Data Structure and Algorithm

Searching and Sorting

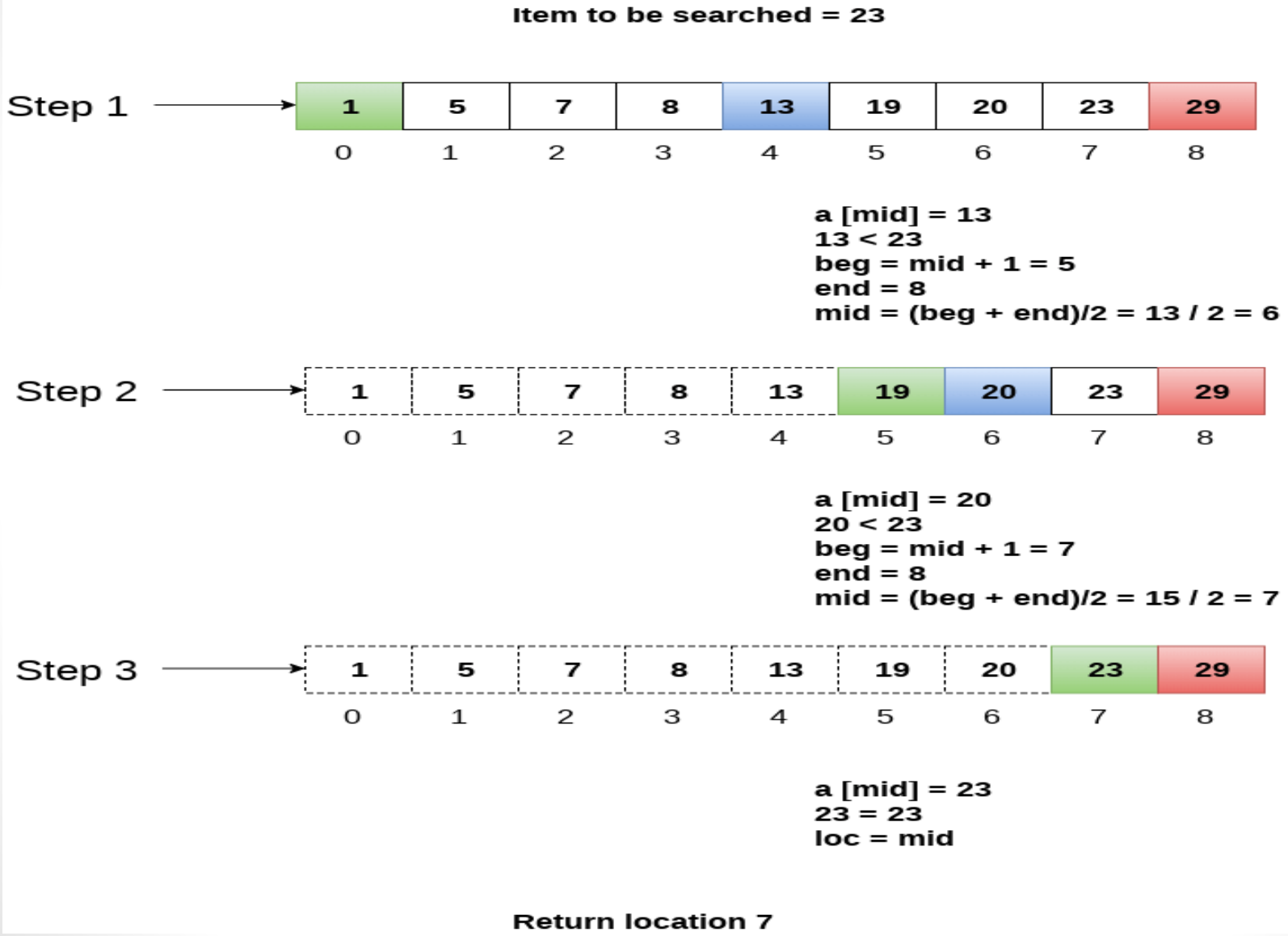
Kanak Kalyani

Searching

Linear Search:- *sequential search*

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 20 // Added so the size of the array can be altered more easily
int main(int argc, char *argv[]) {
int arr[size], num, i, n, found = 0, pos = -1;
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements: ");
for(i=0;i<n;i++)
{
        scanf("%d", &arr[i]);
}
printf("\n Enter the number that has to be searched : ");
scanf("%d", &num);
for(i=0;i<n;i++)
{
   if(arr[i] == num)
   {
        found =1;
        pos=i;
        printf("\n %d is found in the array position= %d", num,i+1);
        break;
  }
}
if (found == 0)
printf("\n %d does not exist in the array", num);
return 0;
}
```

Binary Search

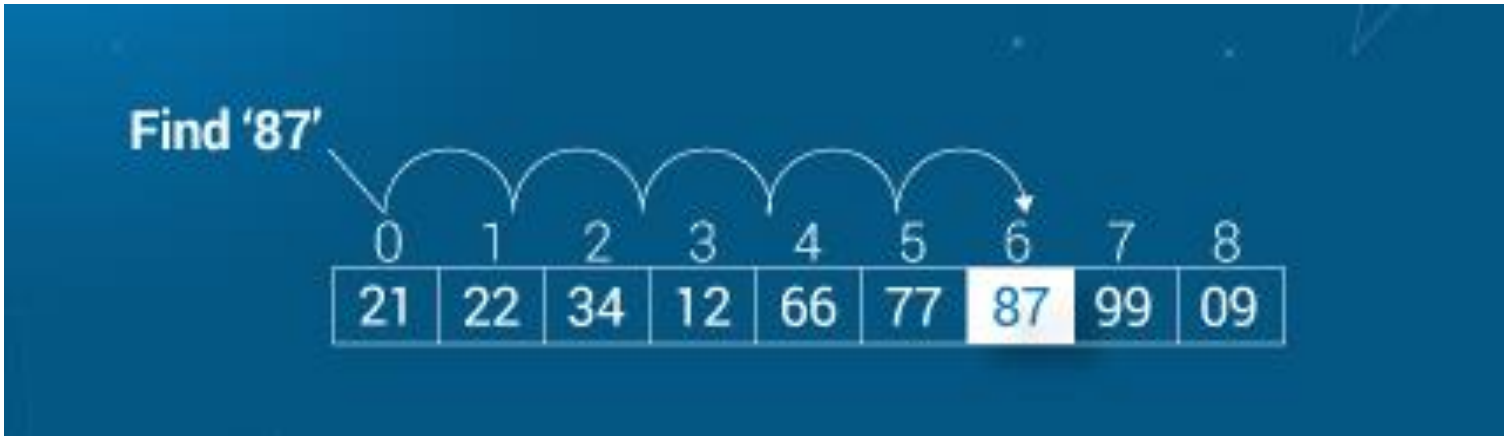


Kanak Kalyani

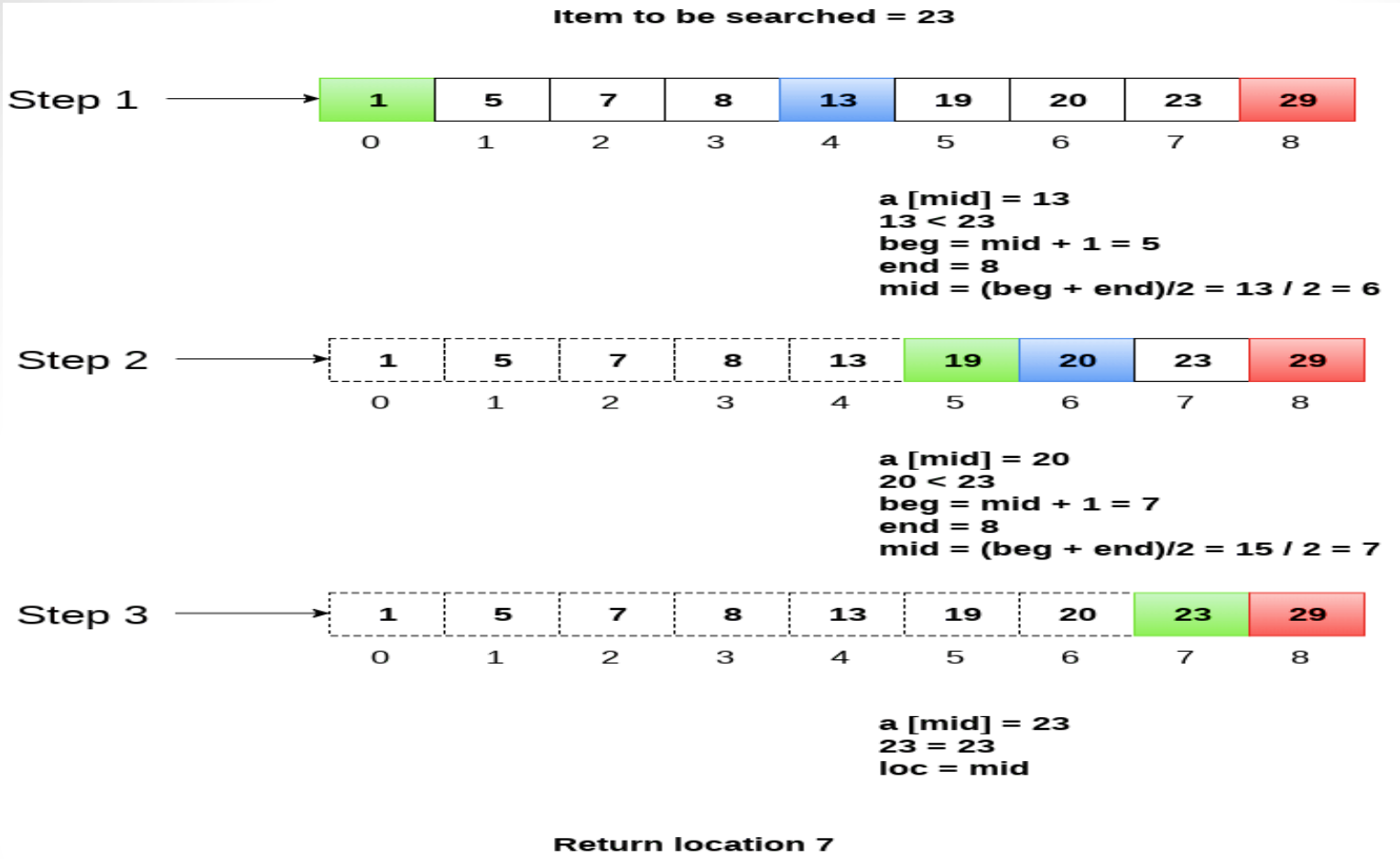
```
int BinarySearch(int[] arr, int n){
    beg = 0, end = n-1;
    while(beg<=end)
    {
        mid = (beg + end)/2;
        if (arr[mid] == num)
        {
        printf("\n %d is present in the array at position %d", num, mid+1);
