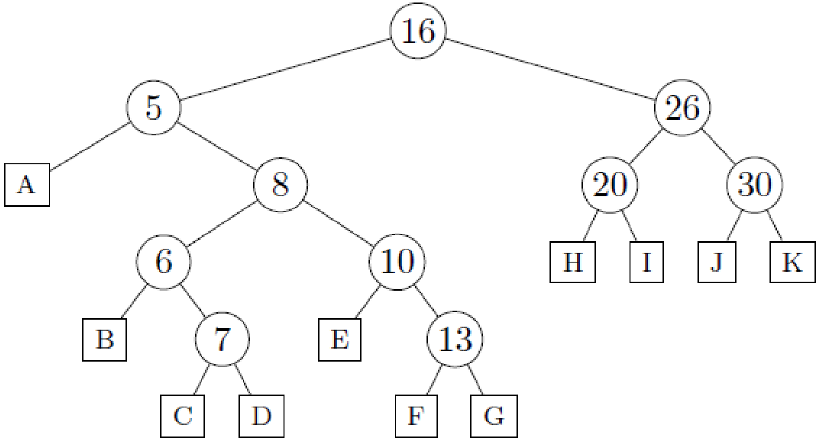
Sorting

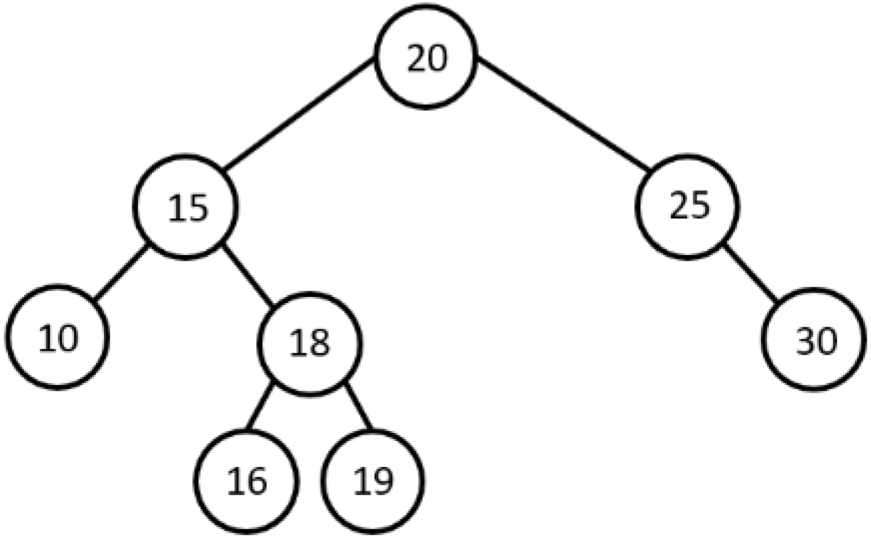
Insert Binary Tree



	A	B	C	D	E	F	G	H	I	J	K
INSERT(12)	A	B	C	D	E	F	G	H	I	J	K
INSERT(25)	A	B	C	D	E	F	G	H	I	J	K
INSERT(11)	A	B	C	D	E	F	G	H	I	J	K
INSERT(14)	A	B	C	D	E	F	G	H	I	J	K
INSERT(29)	A	B	C	D	E	F	G	H	I	J	K

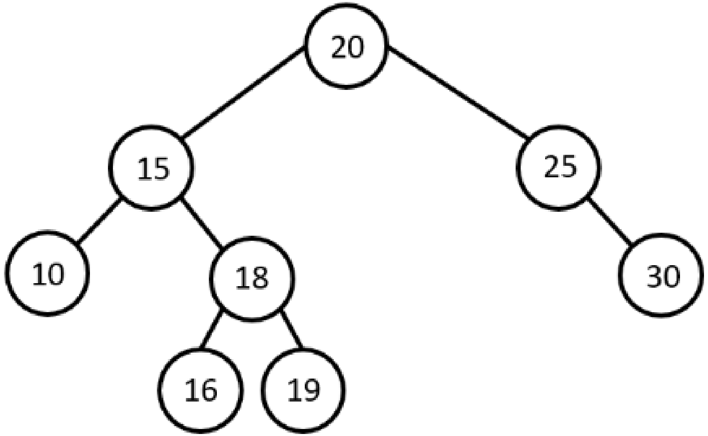
Avl Tree Delete

If you delete 30 from the following binary search tree using the algorithm that keeps the tree height-balanced by doing rotations, what tree do you get?



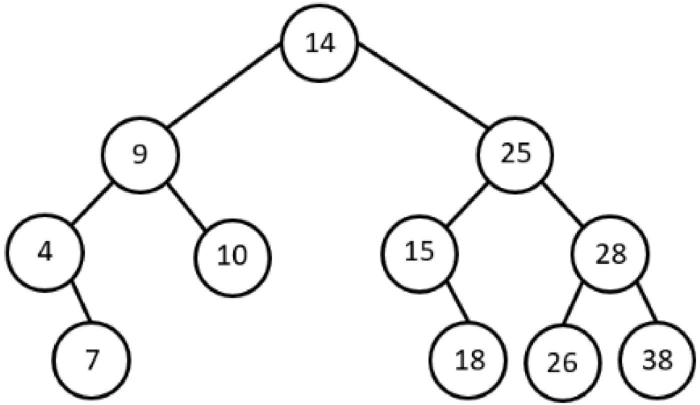
AVL Tree Insert

If you insert 17 into the following binary search tree using the algorithm that keeps the tree height-balanced by doing rotations, what tree do you get?



AVL Tree Delete Root

If you delete 14 from the following binary search tree using the algorithm that keeps the tree height-balanced by doing rotations, what tree do you get?



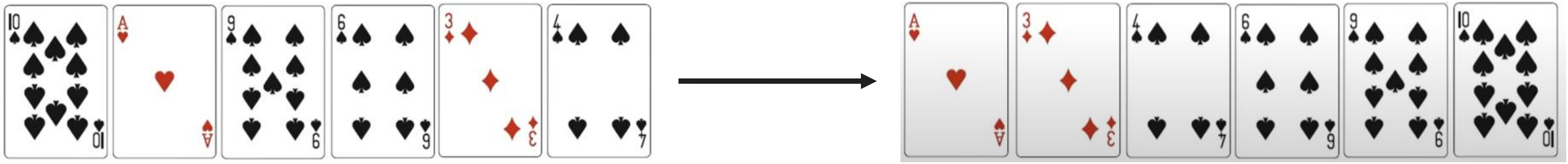
Binary Tree Traversal

Assume you insert 42, 34, 10, 65, 68, 39, 23, 50, 7 into an initially empty binary tree in the stated order. Identify the correct list which is obtained by subsequently performing each traversel.

	Preorder	Inorder	Postorder	Level order traversal (breadth-first)	Neither DFS nor BFS
42 34 65 10 39 50 68 23 7	○	○	○	○	○
7 23 10 39 34 42 50 68 65	○	○	○	○	○
7 10 23 34 39 42 50 65 68	○	○	○	○	○
42 34 10 7 23 39 65 50 68	○	○	○	○	○
7 23 10 39 34 50 68 65 42	○	○	○	○	○

Introduction to Sorting

- Sorting is arranging the elements in a list or collection in increasing or decreasing order of some property
    - Most common orders are in numerical or alphabetical order



Introduction to Sorting

Searching, assume one comparison takes 1 ms

Unsorted: Linear Search

Size = n → n comparisons

$n = 2^{64}$ → 2^{64} ms

Sorted: Binary Search

Size = n → log n comparisons

$n = 2^{64}$ → 64 ms

Insertion
Selection
Bubble
Shell
Merge
Heap
Quick
Quick3
Random
Nearly Sorted
Reversed
Few Unique

Sorting Algorithms: The Usual Suspects

- **Bubble sort**
- Selection sort
- **Insertion sort**
- **Merge sort**
- **Quick sort**
- Heap sort
- Shell sort
- Counting sort
- Radix sort

Classification of Sorting Algorithms

Parameters

1. **Time Complexity**
2. **Space Compexity** or Memory usage
   - ➢ In-place, constant memory, O(1)
   - ➢ Memory usage grows with input size, O(n)
3. **Stability**
   - ➢ Stable sorting maintains the relative order of elements with equal values.
4. **Adaptability**
   - ➢ Sorting algorithms whose running time improves the more pre-sorted the list is.
5. **In- vs out-of-place**
   - ➢ no extra space vs extra space needed
   - ➢ RAM or disk
6. **Recursive vs. non-recursive**
   - ➢ Bubble and Insertion → non-recursive
   - ➢ Merge and quick → recursive

Iterative sorting algorithms 1

Bubble sort

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, etc.
- 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

<u>Algorithm:</u>

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

0	1	2	3	4	5
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

0	1	2	3	4	5
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

0	1	2	3	4	5
42	77	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

0	1	2	3	4	5
42	77	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

0	1	2	3	4	5
42	35	77	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

0	1	2	3	4	5
42	35	77	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

0	1	2	3	4	5
42	35	12	77	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

0	1	2	3	4	5
42	35	12	77	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

0	1	2	3	4	5
42	35	12	77	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

0	1	2	3	4	5
42	35	12	77	5	101

  - End of first iteration/pass -> Largest value correctly placed
  - All other items are still out of order so now we repeat…

Bubbling all elements

0	1	2	3	4	5
42	35	12	77	5	101
0	1	2	3	4	5
35	12	42	5	77	101
0	1	2	3	4	5
12	35	5	42	77	101
0	1	2	3	4	5
12	5	35	42	77	101
0	1	2	3	4	5
5	12	35	42	77	101

N-1

Bubble sort in java

```java
void bubbleSort(int arr[])
{
    int n = arr.length;
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
            {
                // swap arr[j+1] and arr[j]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
}
```

Source: https://www.geeksforgeeks.org/bubble-sort/

What if the array was already sorted?

The algorithm would still need to go through all the iterations through the array! But that can be fixed by adding a few lines.

```
// An optimized version of Bubble Sort
static void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    boolean swapped;
    for (i = 0; i < n - 1; i++)
    {
        swapped = false;
        for (j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                // swap arr[j] and arr[j+1]
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }

        // IF no two elements were
        // swapped by inner loop, then break
        if (swapped == false)
            break;
    }
}
```

If an iteration through the array didn't cause *any* swaps, we are done!

Bubble Sort Summary

1. Time Complexity
   1. Worst case all elements in reverse order: $\mathbf{O(n^2)}$
   2. Average case $O(n^2)$
   3. Best case if largely pre-sorted: Since it is adaptive: O(n)
2. Space Complexity: **O(1)**
3. Bubble Sort is **stable**
4. Bubble Sort is **adaptive**
5. Bubble Sort is **in-Place**
6. Bubble Sort is **non-recursive**

[Implementation in Java]

Iterative sorting algorithms 2

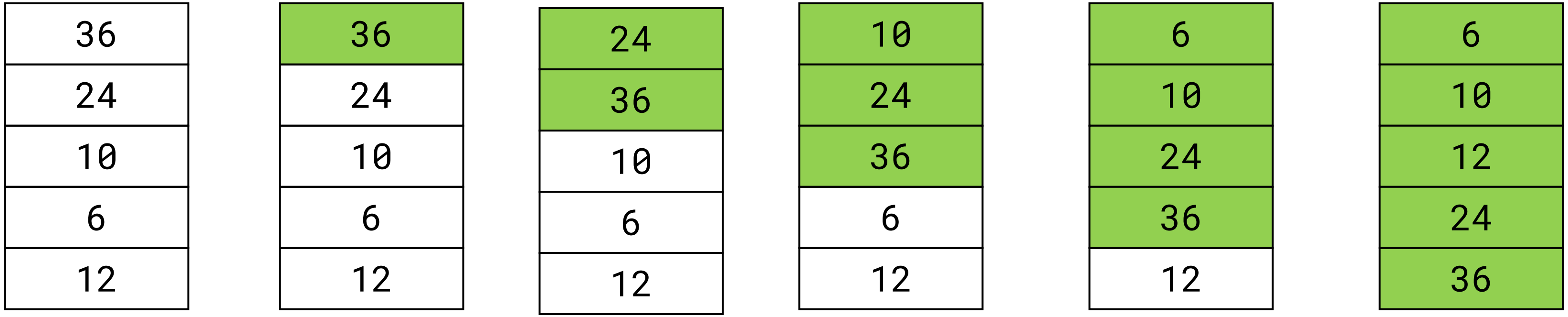
Insertion sort

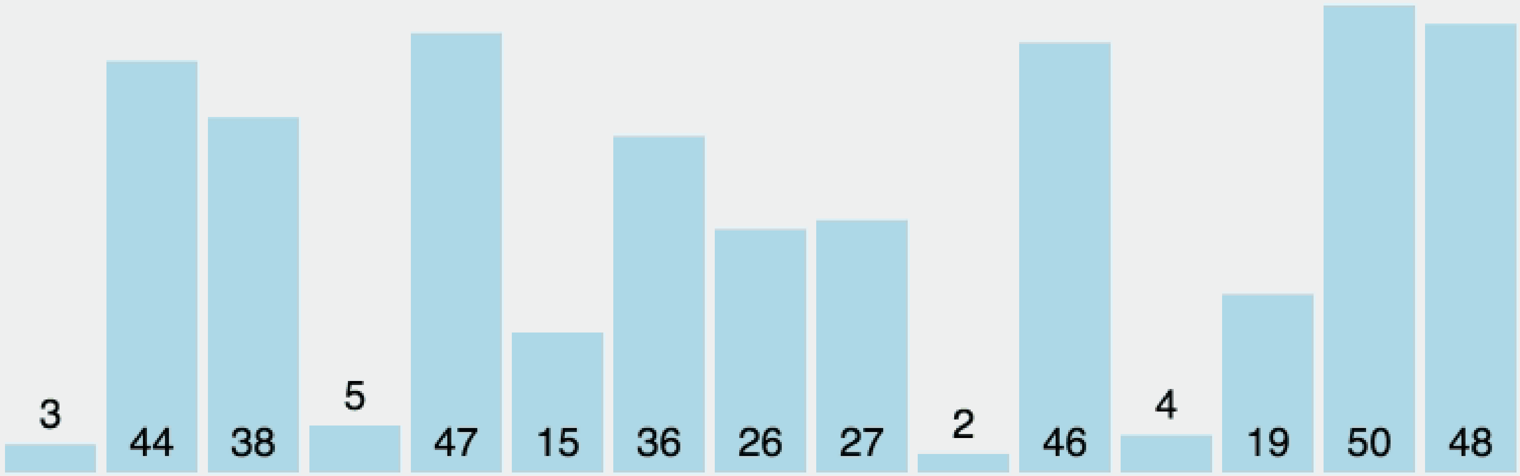
Insertion Sort

- Insertion sort works similar to the way you sort playing cards in your hands.
  - Divide the array in a sorted and an unsorted array.
  - Initially the sorted portion contains only one element: the first element in the array.
  - Take the second element and put it into its correct place
  - Take the third element and put it into its correct place
  - … and so on

Algorithm:

Step 1: If it is the first element, it is already sorted. return 1;
Step 2: Pick next element
Step 3: Compare with all elements in the sorted sub-list
Step 4: Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5: Insert the value
Step 6: Repeat until list is sorted




3
44
38
5
47
15
36
26
27
2
46
4
19
50
48

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 |

```
void insertionSort(int A[], int n) {

    for (int i = 1 to n-1; i = i+1) {
        int key = A[i]
        int j = i - 1

        // Shift elements of A[0..i-1], that are //
        // greater than key, to one position ahead of
           their current position //

        while (j >= 0 && A[j] > key) {

            A[j + 1] = A[j]
            j = j - 1
        }
        A[j + 1] = key
    }
}
```

