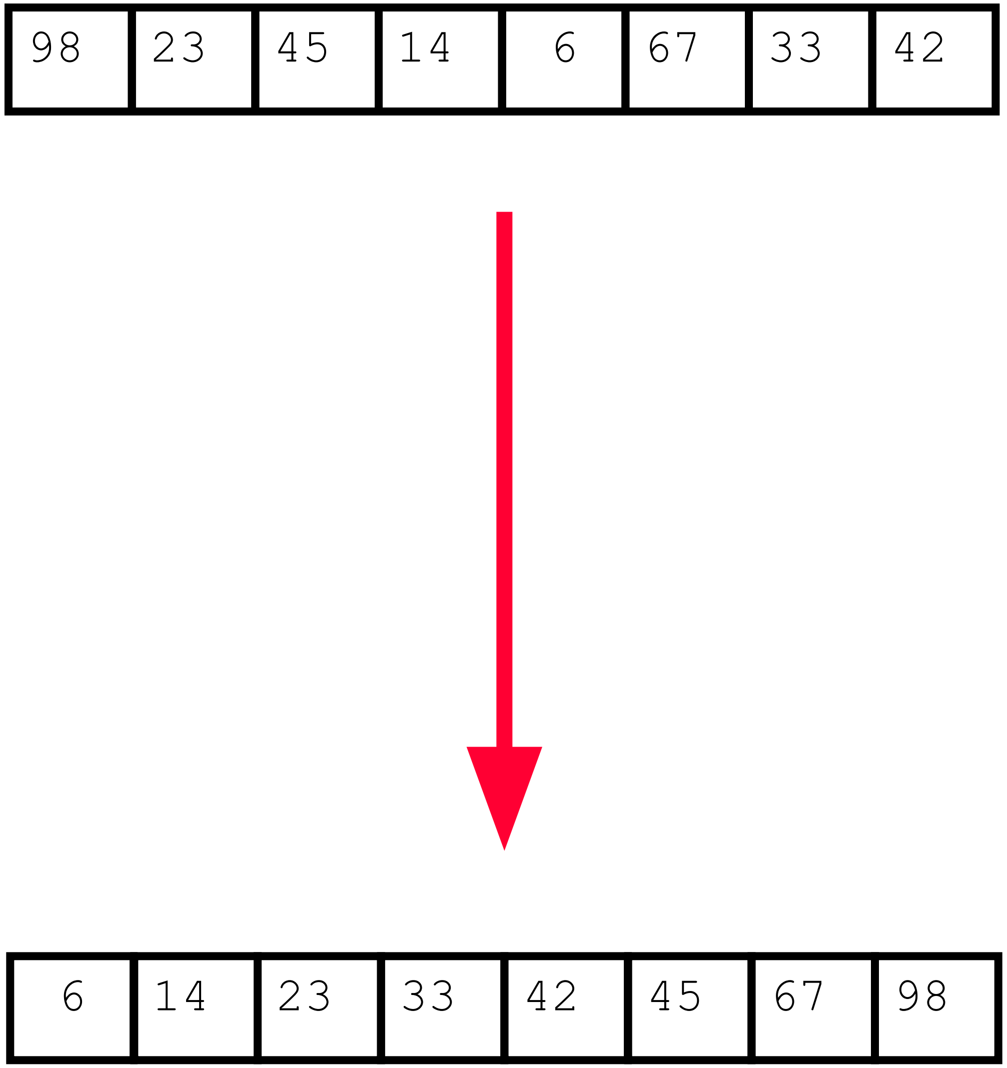
Merge


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

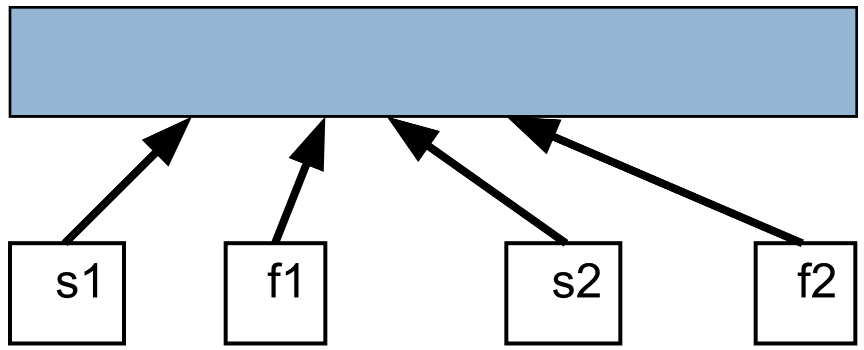
```
Mergesort(Passed an array)
  if array size > 1
    Divide array in half
    Call Mergesort on first half.
    Call Mergesort on second half.