        found =1;
        break;
         }
        else if (arr[mid]>num)
          end = mid-1;
        else
          beg = mid+1;
    }
    if (beg > end && found == 0)
    printf("\n %d does not exist in the array", num);
    return 0;
}
```

Complexity Search Algorithm

Linear Search



Space Complexity:- O(1)
 # Comparisons:- n
f(n)=n
Time Complexity
a. Best case = O(1)
b. Average case = n(n+1)/2n = O(n)
c. Worst case = O(n)



Binary Search

Space Complexity:- O(1)

$f(n) = \log_2 n$

Time Complexity

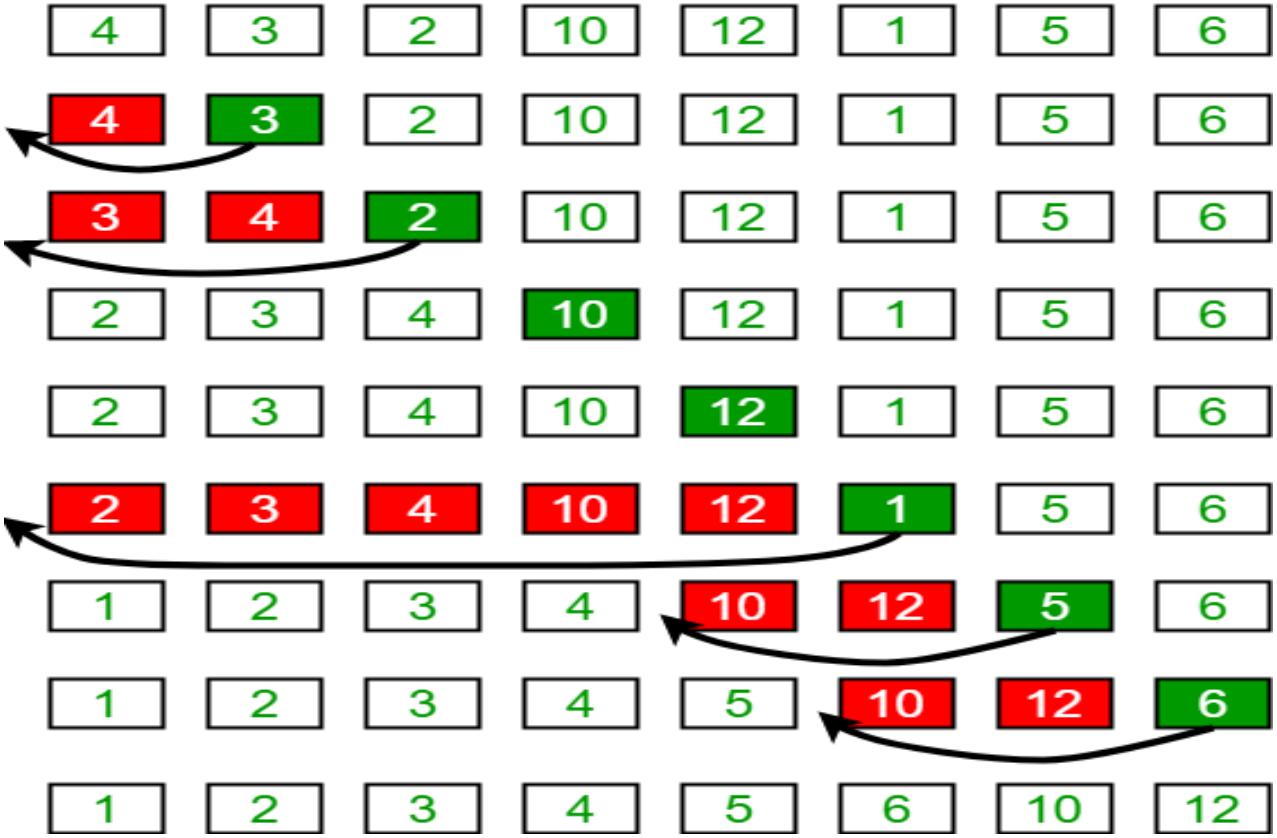
- **Best case = O(1)**
- **Average case =**

  log n(log n+1)/2log n = O(log n)
- **Worst case = O(logn)**

Sorting

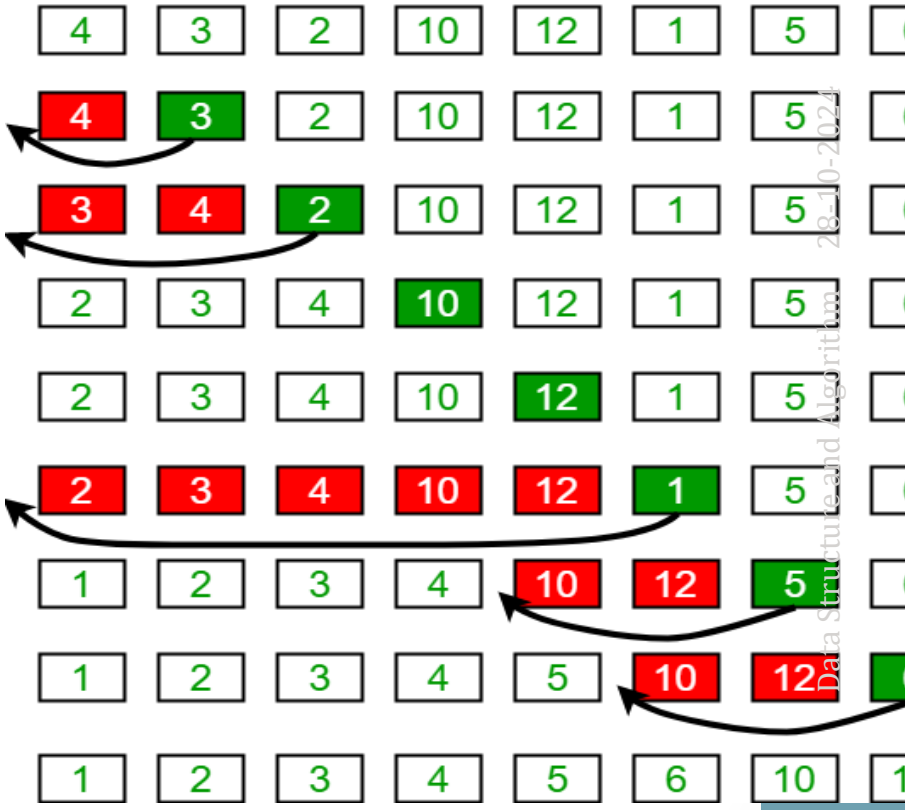
- Insertion Sort
- Shell Sort
- Counting Sort
- Merge Sort
- Quick Sort
- Heap Sort

Insertion Sort



Insertion Sort

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
       while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```



Insertion Sort

Time Complexity

```
for (i = 1; i < n; i++) {    //Outer Loop
…
            while (j >= 0 && arr[j] > key) {Inner Loop
             …..