Complexity Analysis

Worst case scenario: If our input is reversely sorted, then the insertion sort algorithm performs the maximum number of operations

```
for i = 1, 1 comparison and 1 shift operation
for i = 2, 2 comparison and 2 shift operation
and so on.
Total number of comparison operation = 1 + 2 + 3 +......n-1 = n(n-1)/2
Total number of shifting operation = 1 + 2 + 3 +......n-1 = n(n-1)/2
```

Best case scenario: If our input is already sorted, then the insertion sort algorithm perform the minimum number of operations

```
for i = 1, only 1 comparison
for i = 2, only 1 comparison
and so on...
Total number of comparison operation = n-1
Total number of shifting operation = 0
```

Insertion Sort Summary

1. Time Complexity
   1. Worst case all elements in reverse order: $\mathbf{O(n^2)}$
   2. Average case $O(n^2)$
   3. Best case, sorted array as input: O(n)
2. Space Complexity: **O(1)**
3. Insertion Sort is **stable**
4. Insertion Sort is **adaptive**
5. Insertion Sort is **in-Place**
6. Insertion Sort is **non-recursive**

[Implementation in Java]()

Recursive sorting algorithms

Divide and Conquer Revisited

  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- Idea 1:
  - Divide array into two halves, recursively sort left and right halves, then merge two halves → Mergesort
- Idea 2:
  - Partition array into items that are “small” and items that are “large”, then recursively sort the two sets → Quicksort

Recursive sorting algorithms 1

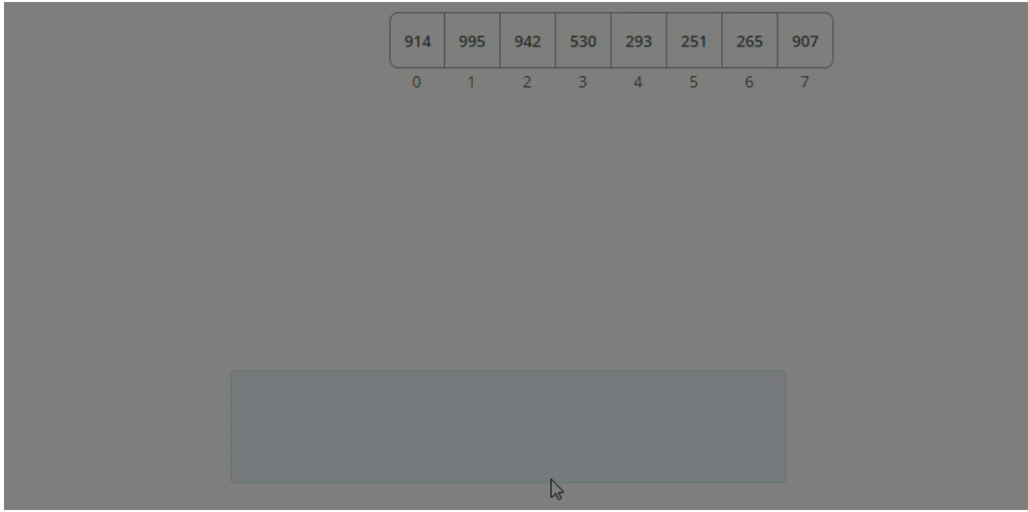
Merge sort

Merge Sort (recursive version)

Algorithm:

```
mergeSort(arr[], l,  r)
If r > l

Step 1:   Find the middle point to divide
          the array into two halves:
                middle m = l+ (r-l)/2
Step 2:   Call mergeSort for first half:
                mergeSort(arr, l, m)
Step 3:   Call mergeSort for second half:
                mergeSort(arr, m+1, r)
Step 4:   Merge the two halves sorted in
          step 2 and 3:
                merge(arr, l, m, r)
```



Notice that the algorithm has two main functions: **divide** and **merge**
If the list is empty or has one item, it is sorted by definition (the base case)

The sort function

```
// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r) {
        // Find the middle point
        int m =l+ (r-l)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

Source: https://www.geeksforgeeks.org/merge-sort/?ref=lbp

The merge function

```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    // Find sizes of two subarrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[] = new int[n1];
    int R[] = new int[n2];

    /*Copy data to temp arrays*/
    for (int i = 0; i < n1; ++i)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays */

    // Initial indexes of first and second subarrays
    int i = 0, j = 0;

    // Initial index of merged subarray array
    int k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy remaining elements of L[] if any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy remaining elements of R[] if any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
```

Complexity of merge?

The sort function

```
// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r) {
        // Find the middle point
        int m =l+ (r-l)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

Complexity of mergeSort?

Master Theorem Analysis: $T(n) = aT\left(\frac{n}{b}\right) + O(n^k)$
We divide arr into two parts so a = 2
Each sub-problem has size n/2 so b = 2
Merge takes O(n) so k = 1

$$a = b^k$$

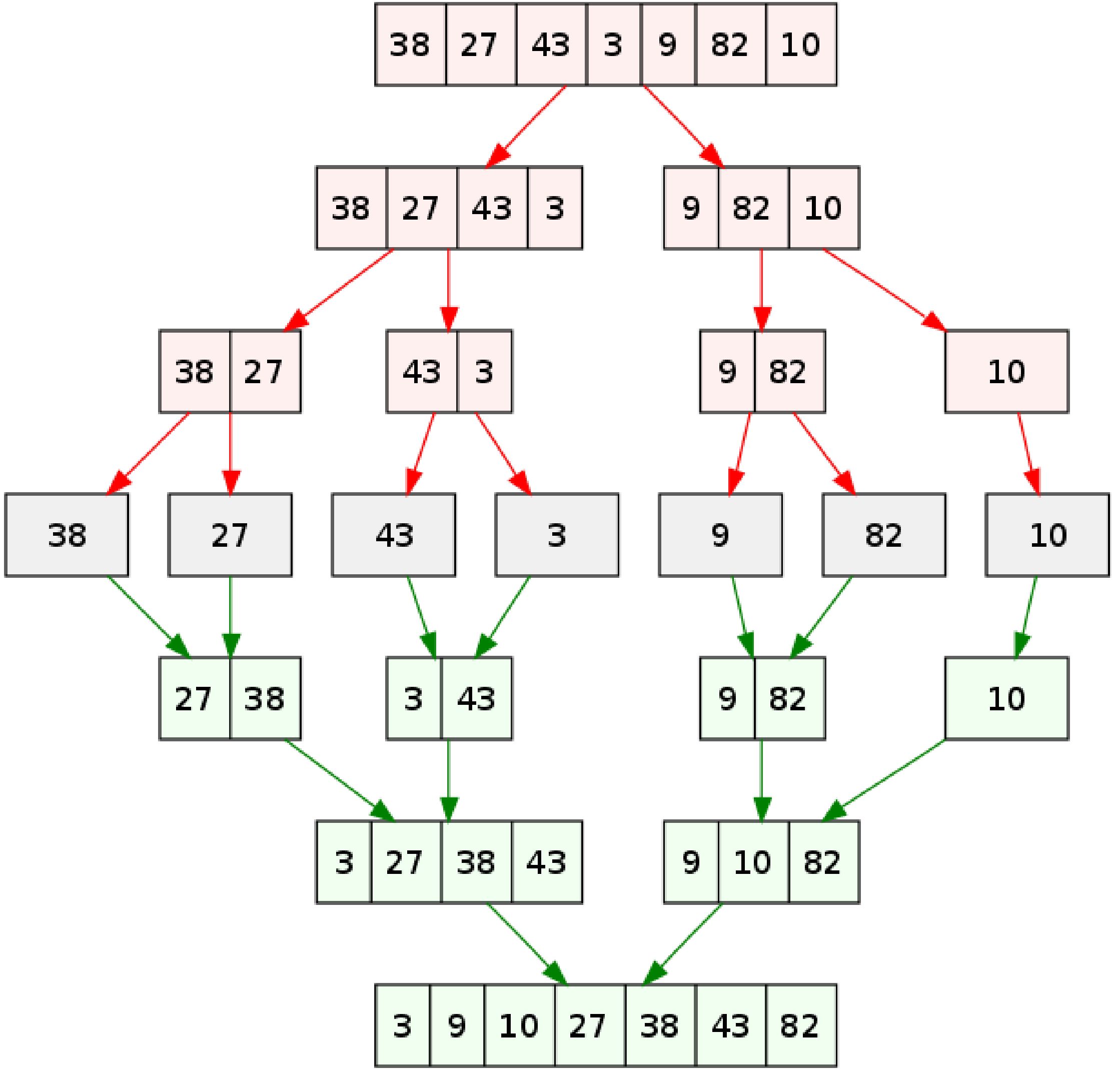
$$T(N) = \begin{cases} \Theta(N^{\log_b(a)}) & \text{if } a > b^k \\ \Theta(N^k \cdot \log N) & \text{if } a = b^k \\ \Theta(N^k) & \text{if } a < b^k \end{cases}$$

$$T(n) = 2T(n/2) + \theta(n)$$

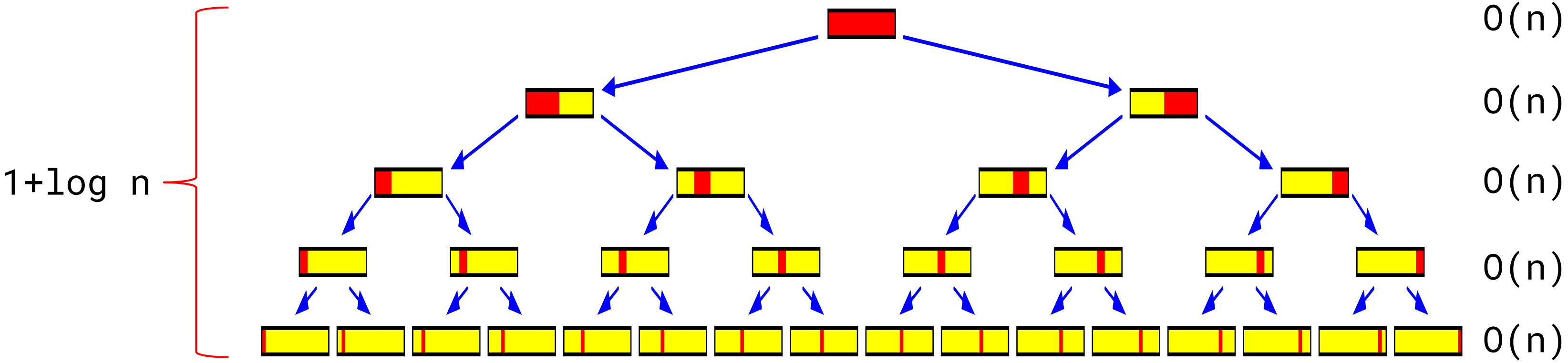
Source: https://www.geeksforgeeks.org/merge-sort/?ref=lbp

MergeSort

From Wiki



Merge Sort Analysis



We are doing O(n) work per level. We have log n levels so we get O(n log n)

<u>Master Theorem Analysis:</u> $T(n) = aT\left(\frac{n}{b}\right) + O\left(n^k\right)$

We divide arr into two parts so a = 2
Each sub-problem has size n/2 so b = 2
Merge takes O(n) so k = 1

$a = b^k$

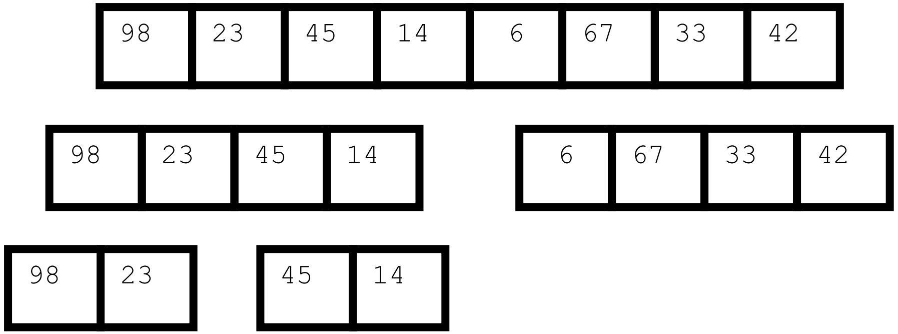
$$T(N) = \begin{cases} \Theta\left(N^{\log_b(a)}\right) & \text{if } a > b^k \\ \Theta\left(N^k \cdot \log N\right) & \text{if } a = b^k \\ \Theta\left(N^k\right) & \text{if } a < b^k \end{cases}$$