    Merge two halves.

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]; }
```

Quicksort / partition-exchange sort

Quick sort is a divide and conquer algorithm.

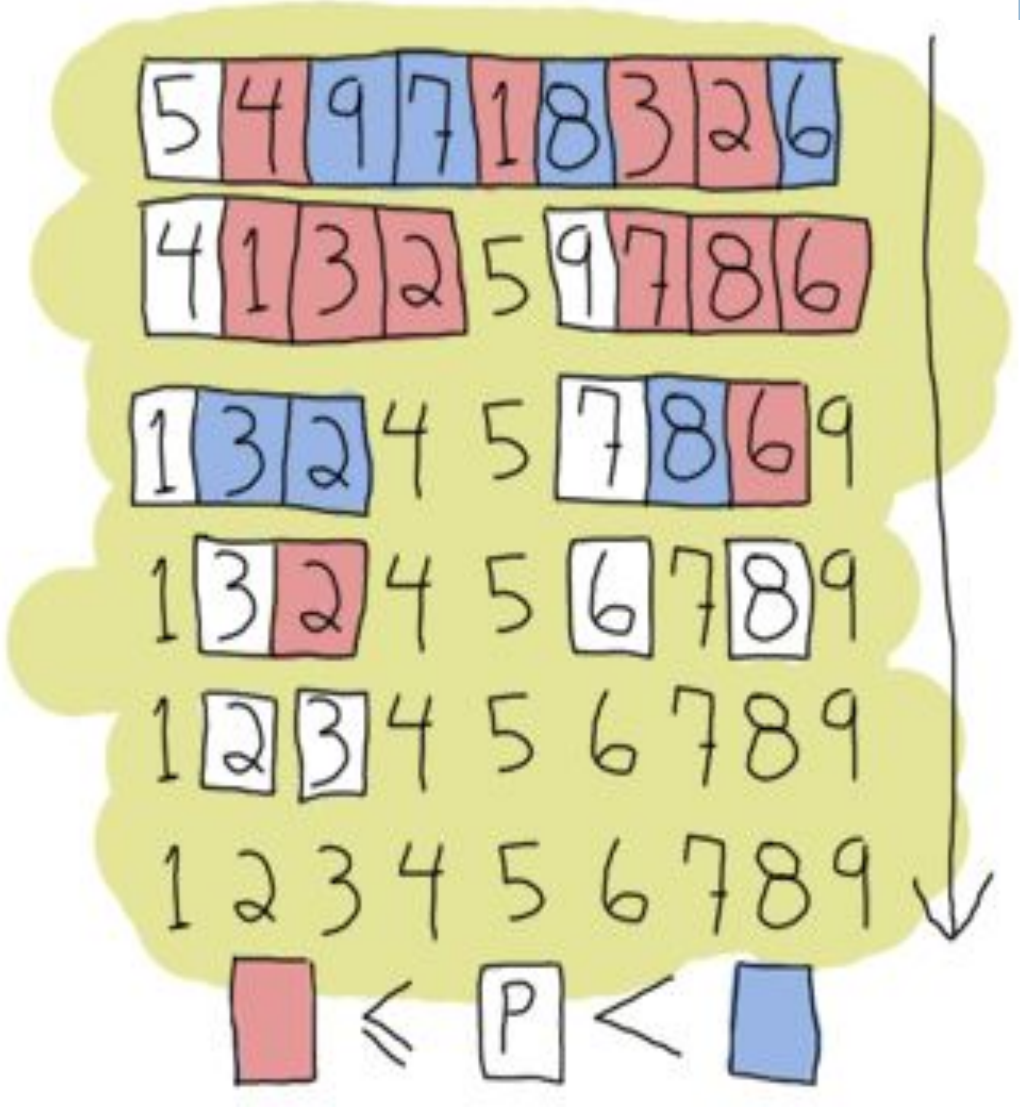
Quick sort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