```

Comparison: $1 + 2+ 3 +4 \ldots + n-1 = n * (n +1)/2 = n^2/2+n/2$

Time Complexity: $O(n^2)$

Space Complexity: O(1)

Shell Sort

Step 1: Arrange the elements of the array in the form of a table and sort the columns (using
insertion sort).
Step 2: Repeat Step 1, each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted.

63, 19, 7, 90, 81, 36, 54, 72, 27, 22, 9, 41, 59, 33

Arrange the elements of the array in the form of a table and sort the columns

Step-1

63	19	7	90	81	36	54	
72	27	22	9	41	59	33	

Result

63	19	7	9	41	36	33	
72	27	22	90	81	59	54	

63	19	7	9	41	36	33	72	27	22	90	81	59	54

- Step- 2

63	19	7
9	41	36
33	72	27
22	90	81
59	54	

9	19	7
22	41	27
33	54	36
59	72	81
63	90	

9	19	7	22	41	27	33	54	36	59	72	81	63	90	

- Step-3

9	19	7	22	41	27	33	54	36	59	72	81	63	90	

7	9	19	22	41	27	33	54	36	59	72	81	63	90	

7	9	19	22	41	27	33	54	36	59	72	81	63	90	

7	9	19	22	27	41	33	54	36	59	72	81	63	90	

7	9	19	22	27	33	41	54	36	59	72	81	63	90	

.
.
.

7	9	19	22	27	33	36	41	54	59	72	63	81	90	

```
void shellsort(int arr[], int num){
 int i, j, k, tmp;
for (i = num / 2; i > 0; i = i / 2) {
        for (j = i;  j < num; j++) {
                for(k = j - i; k >= 0; k = k - i) {
                        if (arr[k+i] >= arr[k])
                                break;
                        else {
                                tmp = arr[k]; arr[k] = arr[k+i];
                                arr[k+i] = tmp;
                        }
                }       }}      }}
```

Shell Sort Complexity

```
for (i = num / 2; i > 0; i = i / 2) {    // log2n
        for (j = i;  j < num; j++) {  // loop 2
                for(k = j - i; k >= 0; k = k - i) {//loop 3
                        if (arr[k+i] >= arr[k])
```

Loop 2 and Loop3:

n/n + . . . +n/16 +n/8+n/4+n/2