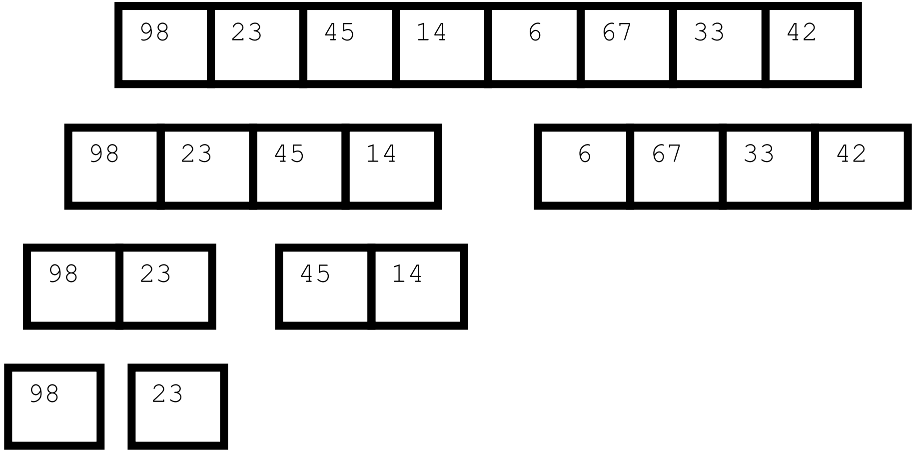
98	23	45	14	6	67	33	42

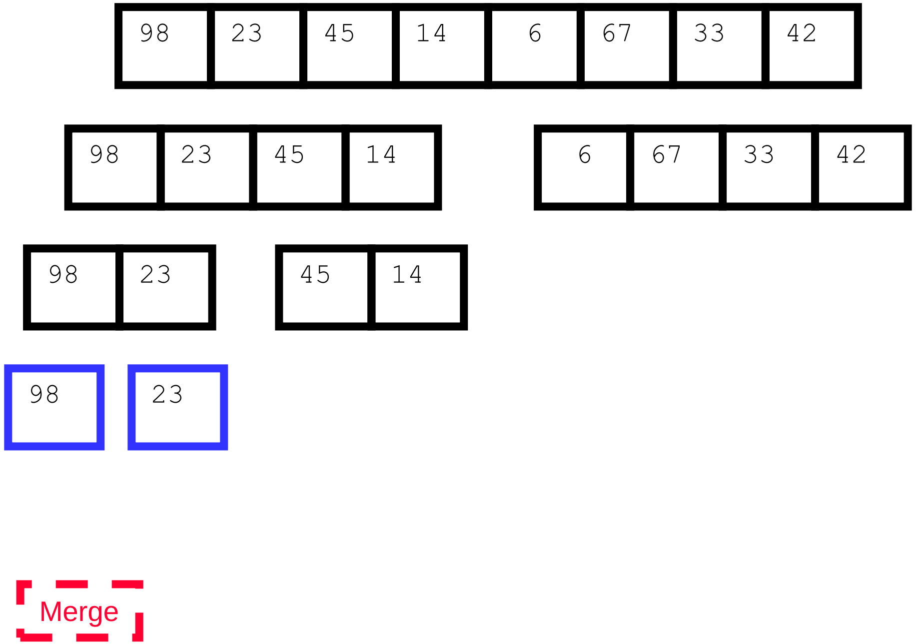
98	23	45	14	6	67	33	42

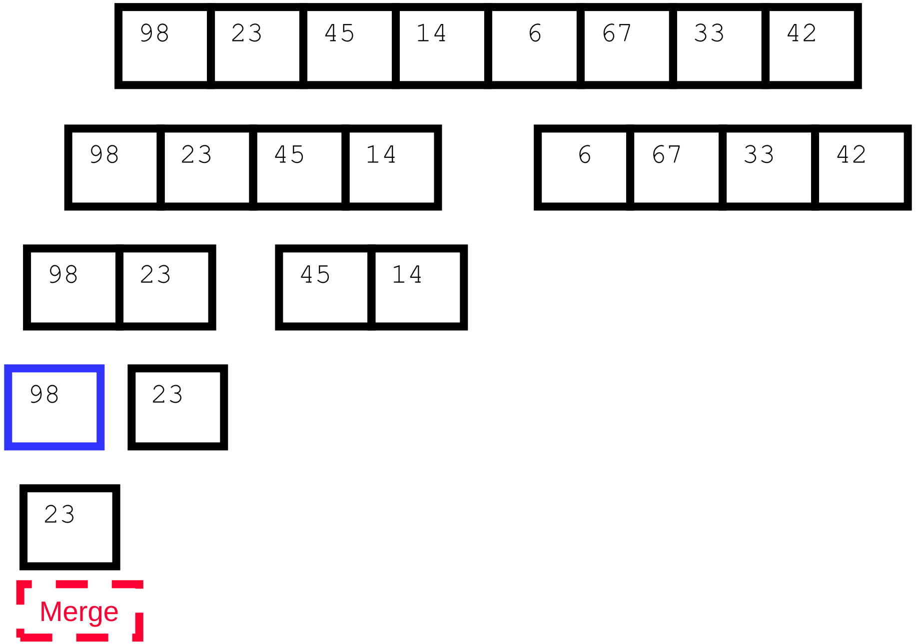
98	23	45	14

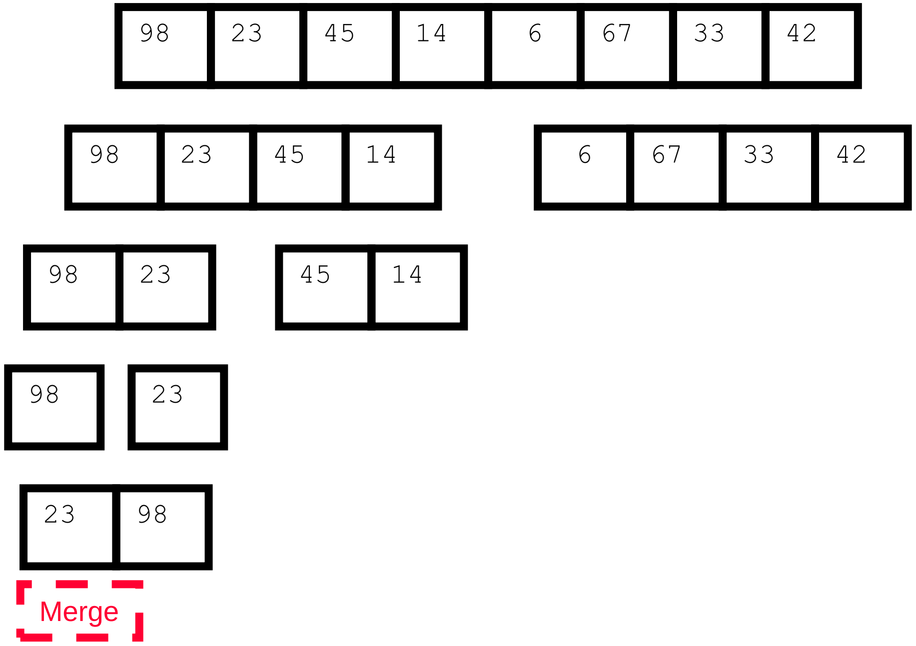
6	67	33	42

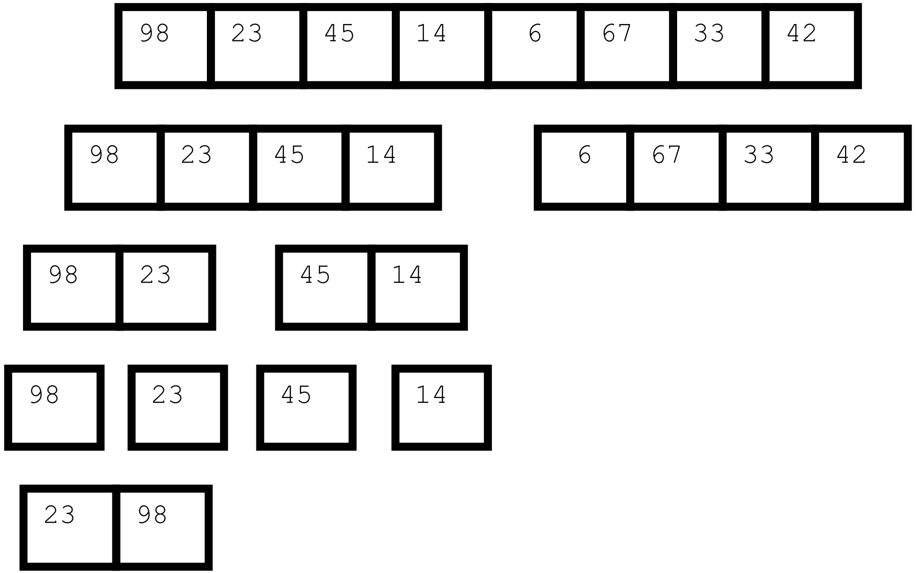

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

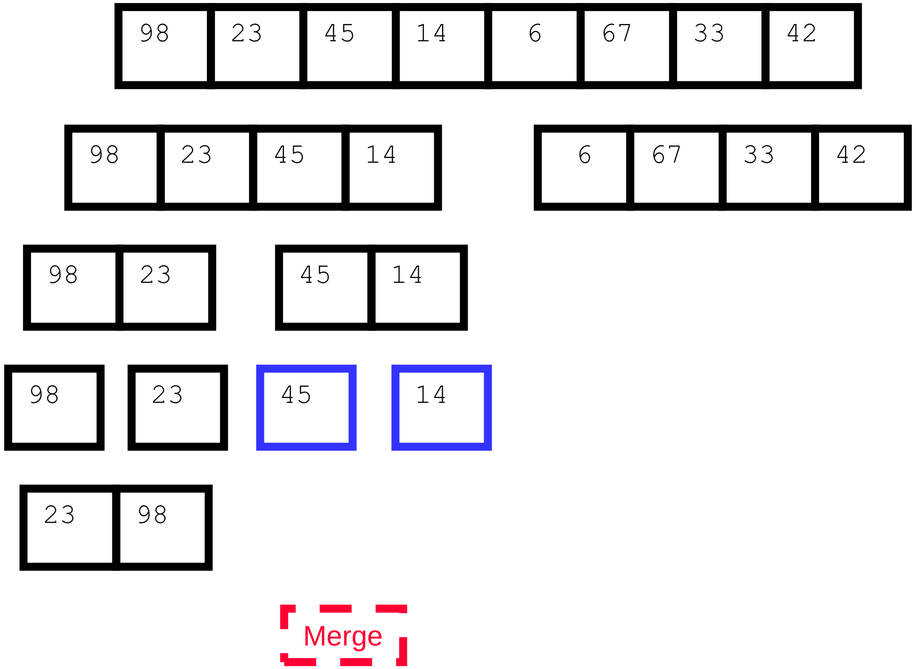

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

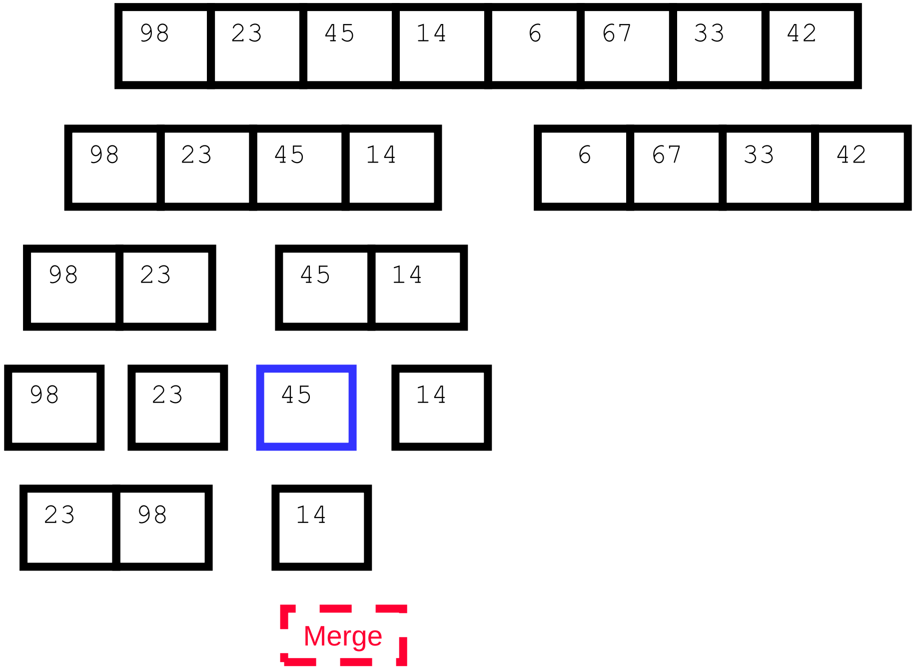

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

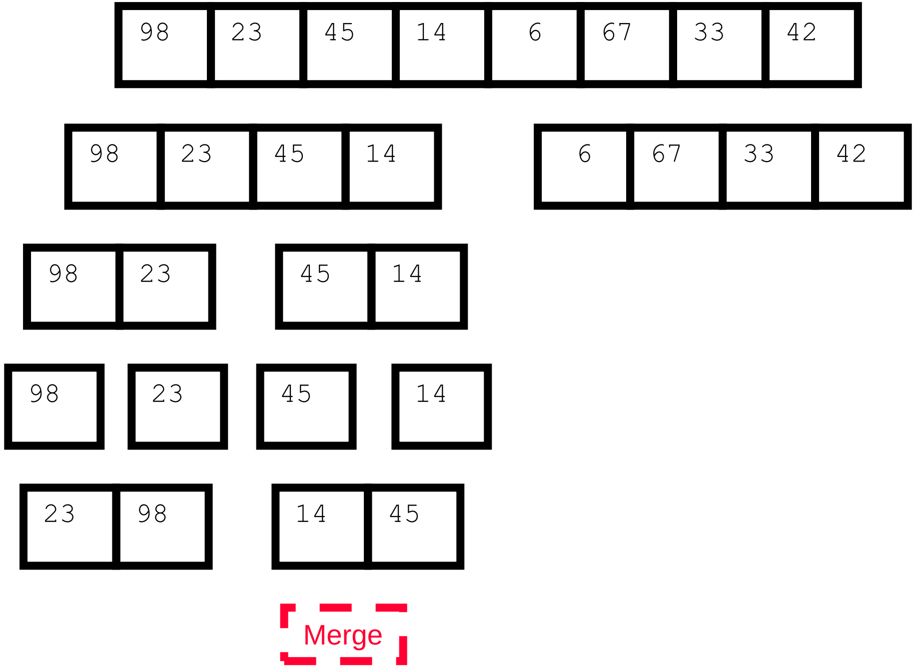