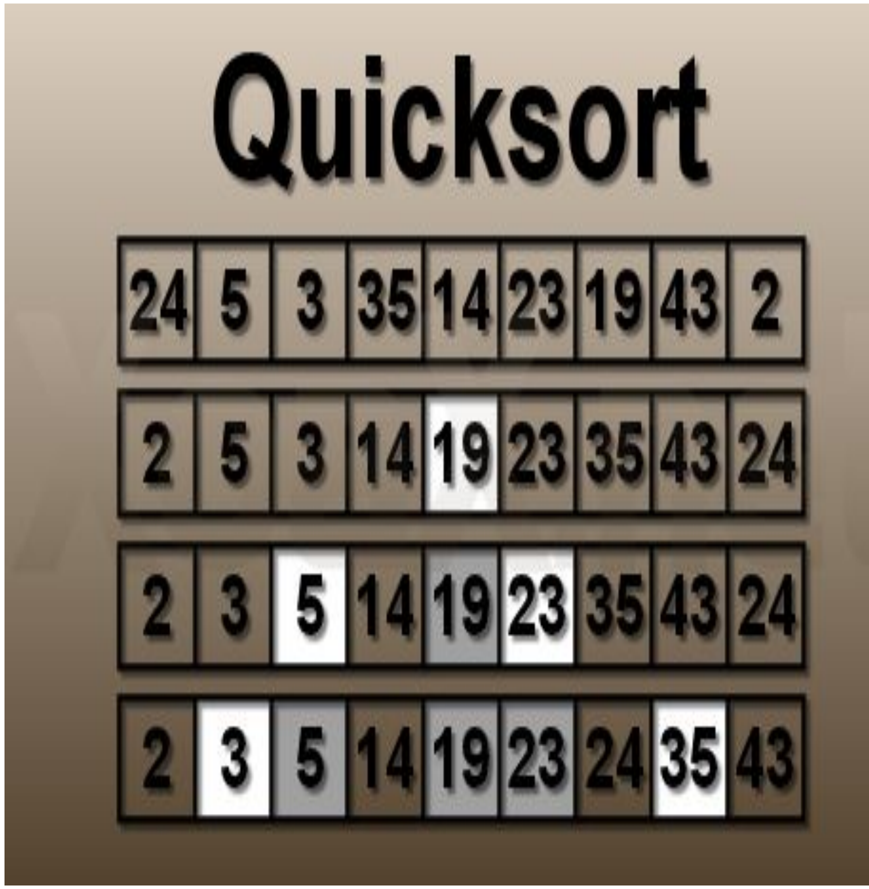
The steps are:

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

```
44 33 11 55 77 90 40 60 99 22 88
```

Let **44** be the **Pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. As **22** is smaller than **44** so swap them.

22 33 11 55 77 90 40 60 99 **44** 88

Now comparing **44** to the left side element and the element must be **greater** than 44 then swap them. As **55** are greater than **44** so swap them.

22 33 11 **44** 77 90 40 60 99 **55** 88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element **44** & one right from pivot element.

22 33 11 **40** 77 90 **44** 60 99 55 88

Swap with 77:

22 33 11 40 **44** 90 **77** 60 99 55 88

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;
}
```

```
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;
}
```

```
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