=n(1/2 + 1/4 + 1/8 + 1/16 + ... + 1/n) //harmonic series

= n * $\log_2 n$

Time Complexity: -

Worst Case:- O(n $(\log_2 n)^2$)

Best Case:- O(n log n)

Space Complexity:- O(1)

Input Size (n)	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n(\log n)^2)$	$O(n^2)$	$O(2^n)$
10	3	10	33	99	100	1024
100	7	100	700	4900	10,000	1.27×10^{30}
1,000	10	1,000	10,000	100,000	1,000,000	Huge!
10,000	13	10,000	130,000	1,690,000	100,000,000	-
100,000	17	100,000	1,700,000	28,900,000	10,000,000,000	-
1,000,000	20	1,000,000	20,000,000	400,000,000	1,000,000,000,000	-

Advantages

- It can sort large arrays more efficiently than bubble sort and plain insertion sort.
- Shell sort is useful for moderately sized datasets where other algorithms may have more overhead.
- Shell sort requires very little additional space, using only a constant amount of memory (O(1) extra space).
- **It performs well on data that is already partially sorted, requiring fewer swaps and comparisons in such cases.**
- It is relatively simple to code compared to more complex algorithms like quicksort or heapsort.
- Although the worst-case time complexity is O(n2)O(n^2)O(n2) with some gap sequences, in practice, Shell sort often runs much faster due to fewer data movements and comparisons.

Merge Sort

Merging

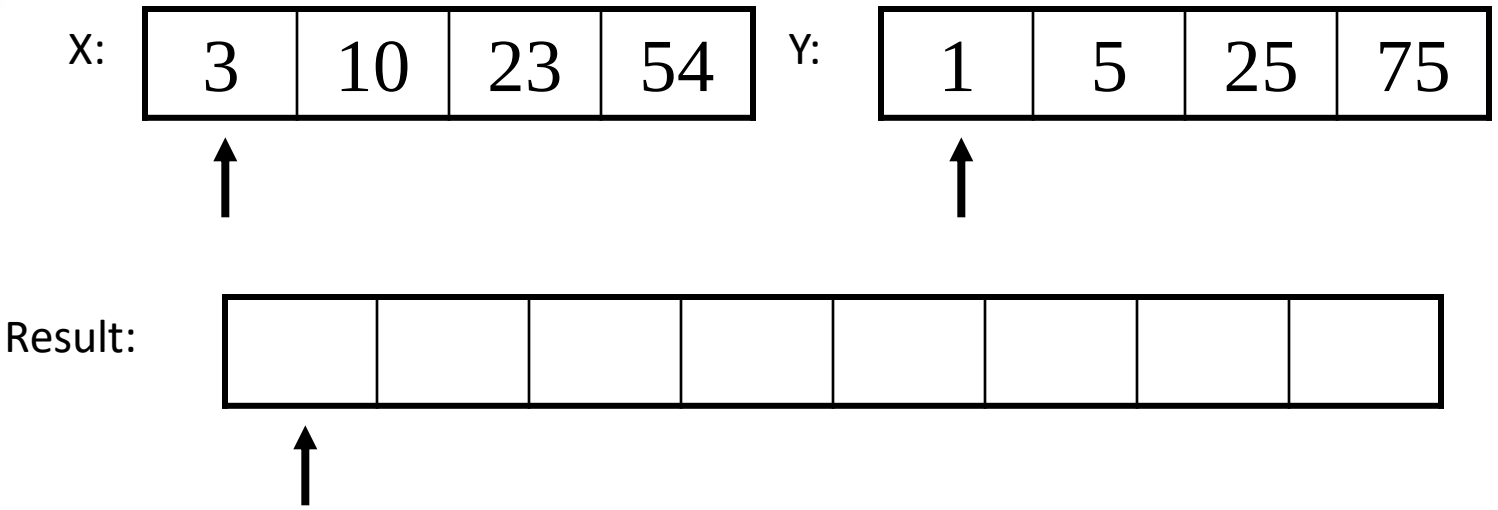
- The key to Merge Sort is merging two sorted lists into one, such that if you have two lists X ($x_1 \leq x_2 \leq \cdots \leq x_m$) and Y($y_1 \leq y_2 \leq \cdots \leq y_n$) the resulting list is Z($z_1 \leq z_2 \leq \cdots \leq z_{m+n}$)

- Example:

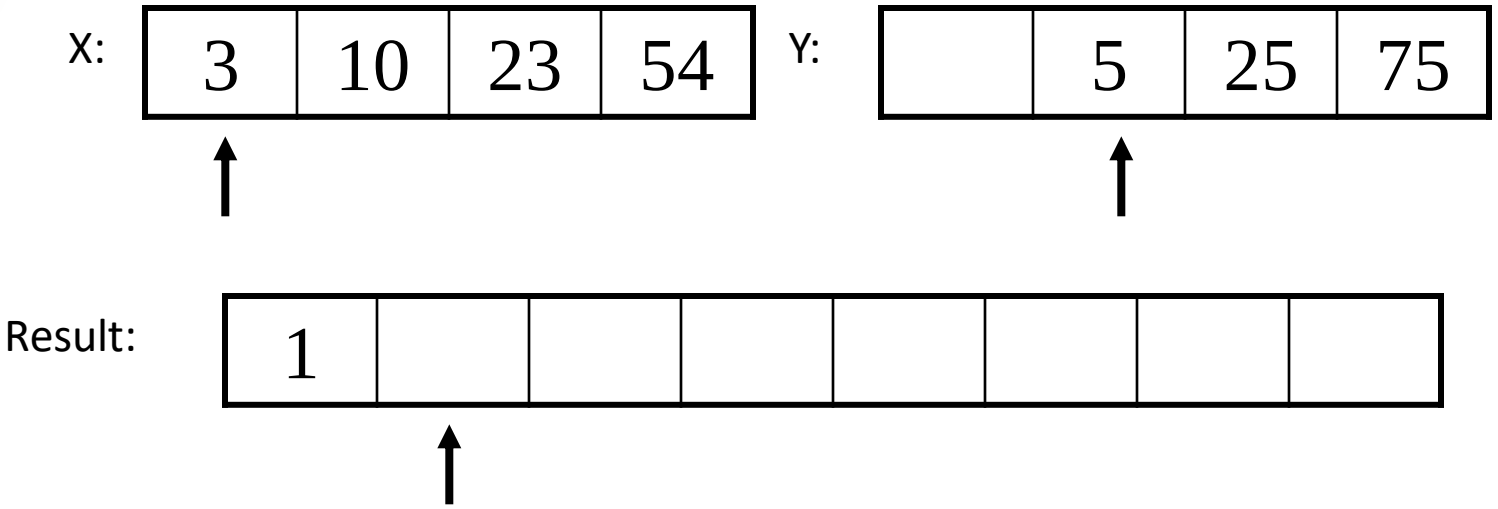
L_1 = { 3 8 9 } L_2 = { 1 5 7 }

merge(L_1, L_2) = { 1 3 5 7 8 9 }

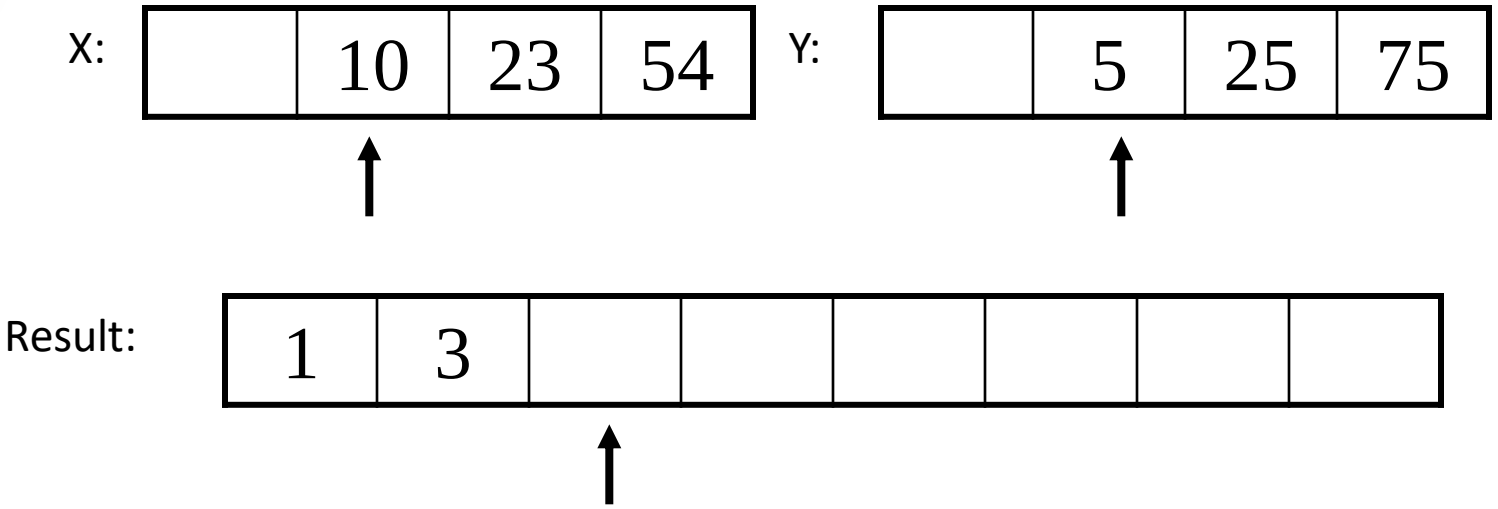
Merging (cont.)



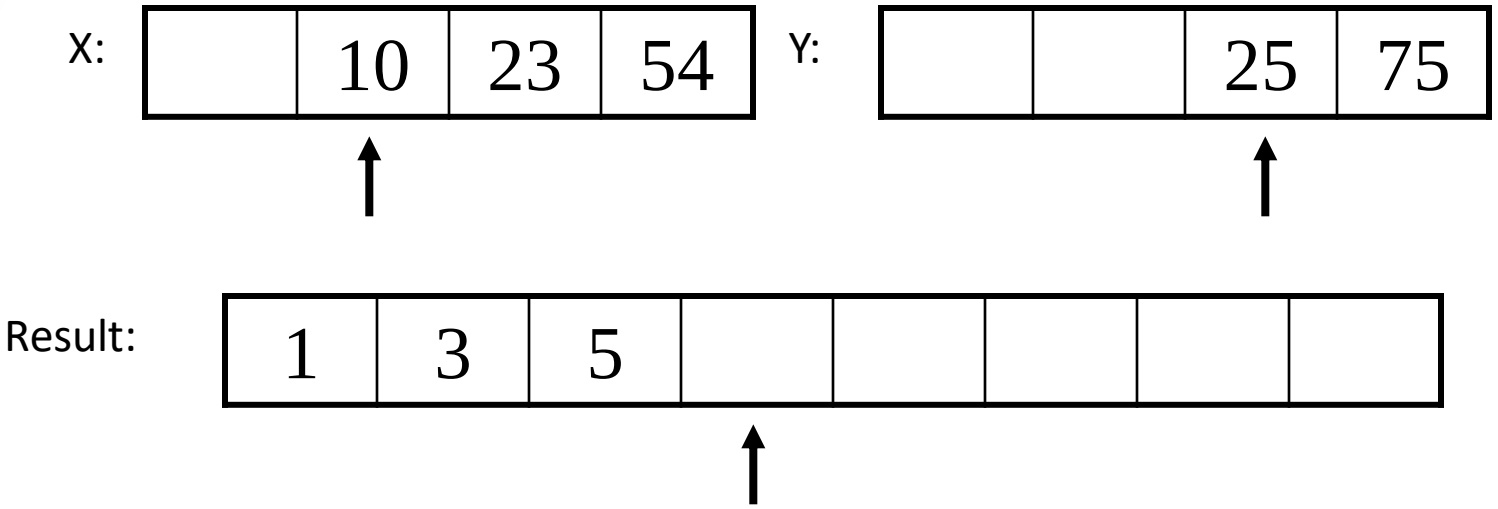
Merging (cont.)