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

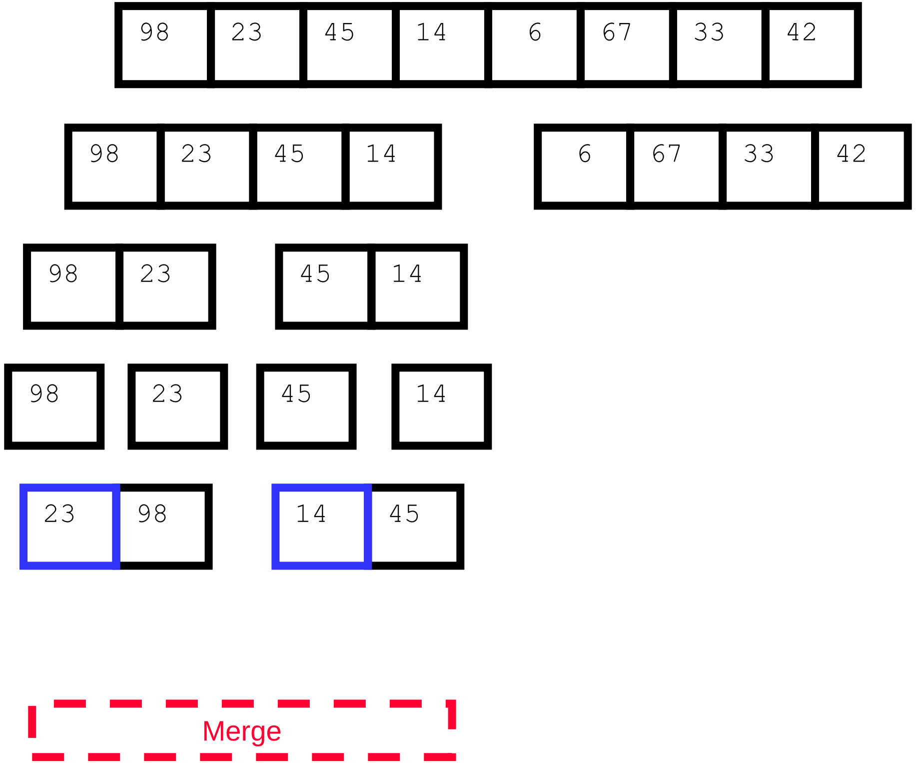

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge


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

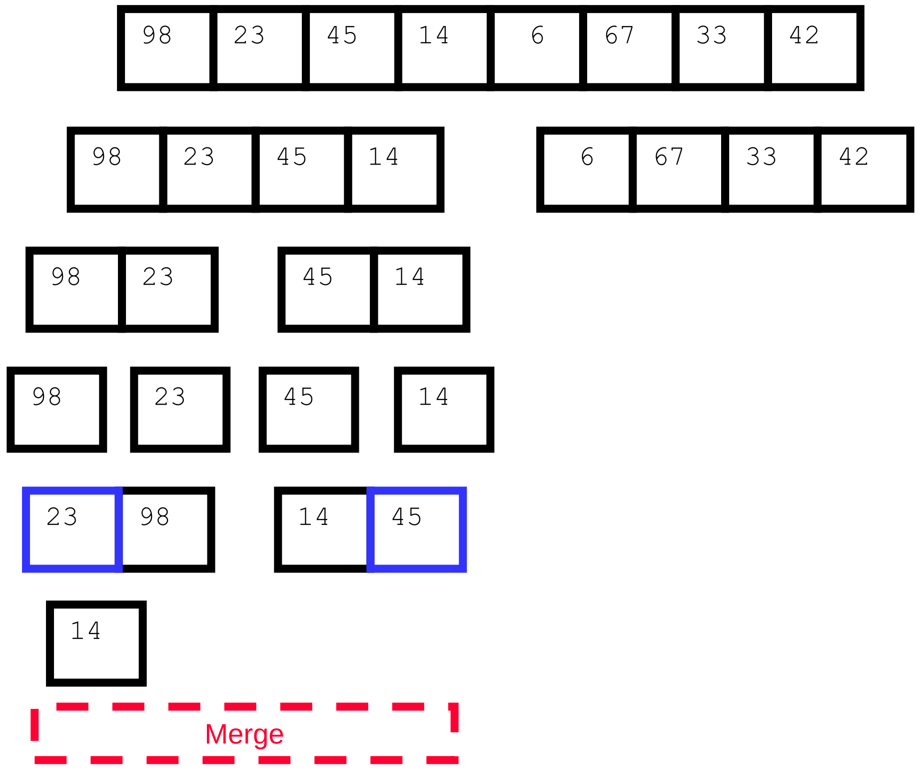

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
Merge

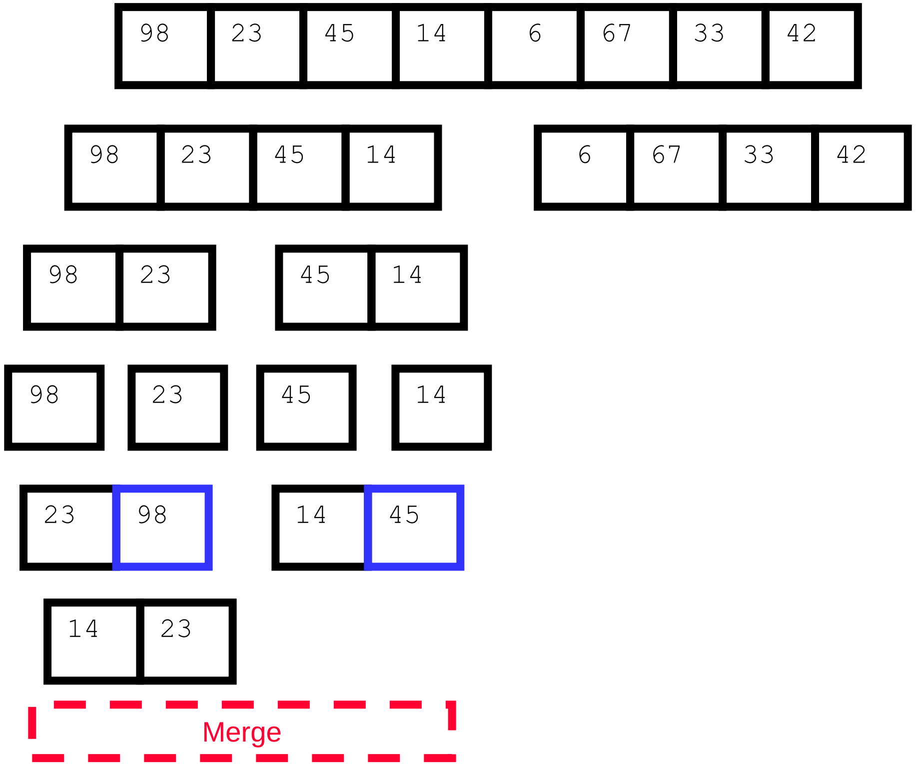

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14
Merge


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

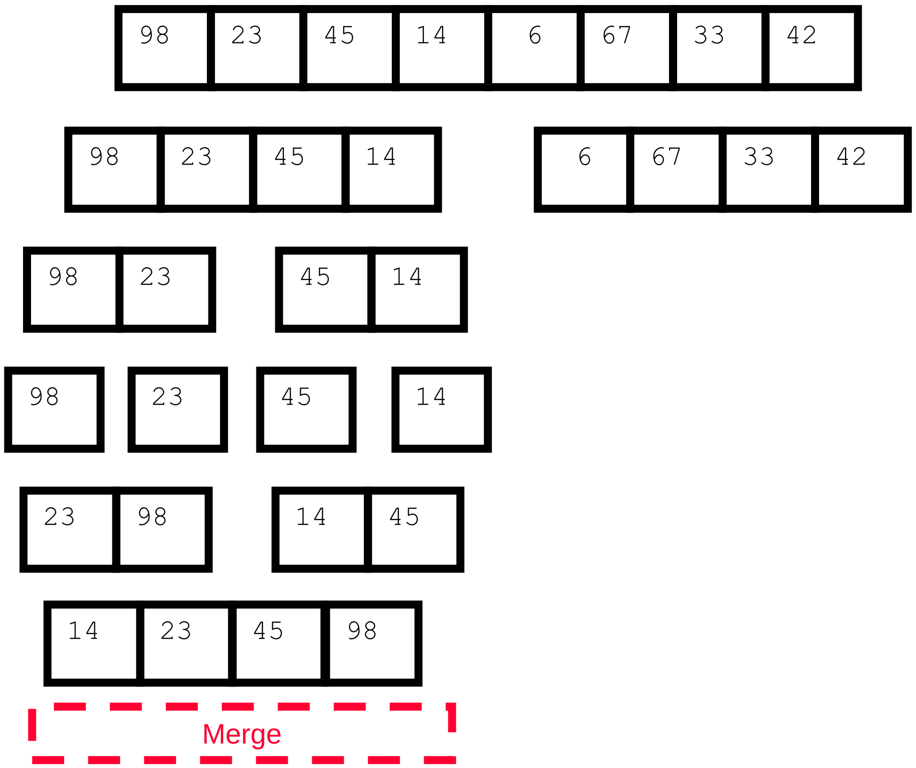


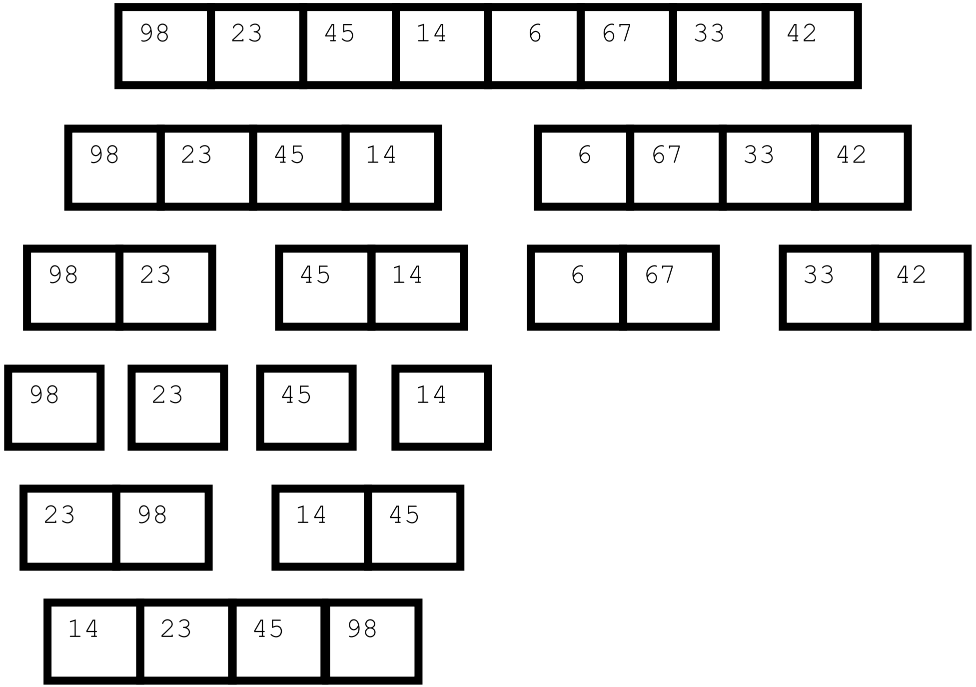
98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