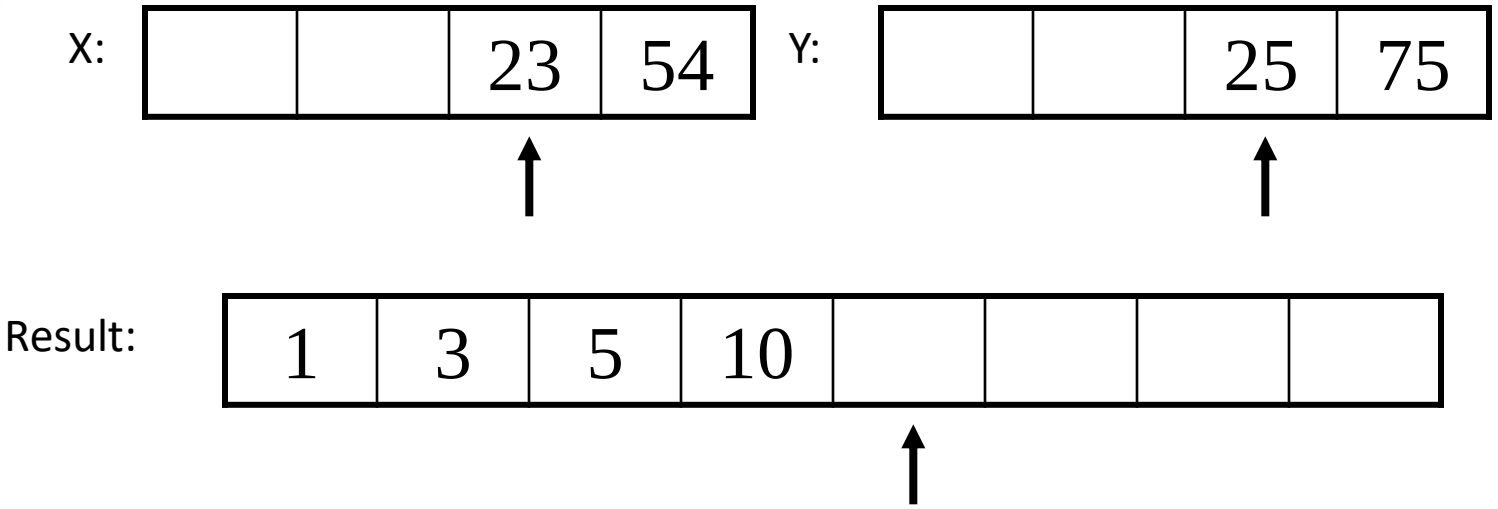
Merging (cont.)



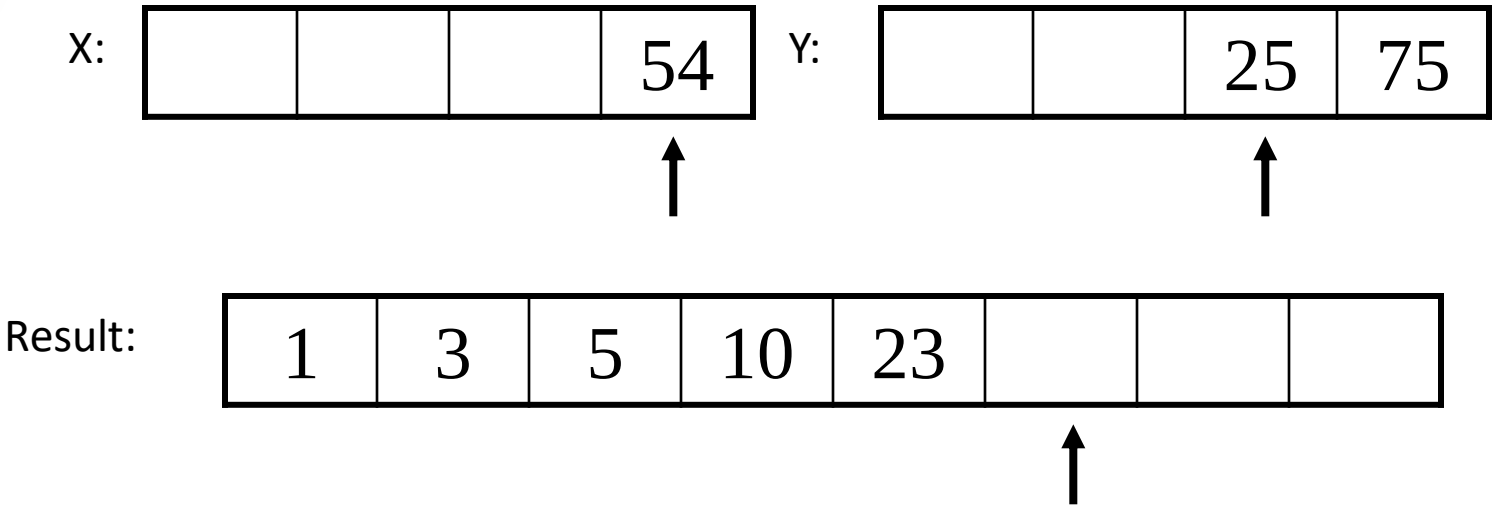
Merging (cont.)



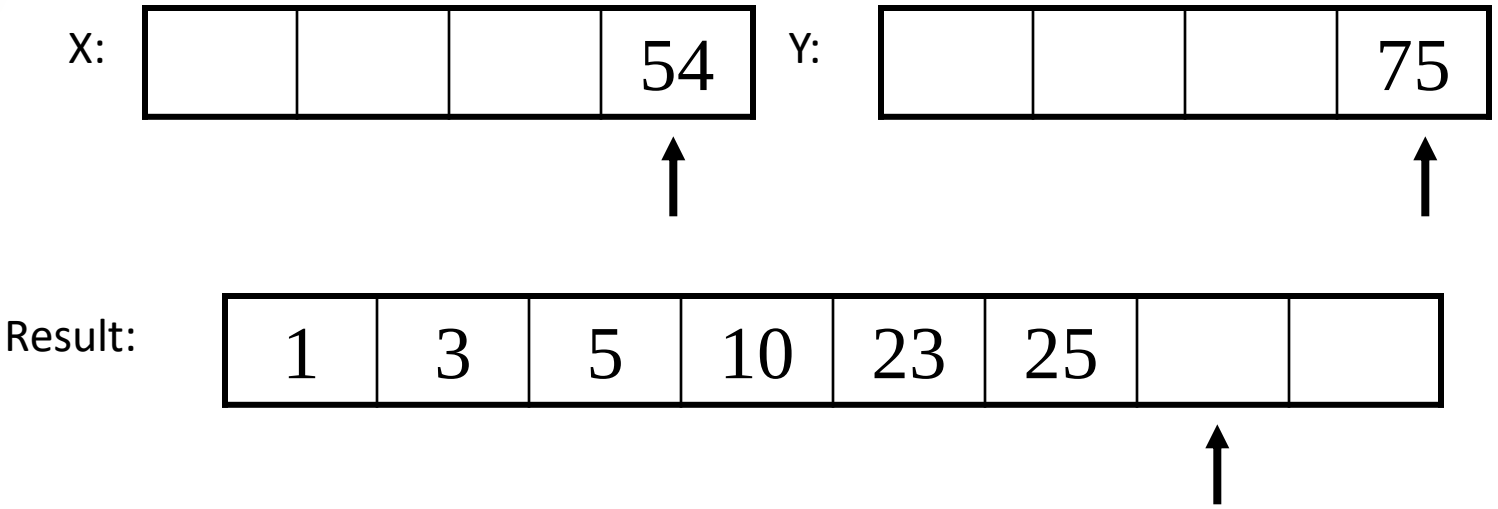
Merging (cont.)



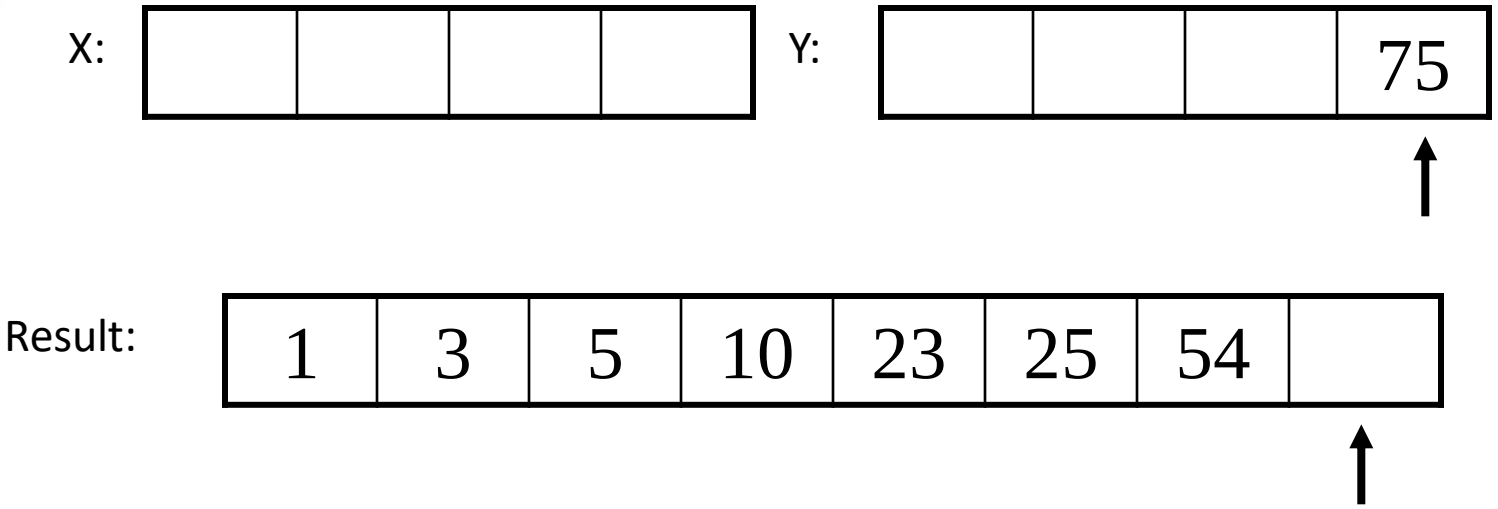
Merging (cont.)



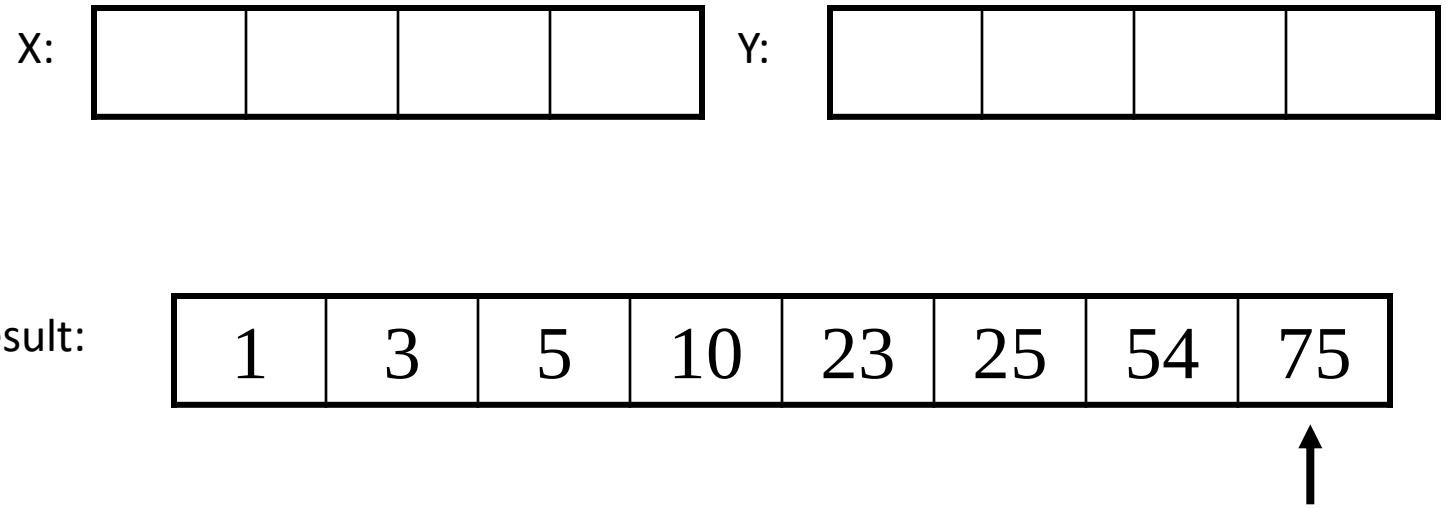
Merging (cont.)



Merging (cont.)



Merging (cont.)



Divide And Conquer

- Merging a two lists of one element each is the same as sorting them.
- Merge sort divides up an unsorted list until the above condition is met and then sorts the divided parts back together in pairs.
- Specifically this can be done by recursively dividing the unsorted list in half, merge sorting the right side then the left side and then merging the right and left back together.