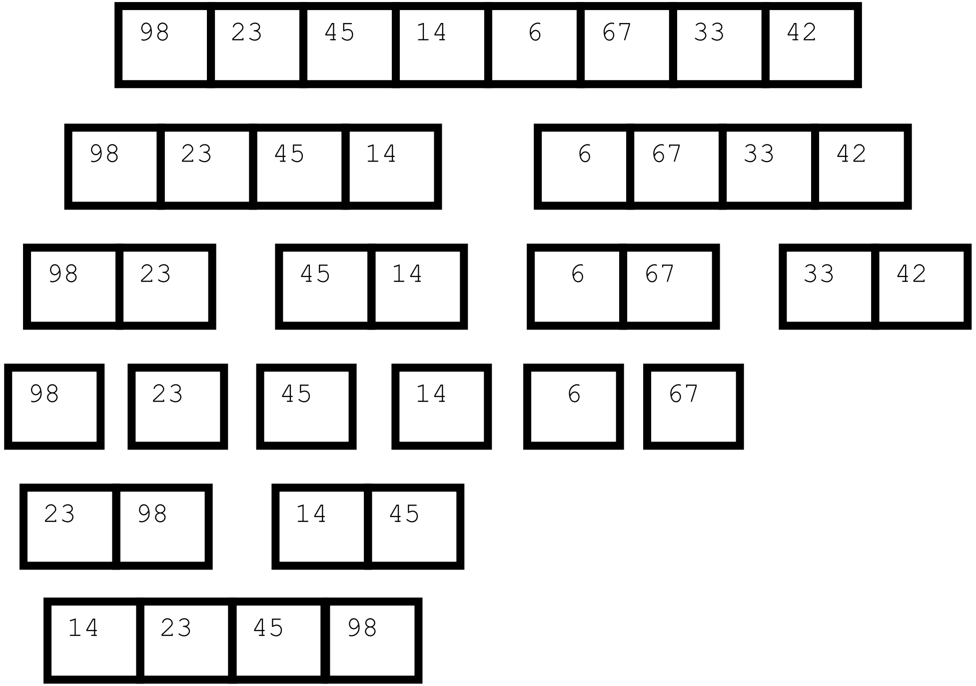

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

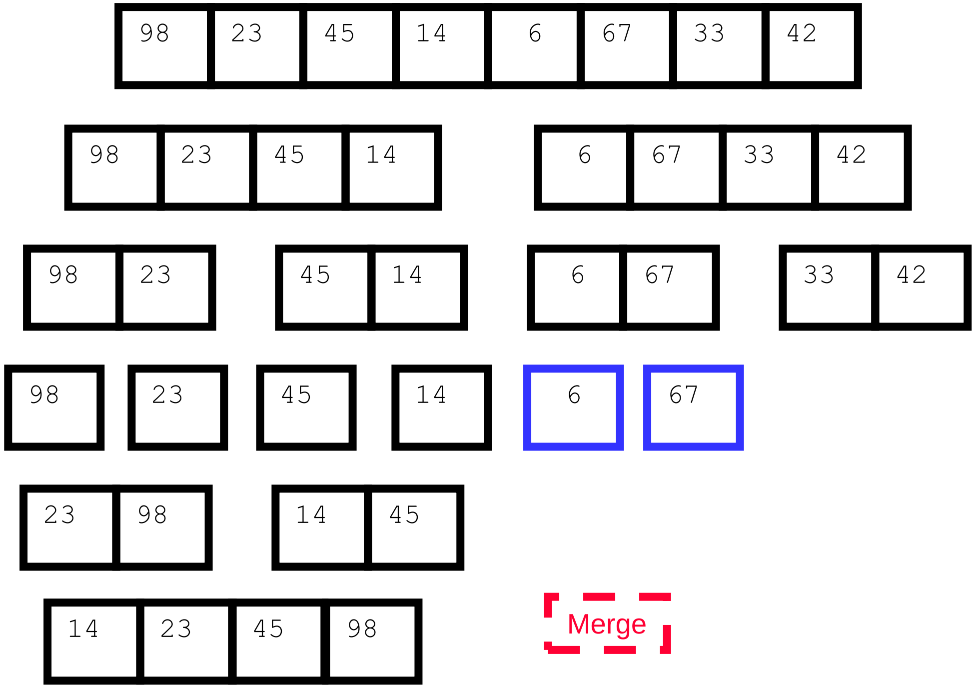

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

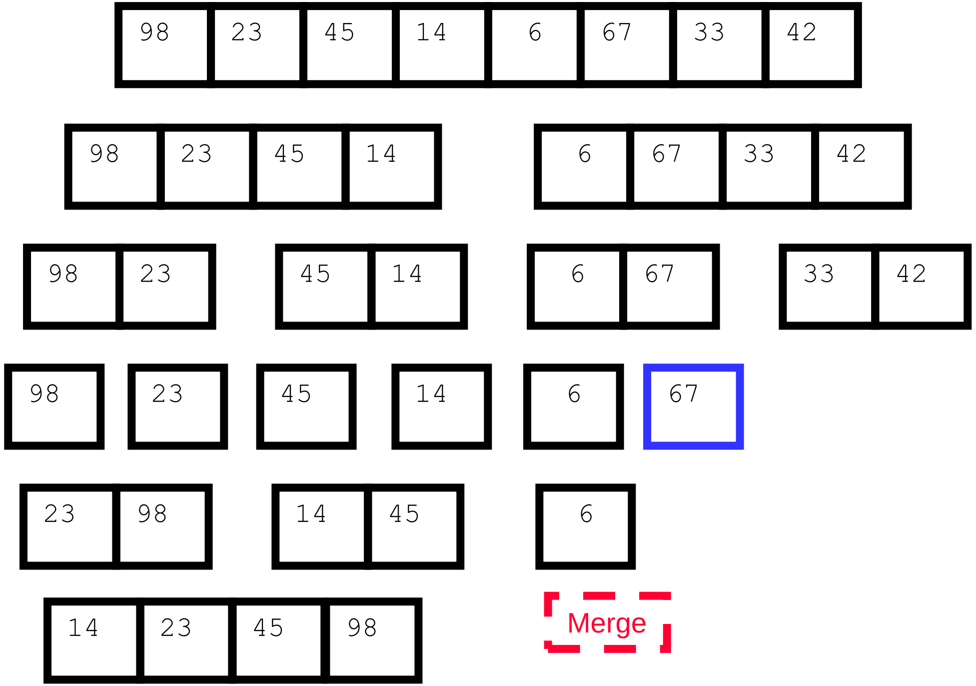

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

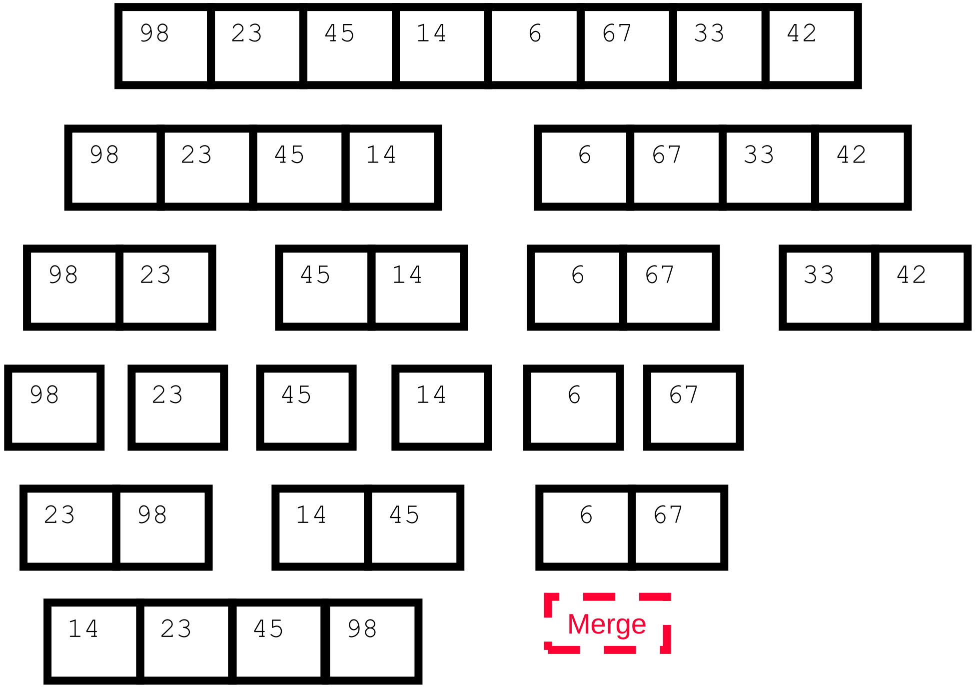

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

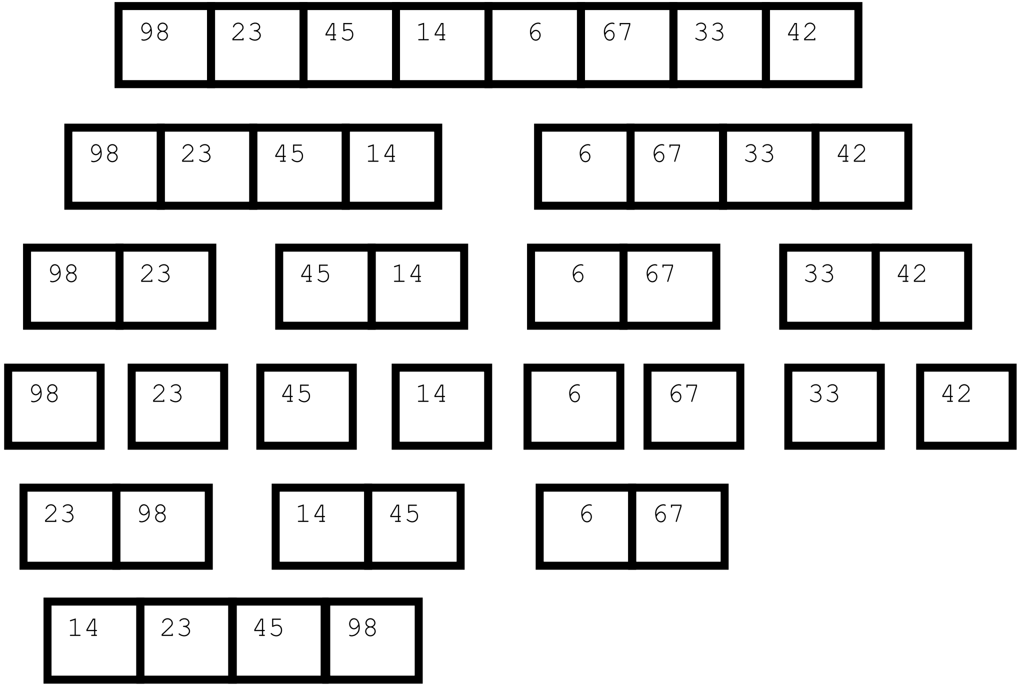

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

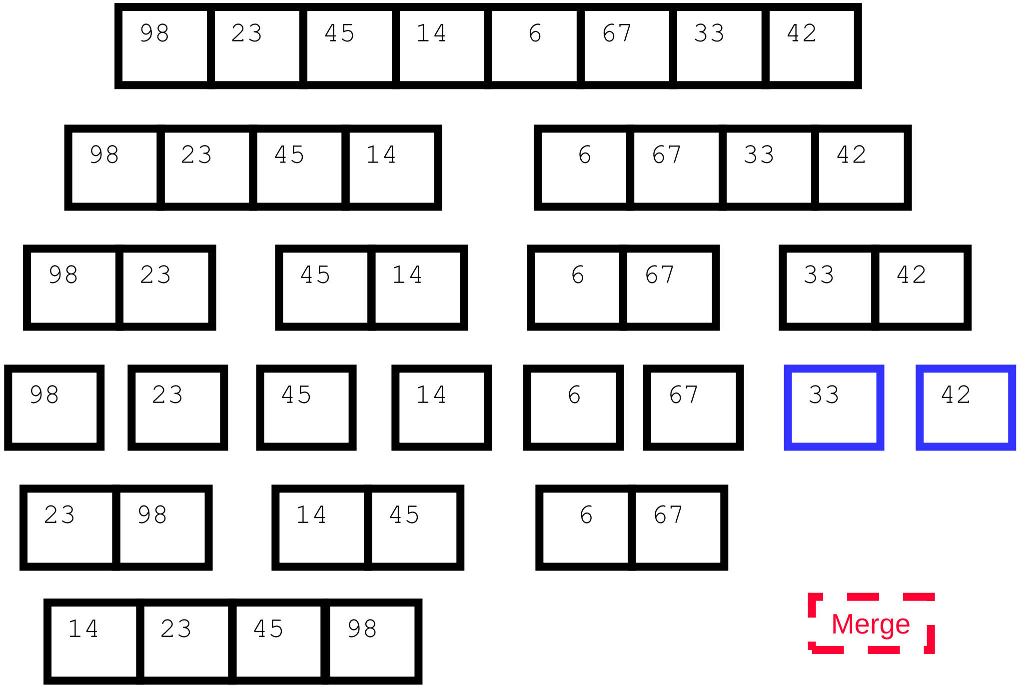

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

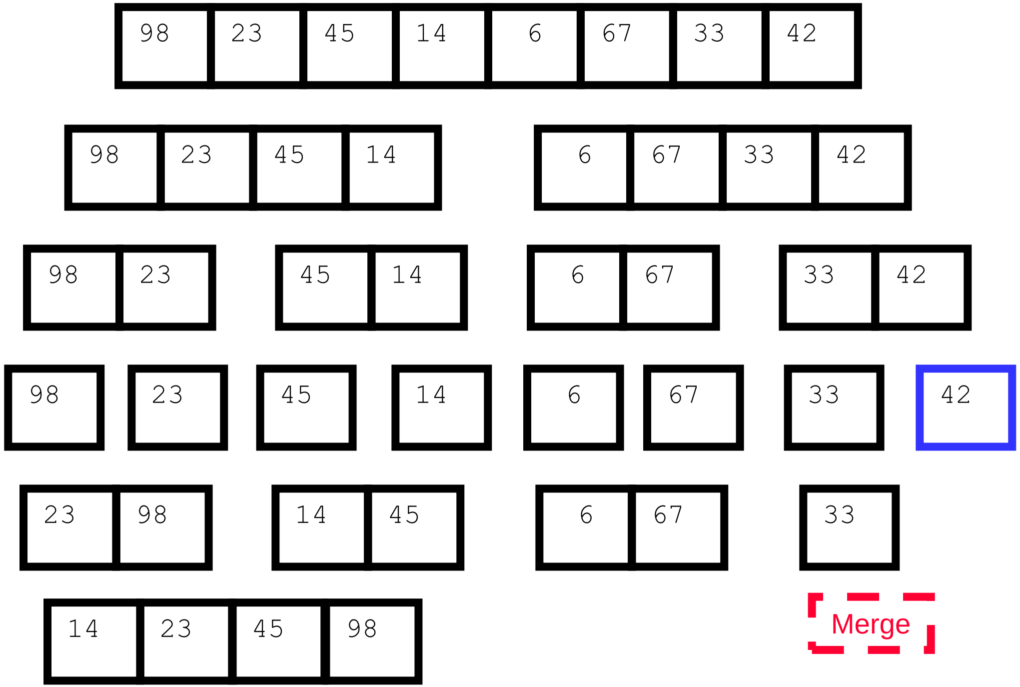

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

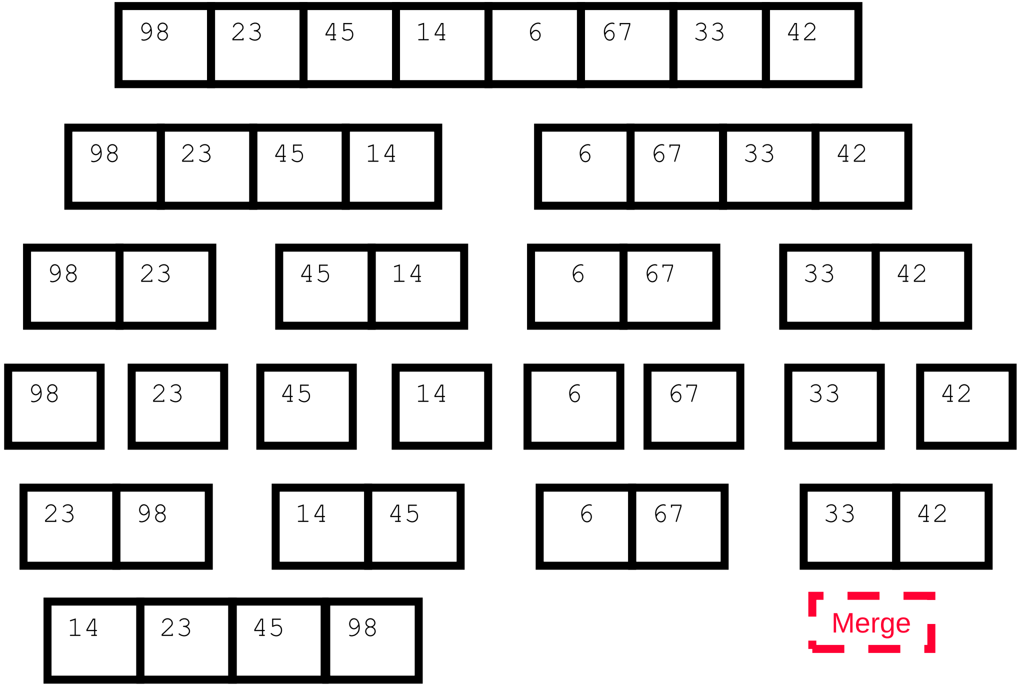