Merge Sort Algorithm

Given a list L with a length k:

- If k == 1 → the list is sorted
- Else:
  - Merge Sort the left side (1 thru k/2)
  - Merge Sort the right side (k/2+1 thru k)
  - Merge the right side with the left side

Merge Sort Example

99	6	86	15	58	35	86	4	0

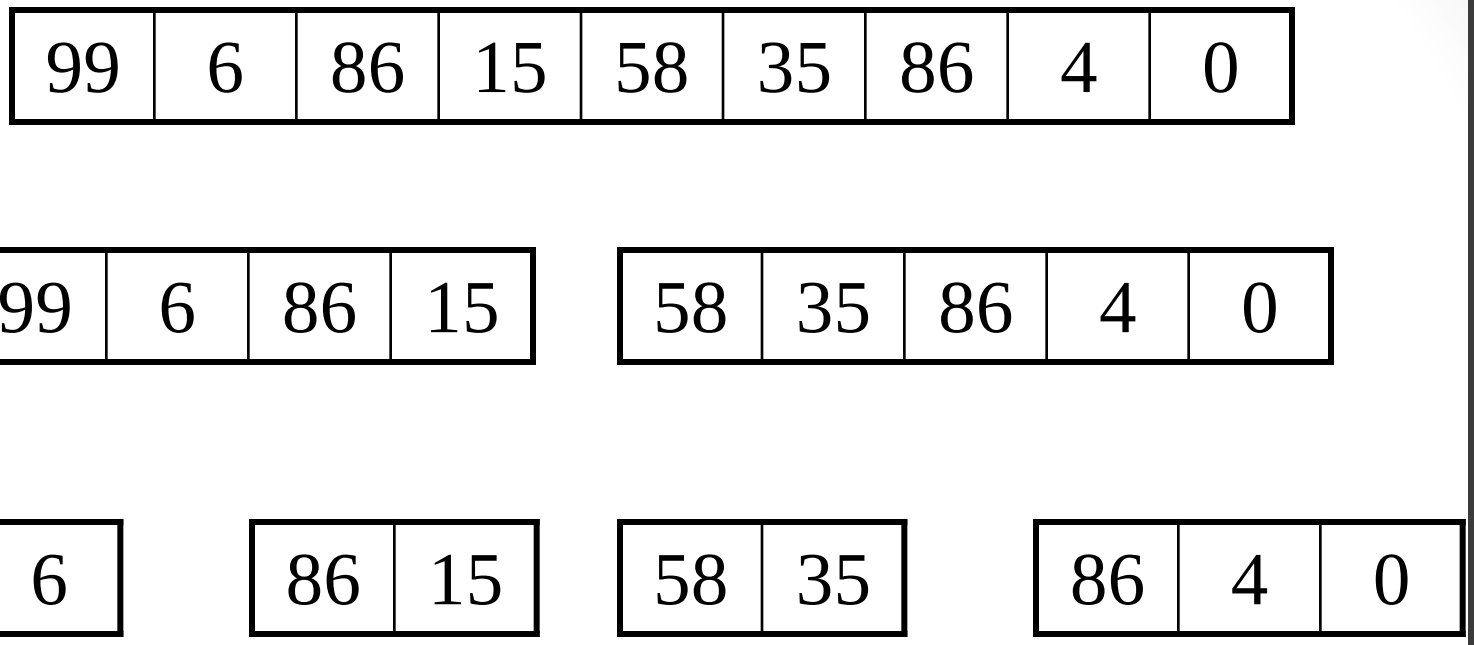
Merge Sort Example

99	6	86	15	58	35	86	4	0

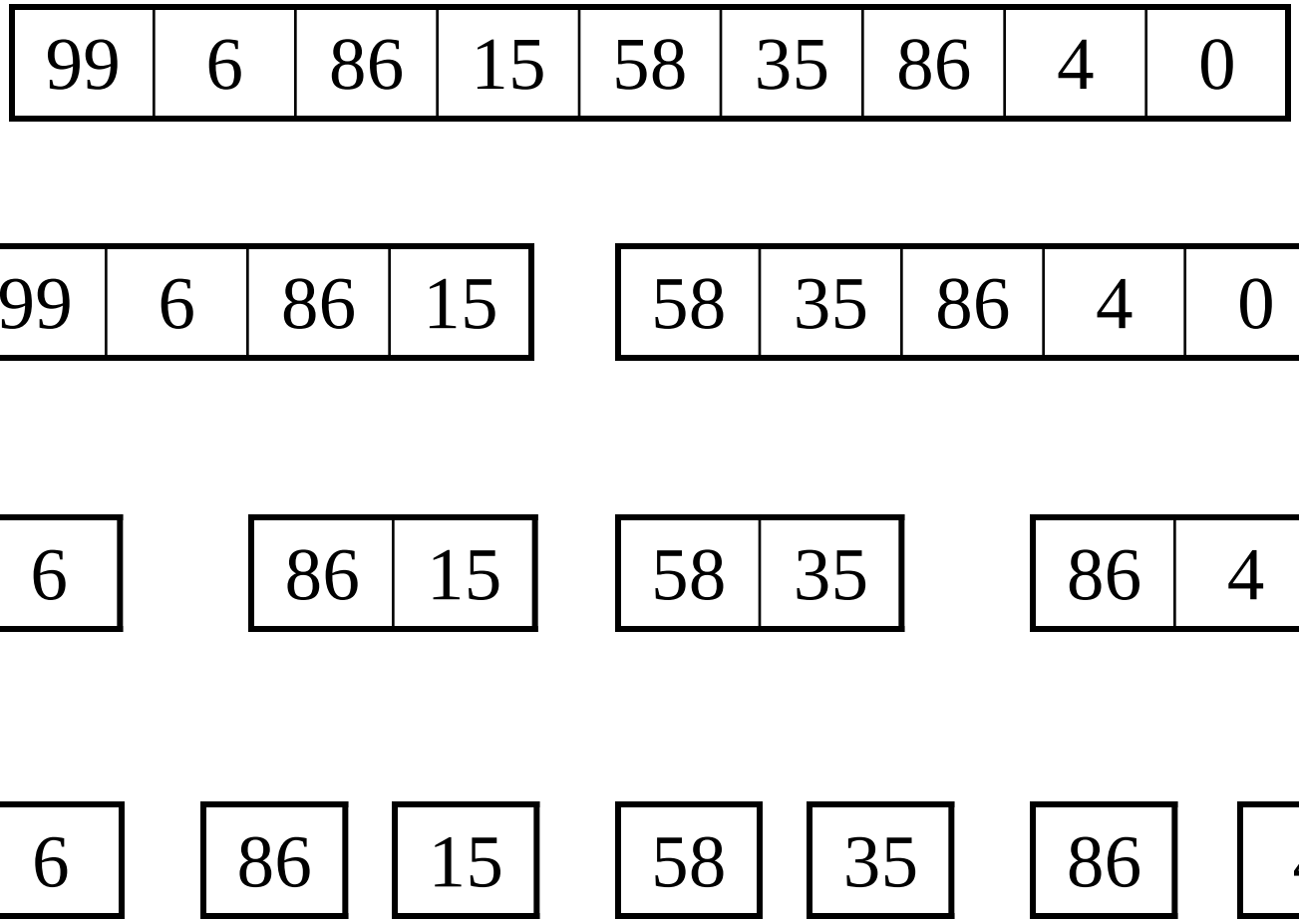
99	6	86	15