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge

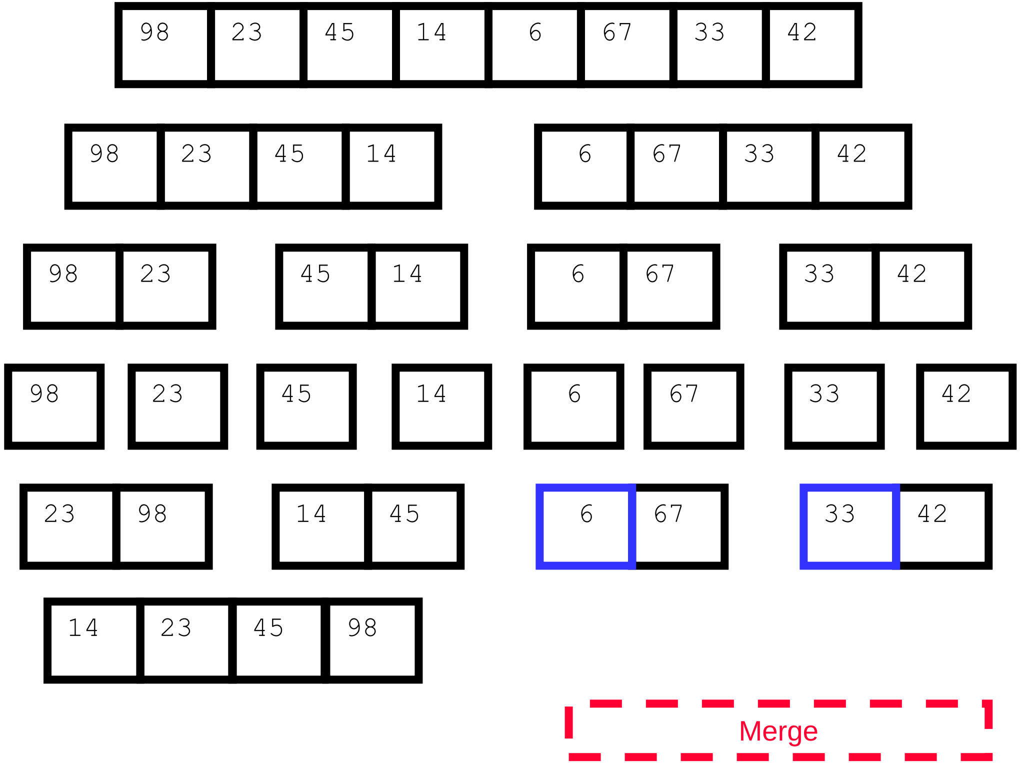

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6 67
14 23 45 98
Merge


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
Merge
14 23 45 98

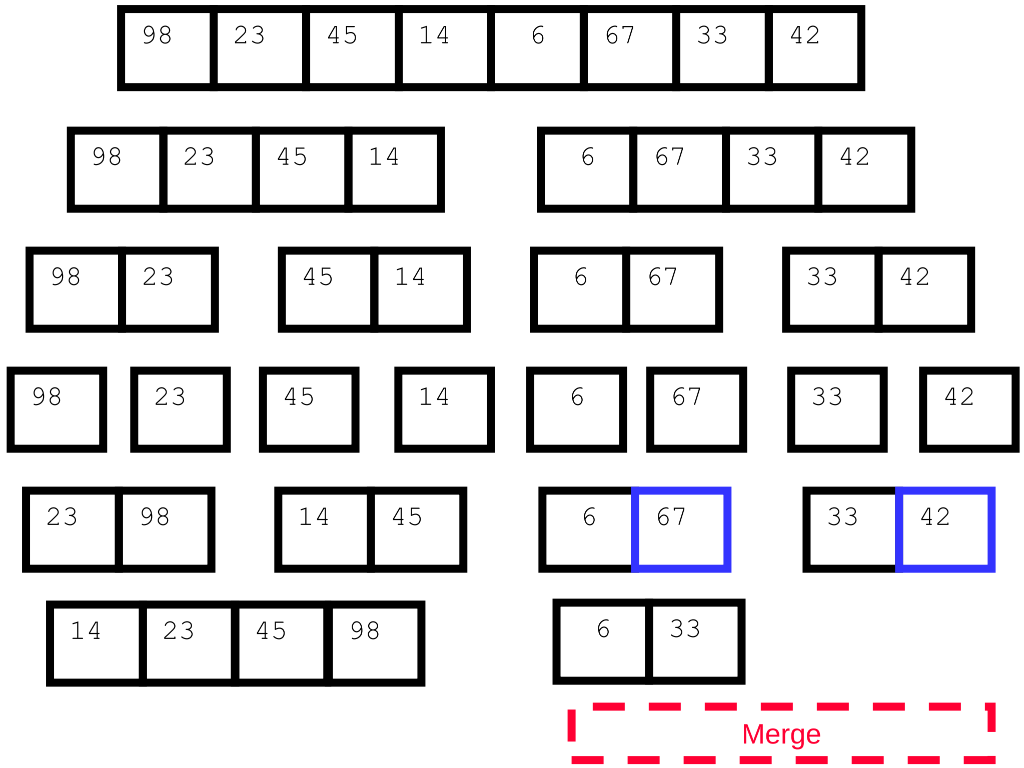

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge

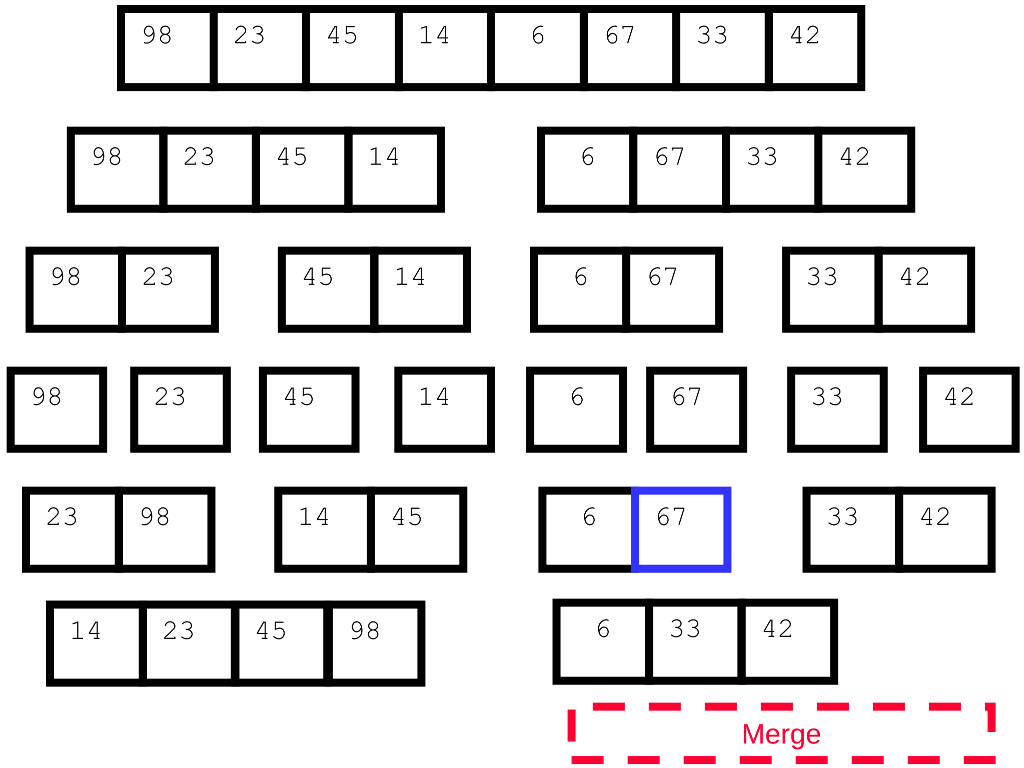

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

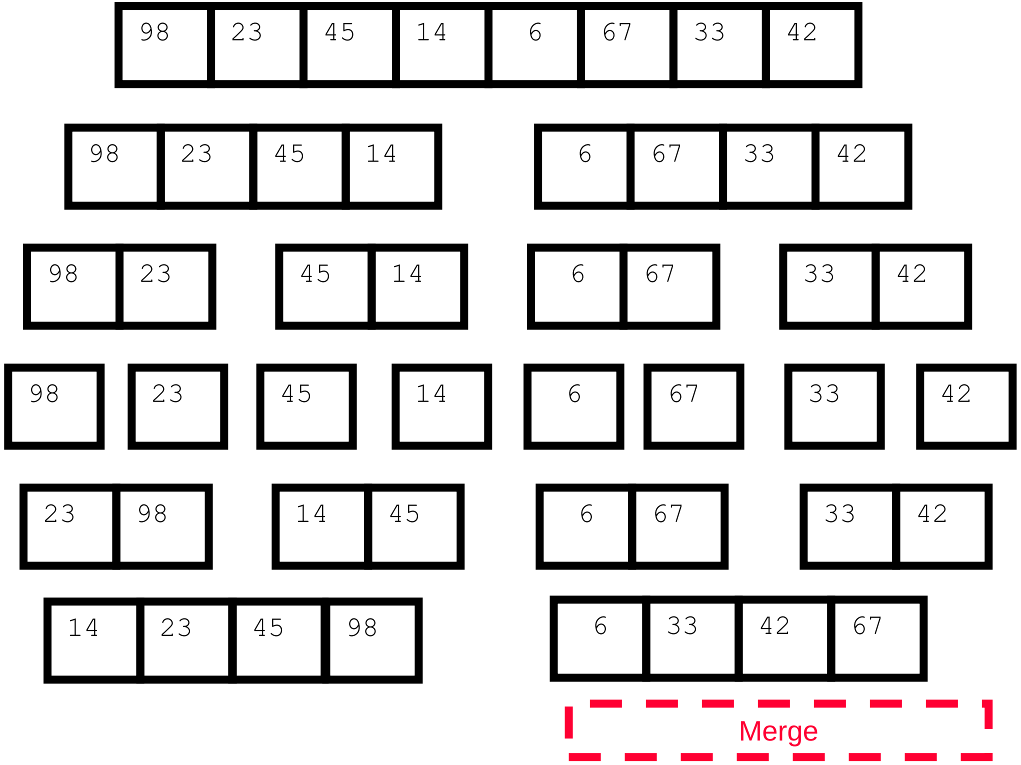


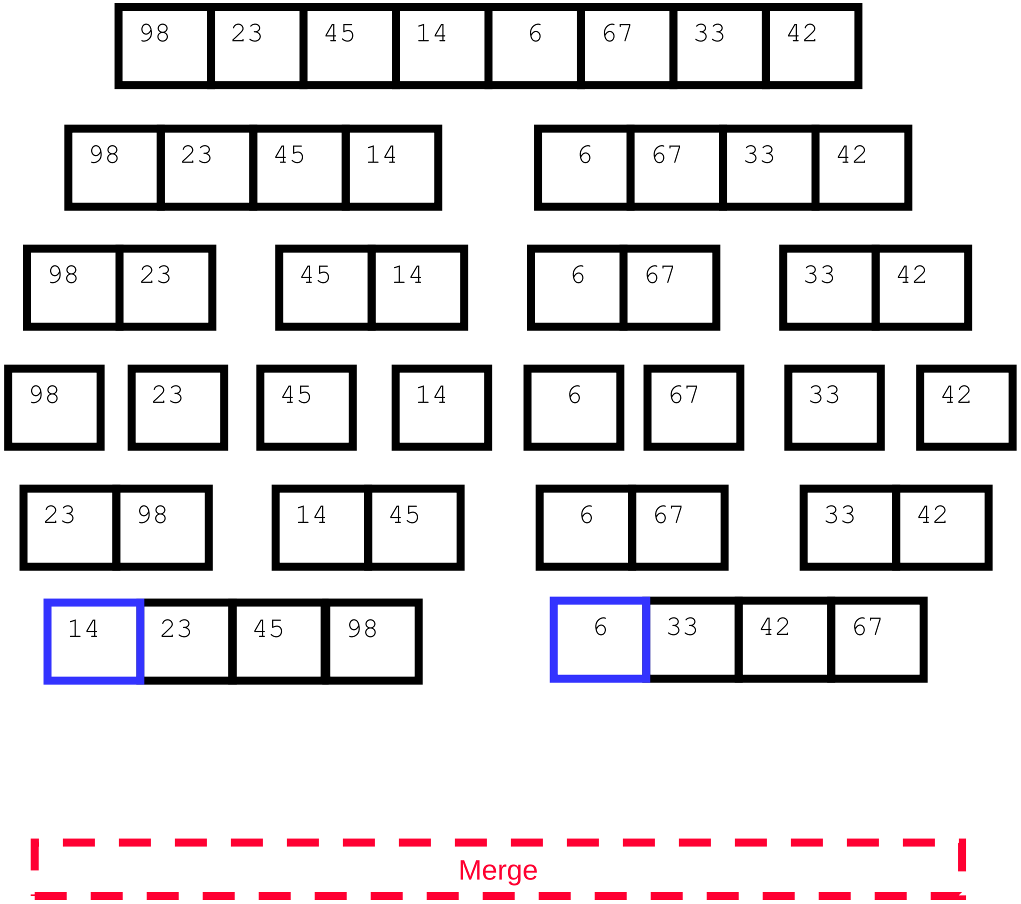
98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

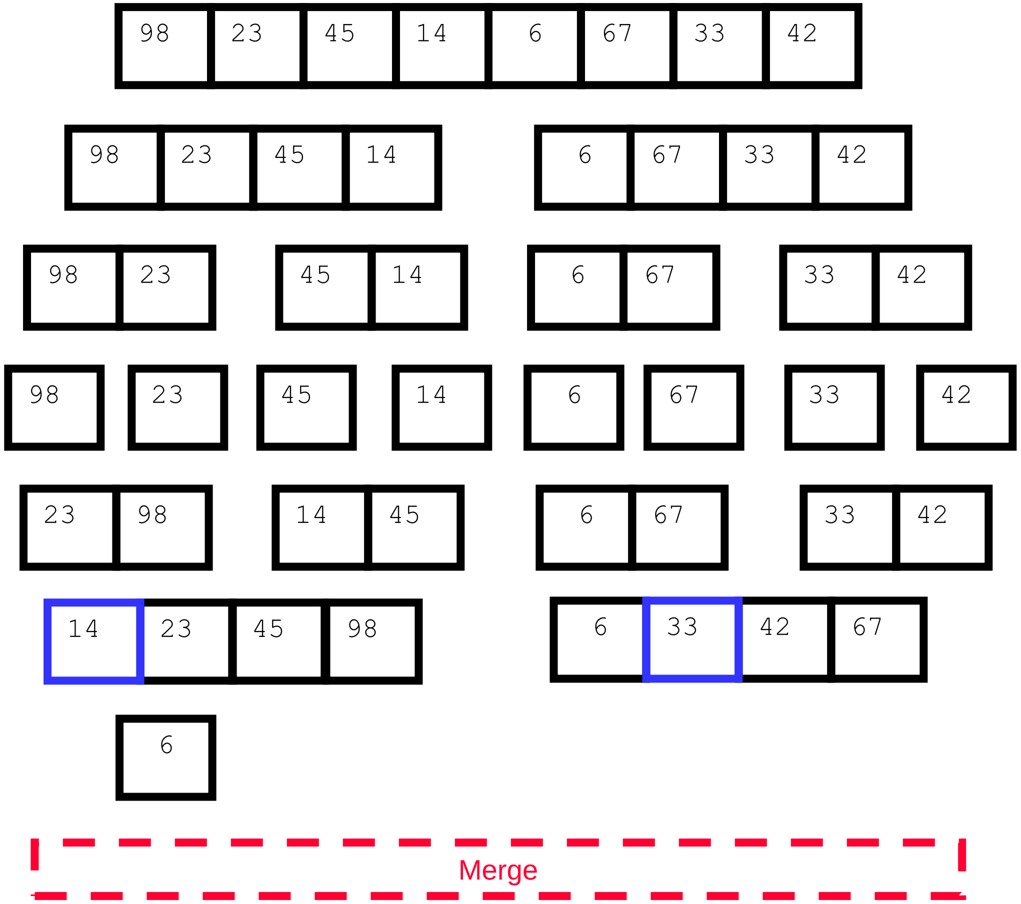
98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6
Merge

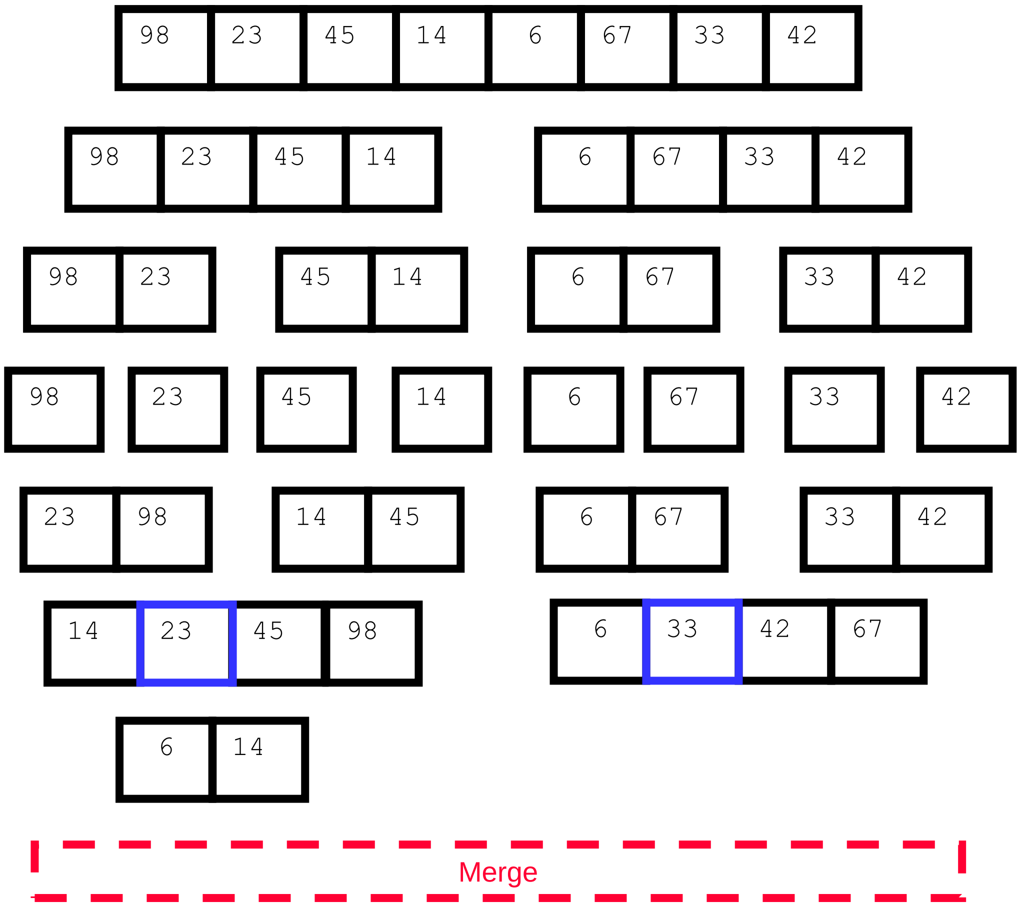

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33
Merge

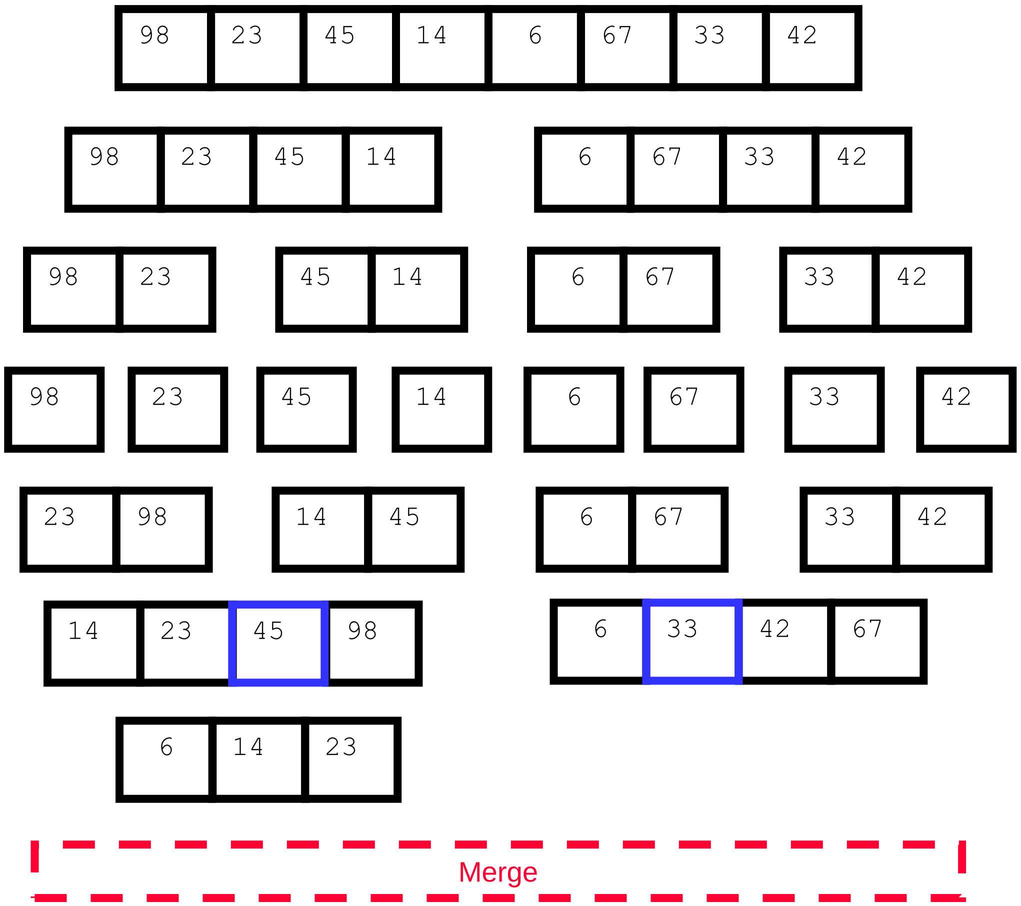

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42
Merge

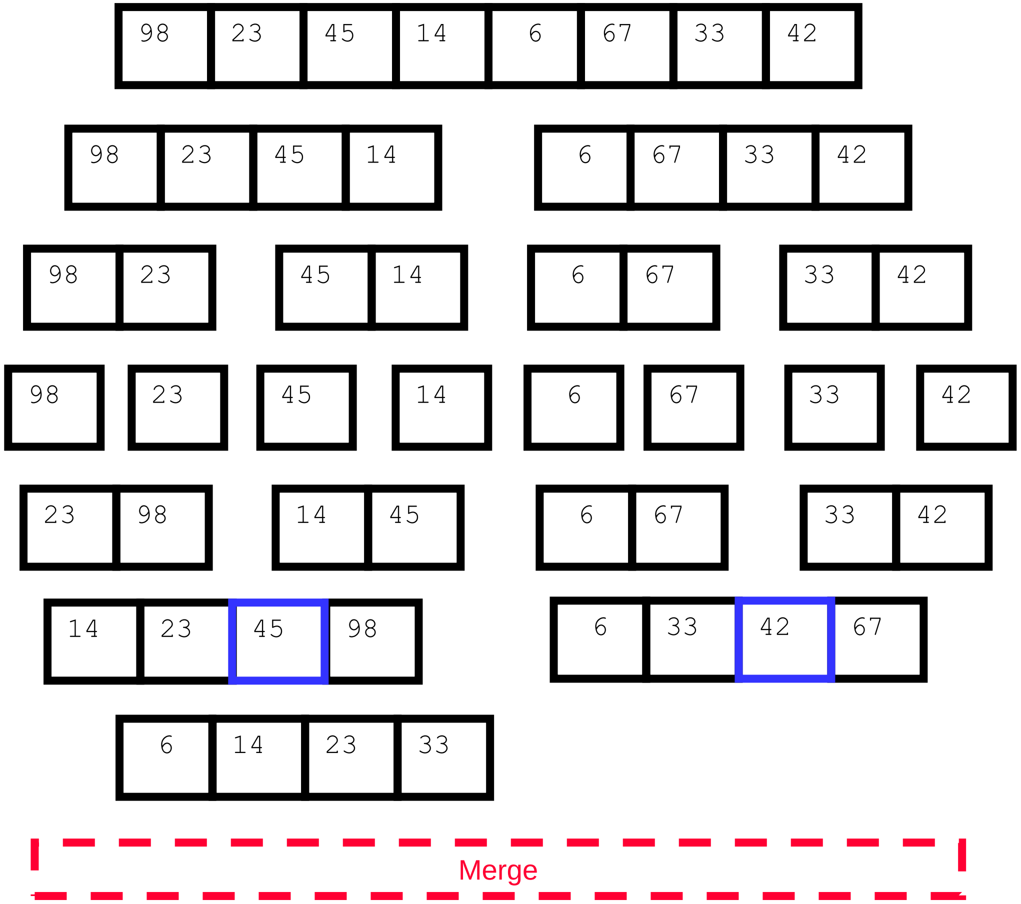

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

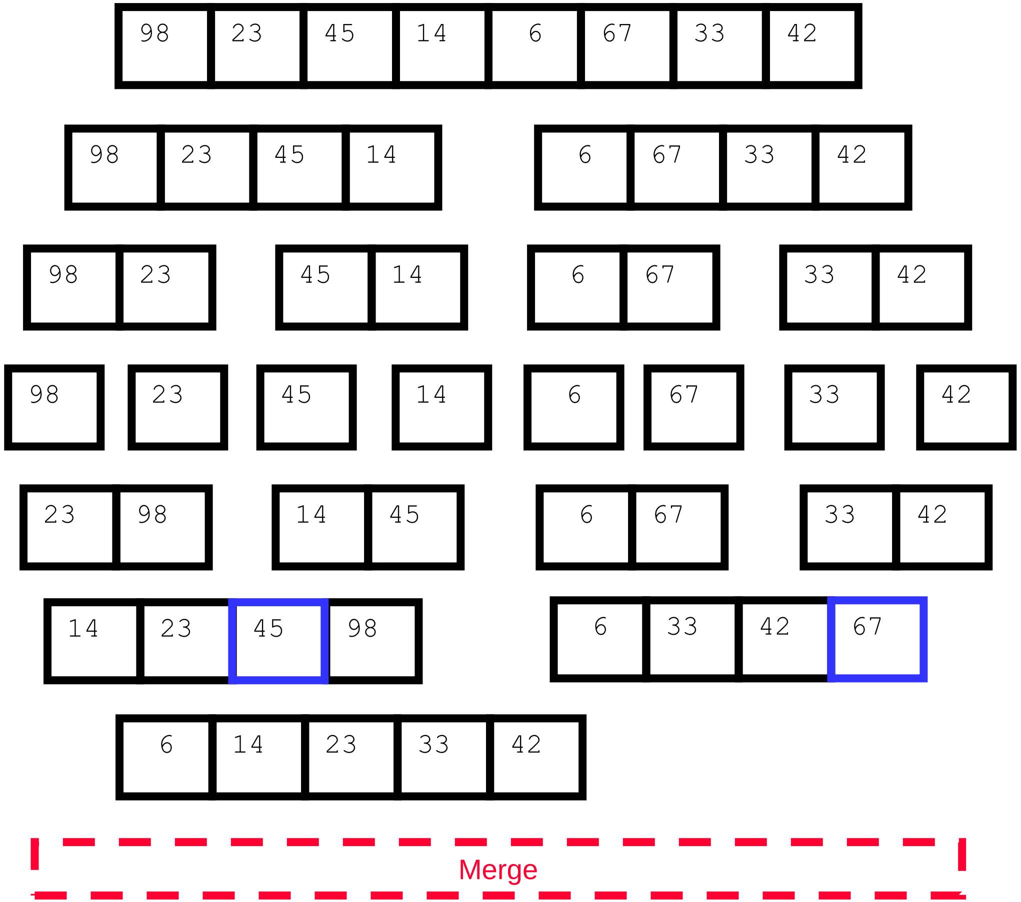

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

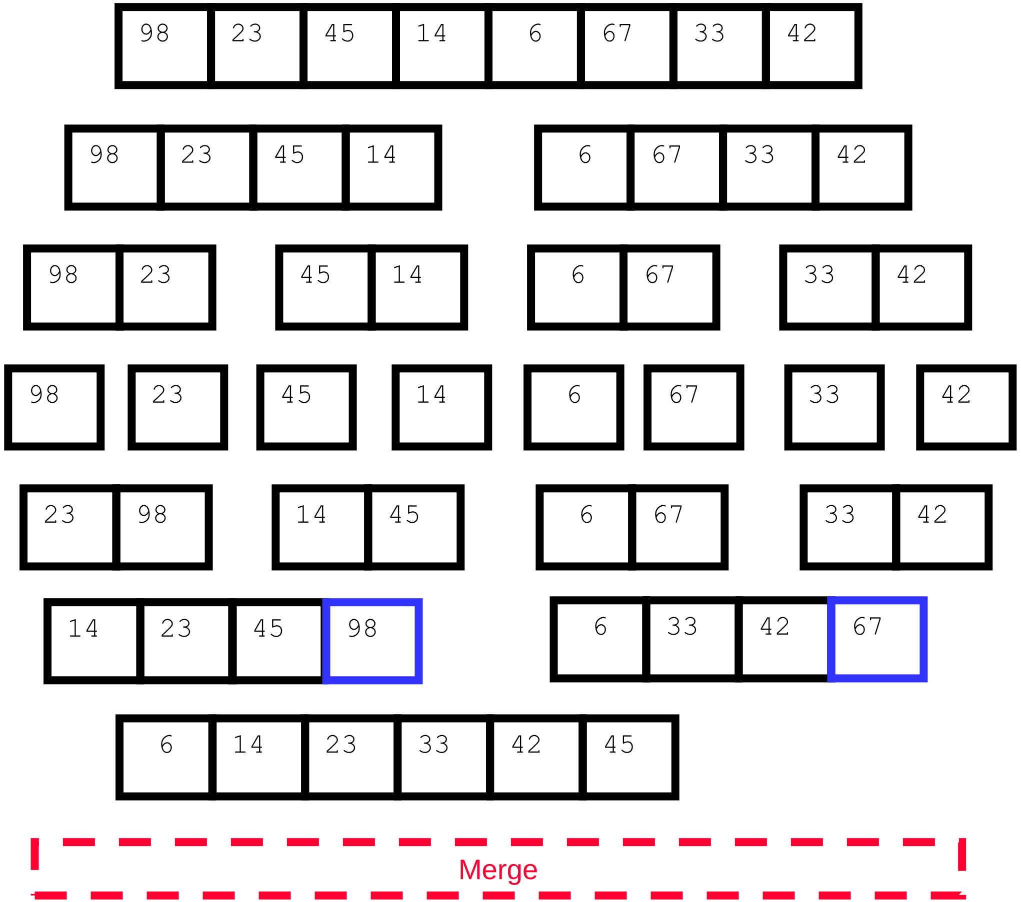

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6
Merge

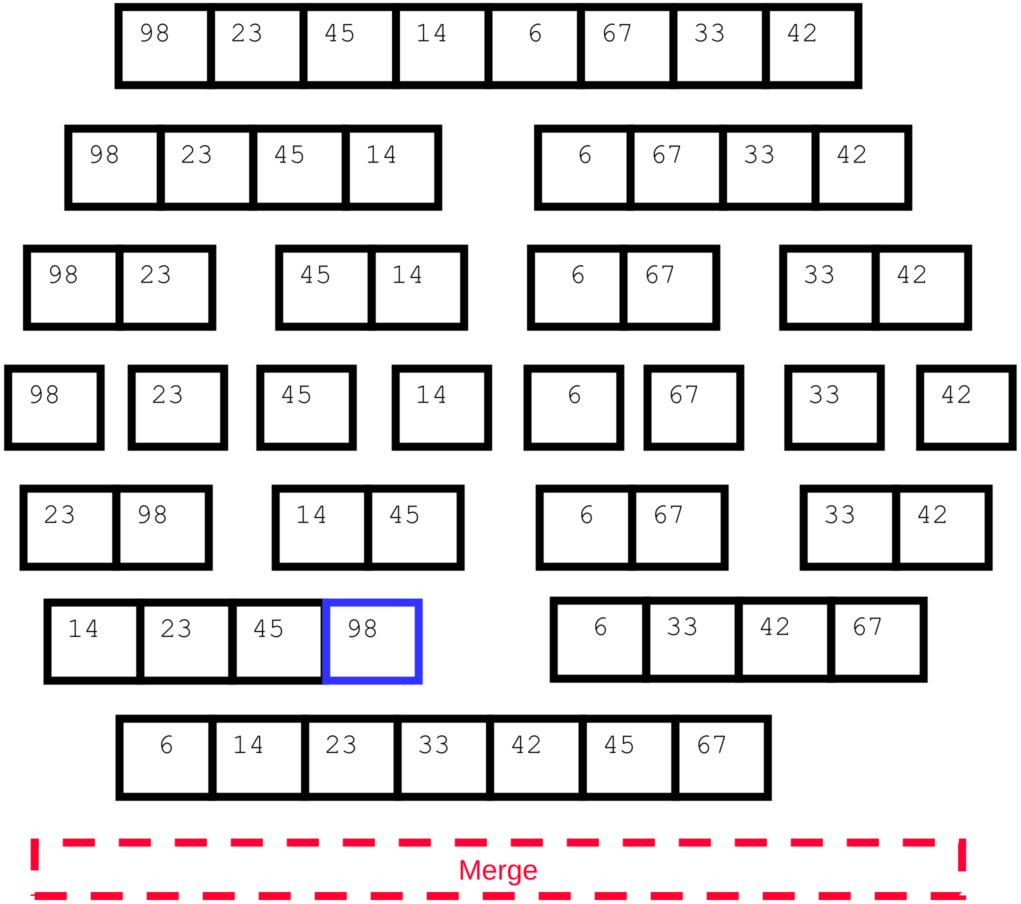

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14
Merge


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

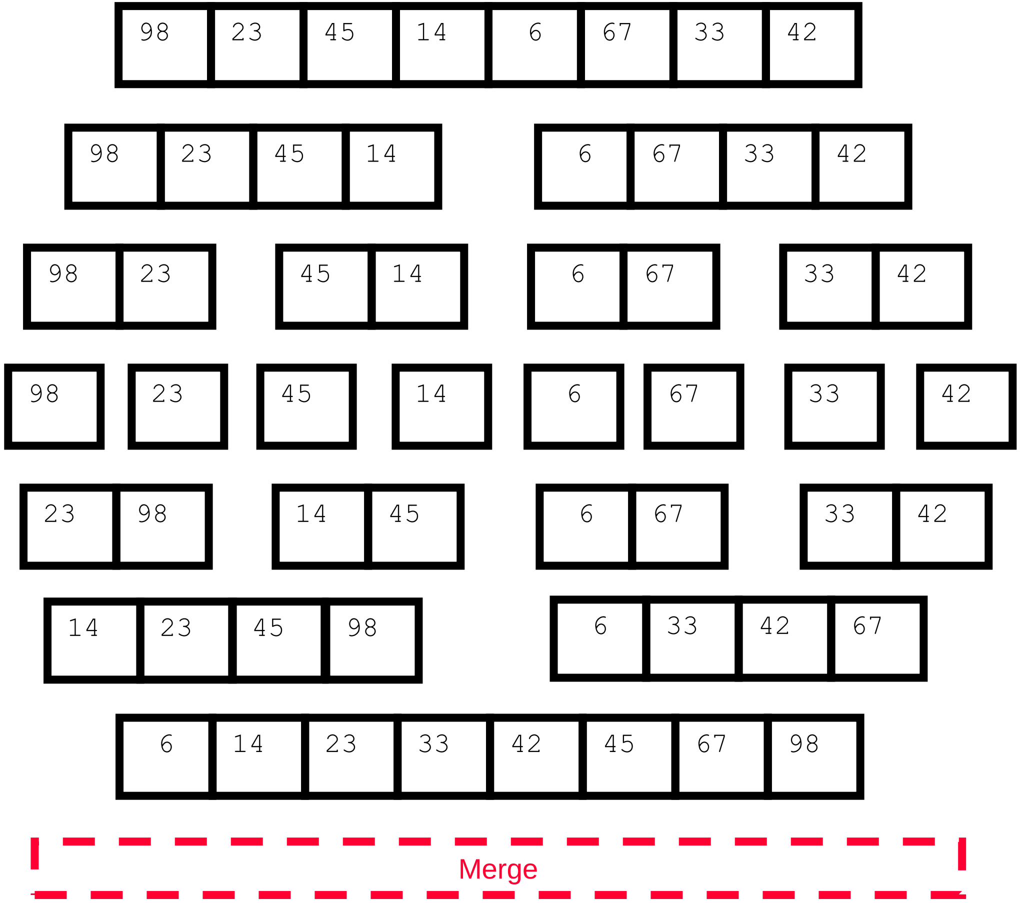

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33
Merge

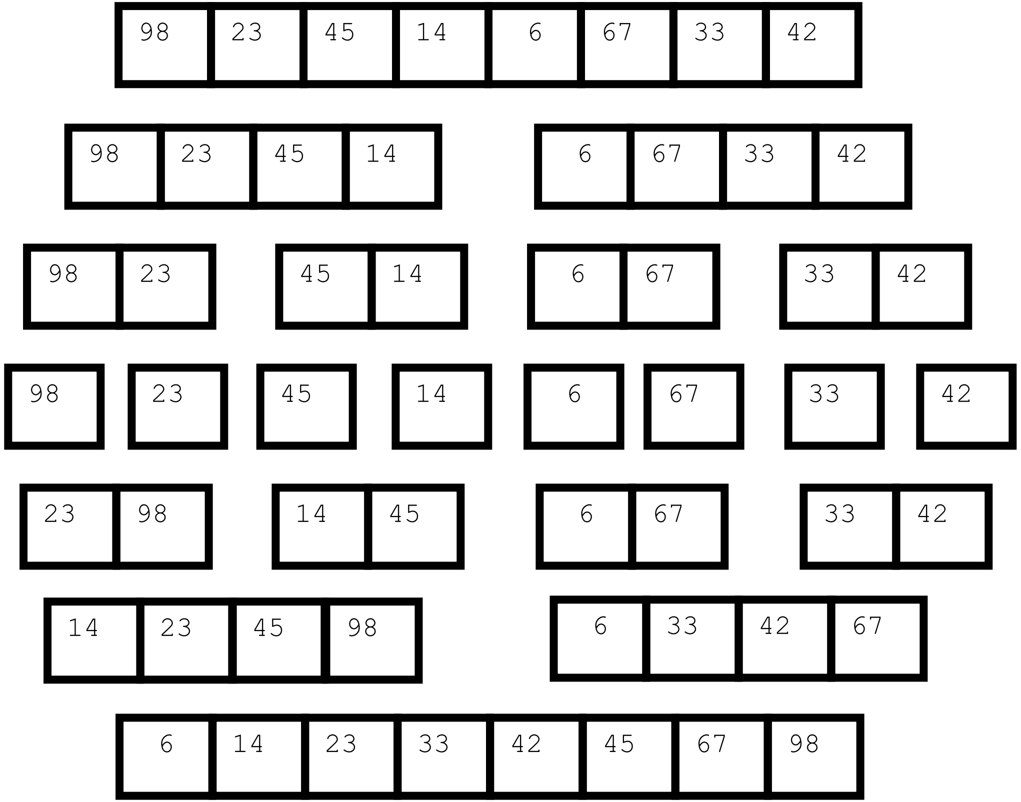

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42
Merge


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

Mergesort Visualization

About

Run Reset List size: 11

Your values: Type some array values, or click 'run' to use random values

1 / 156

<< < > >>

477	395	639	816	373	524	463	279	643	367	954
0	1	2	3	4	5	6	7	8	9	10

MergeSort Summary

1. Time Complexity
   - The time complexity of MergeSort is O(n log n) in all the 3 cases (worst, average and best) as the mergesort always divides the array into two halves and takes linear time to merge two halves.
2. Space Complexity: **O(n)**
3. Merge Sort is **stable**
4. Merge Sort is **not adaptive**
5. Merge Sort is **Out-of-Place (but can be in-place)**
6. Merge Sort is **recursive (but also comes in iterative ver.)**

Insertion

Implementation in Java

QuickSort

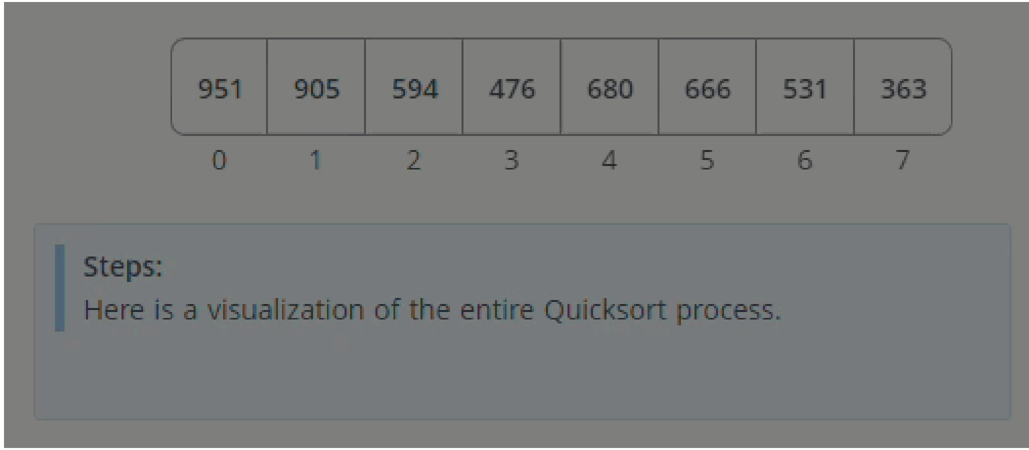
* There are many different versions of quickSort that pick pivot in different ways:

Always pick first element as pivot.
Always pick last element as pivot (used here)
Pick a random element as pivot – gives good results
Pick median as pivot.

Algorithm:

mergeSort(arr[], l, r)
If r > l

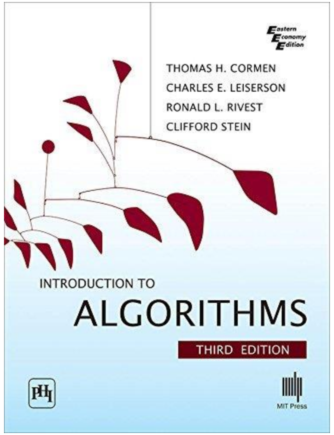
Step 1: Make the right-most* index value pivot
Step 2: partition the array using pivot value
Step 3: quicksort left partition recursively
Step 4: quicksort right partition recursively



The key process in quickSort is partition
If the list is empty or has one item, it is sorted by definition (the base case)

Outcome of partition is, given arr and pivot x, put x at its correct position in sorted array and put all elements smaller than x before x, and put all elements greater than x after x. Partition is **Linear time.**

QuickSort (from CLRS)

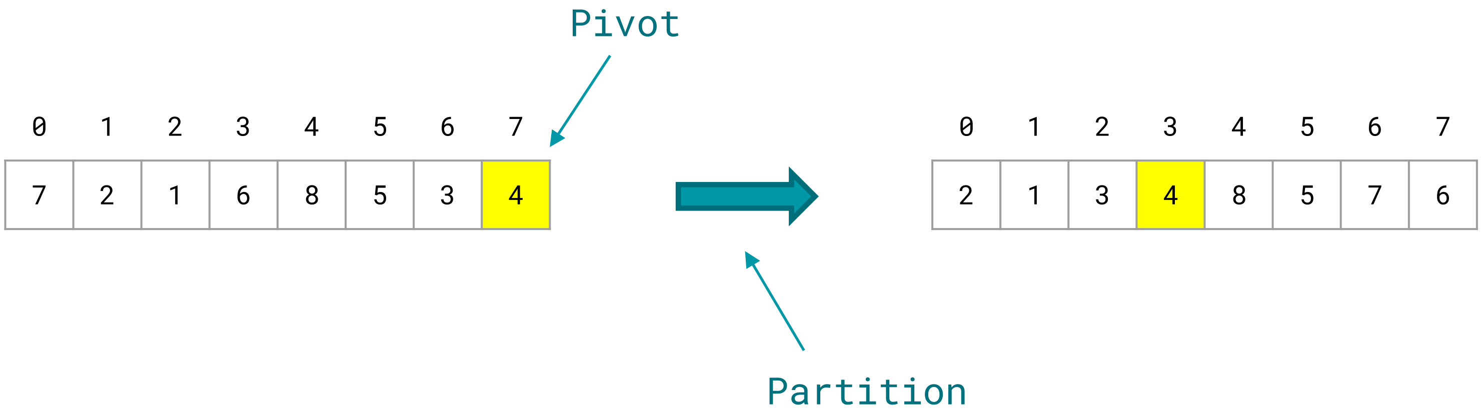



```
QUICKSORT(A, p, r)  /* p  --> Starting index,  r  --> Ending index */
if p < r
    q = PARTITION(A, p, r)  /* q is partitioning index, A[q] is now at right place */
    QUICKSORT(A, p, q − 1)  // Before q
    QUICKSORT(A, q + 1, r)  // After q
```

```
PARTITION(A, p, r)
x = A[r]  // x (Element to be placed at right position)
i = p − 1 // Index of smaller element and indicates the right position of pivot found so far
for j = p to r − 1
    if A[j] ≤ x  // If current element is smaller than the pivot
        i = i + 1 // increment index of smaller element
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

The partion function

To partition an array with respect to an element (the “pivot”) means to rearrange the array (following a specific algorithm!) so that all the elements smaller than the pivot is to its left and all the elements larger than the pivot is to its right.



QuickSort

	0	1	2	3	4	5	6	7
	7	2	1	6	8	5	3	4
i	j							

PARTITION(A, p, r)

```
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

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

i j

PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$

QuickSort

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

i j

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
7	2	1	6	8	5	3	4
i	j						

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
2	7	1	6	8	5	3	4
i	j						

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
2	7	1	6	8	5	3	4
i	j						

PARTITION(A, p, r)

```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
2	7	1	6	8	5	3	4
	i	j					

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
2	1	7	6	8	5	3	4
	i	j					

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
2	1	7	6	8	5	3	4
	i		j				

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
2	1	7	6	8	5	3	4
	i			j			

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
2	1	7	6	8	5	3	4
	i				j		

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
2	1	7	6	8	5	3	4
		i				j	

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort

0	1	2	3	4	5	6	7
2	1	3	6	8	5	7	4
		i				j	

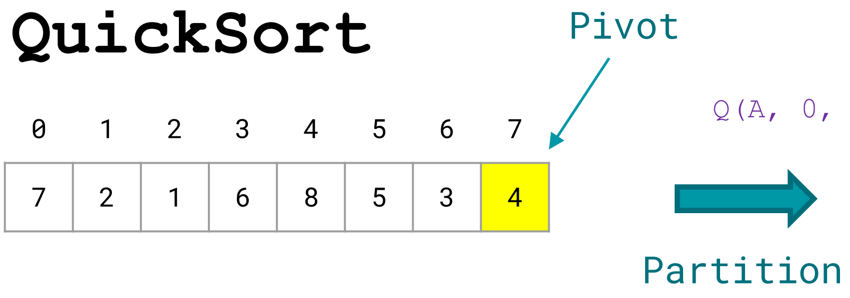
```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

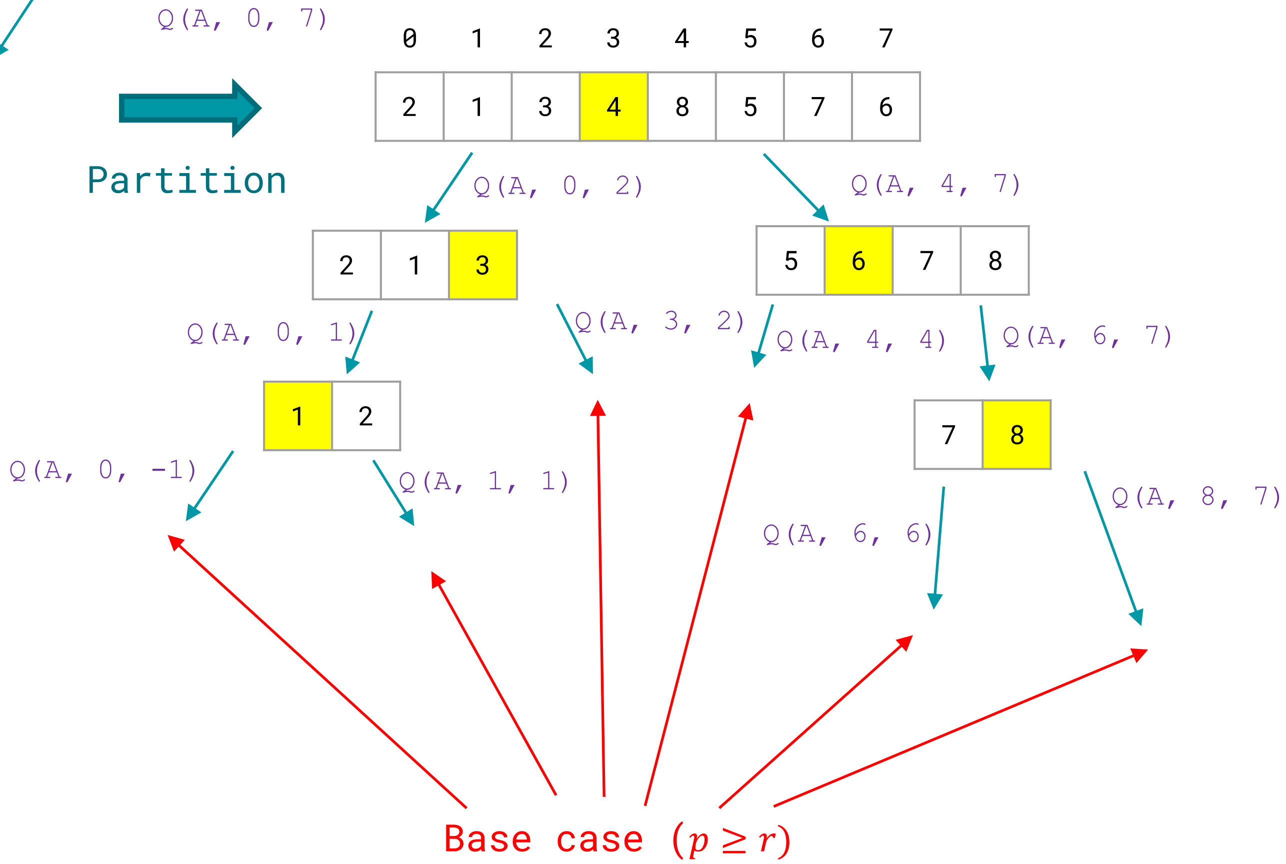
QuickSort

0	1	2	3	4	5	6	7
2	1	3	4	8	5	7	6
		i				j	

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort





QUICKSORT(A, p, r)

1 **if** $p < r$
2 $\quad q = $ PARTITION(A, p, r)
3 $\quad$ QUICKSORT$(A, p, q-1)$
4 $\quad$ QUICKSORT$(A, q+1, r)$

Example

		p									r	
initially:		2	5	8	3	9	4	1	7	10	**6**	
	i	j										
next iteration:		2	5	8	3	9	4	1	7	10	**6**	
		i	j									
next iteration:		2	5	8	3	9	4	1	7	10	**6**	
			i	j								
next iteration:		2	5	8	3	9	4	1	7	10	**6**	
			i		j							
next iteration:		2	5	3	8	9	4	1	7	10	**6**	
				i		j						

note: pivot (x) = 6

PARTITION(A, p, r)

```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Example (Continued)

next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i		j					
next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i			j				
next iteration:	2	5	3	4	9	8	1	7	10	**6**
				i			j			
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i			j		
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i				j	
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i					j
after final swap:	2	5	3	4	1	**6**	9	7	10	8
					i					j

```
PARTITION(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

QuickSort Summary

1. Time Complexity
   - Despite having a worst case running time of $O(n^2)$, Quick Sort is pretty fast and efficient in practical scenarios
   - The worst-case running time of Quick Sort is almost always avoided by using what we call a randomized version of Quick Sort which gives us O(nlogn) running time with very high probability
   - In fact sort function given to us by most of the language libraries are implementations of Quick Sort only

2. Space Complexity: **O(log in)** (depending on pivot method)
3. QuickSort is **not stable** (because of swap relative to pivot)
4. QuickSort is **adaptive**
5. QuickSort is **in-Place**
6. QuickSort is **recursive**

Implementation in Java

Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)