58	35	86	4	0

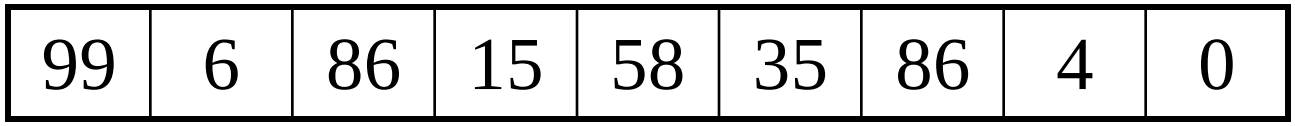
Merge Sort Example

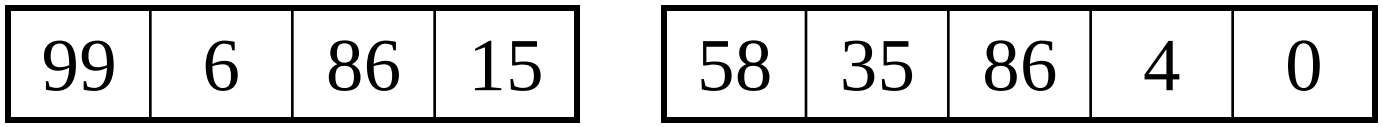


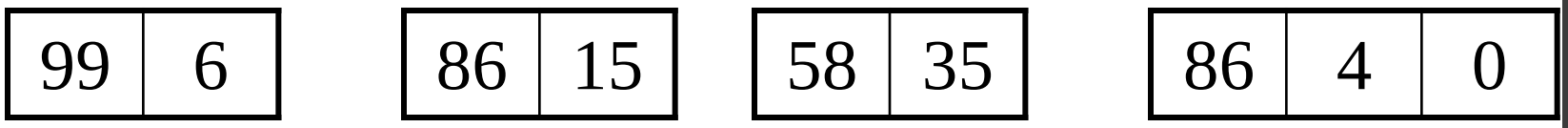
Merge Sort Example

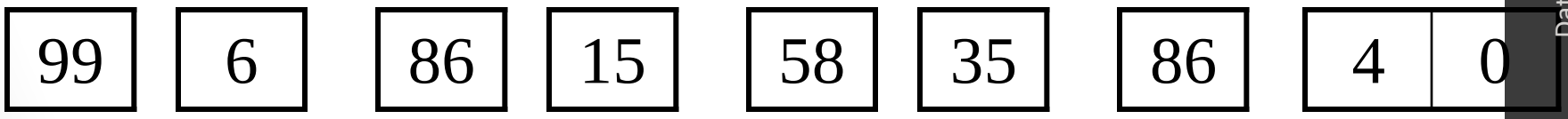


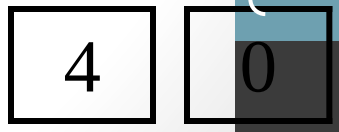
Merge Sort Example



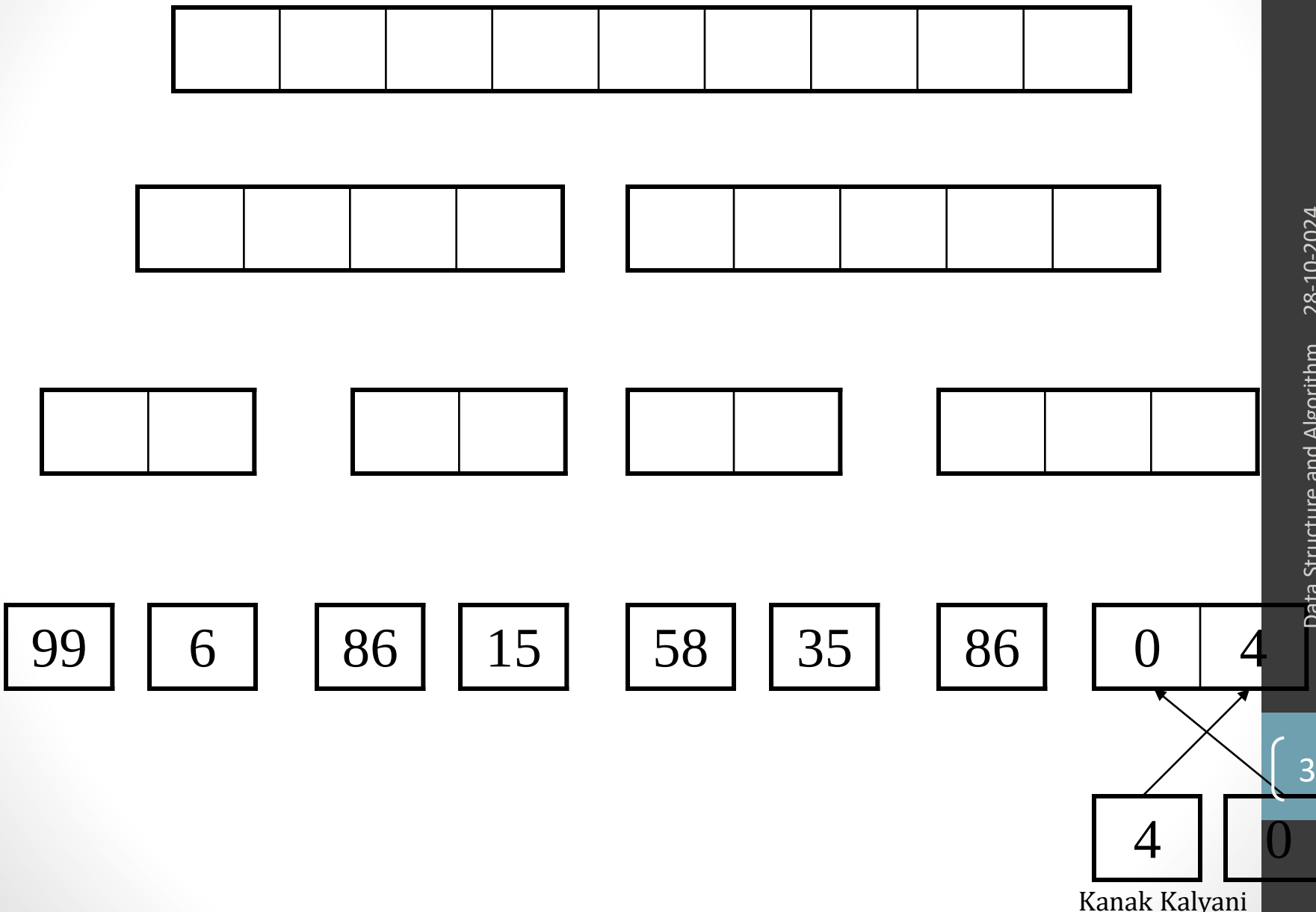




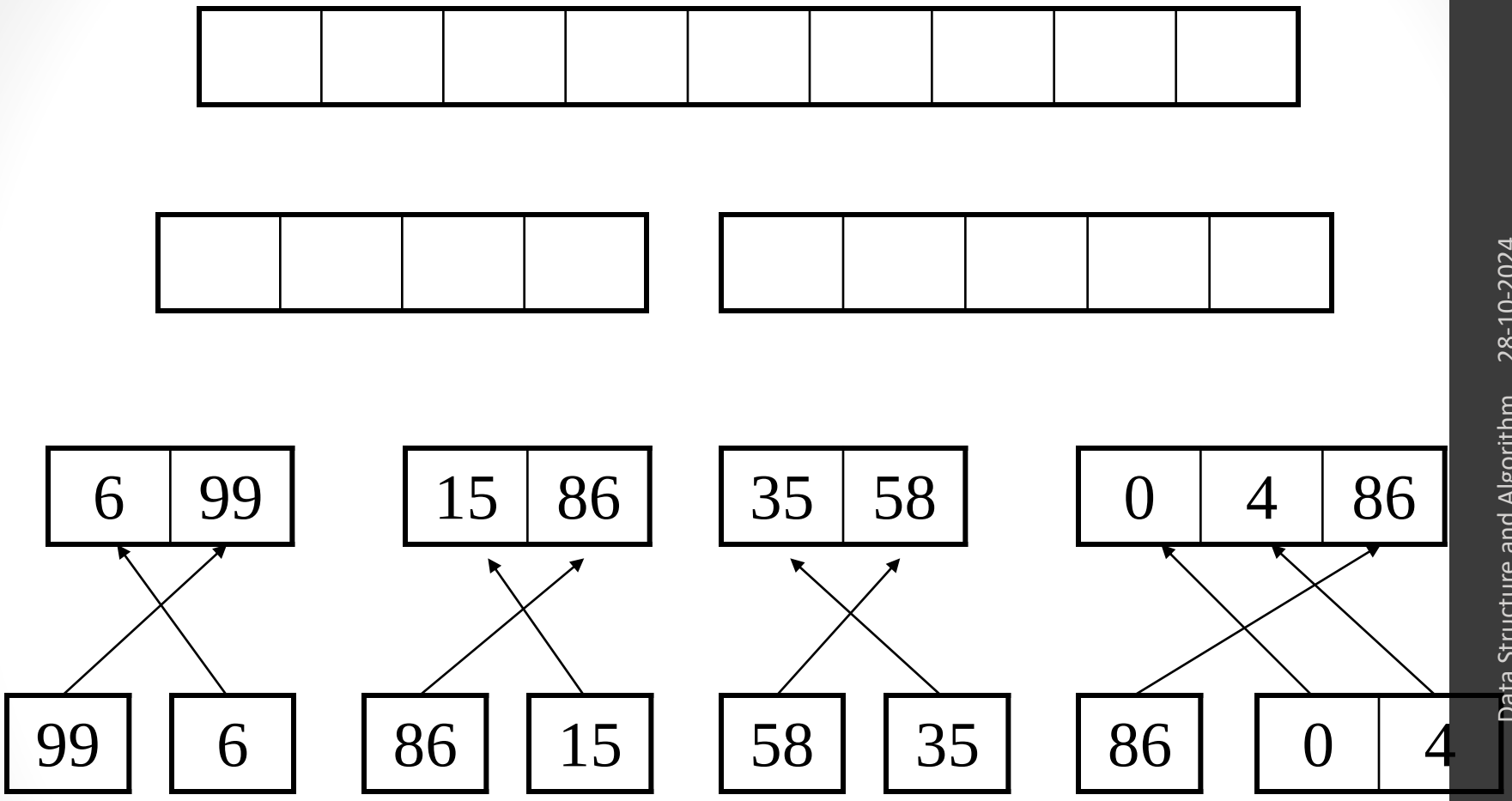




Merge Sort Example

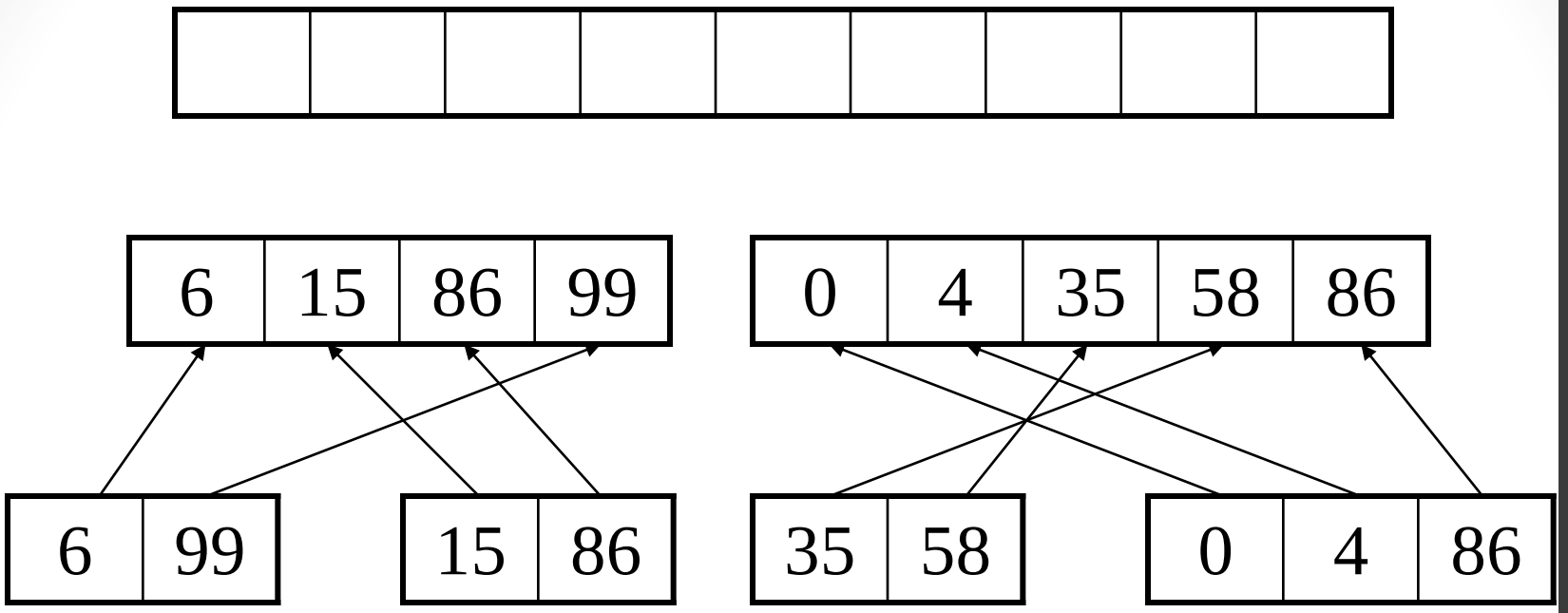


Merge Sort Example



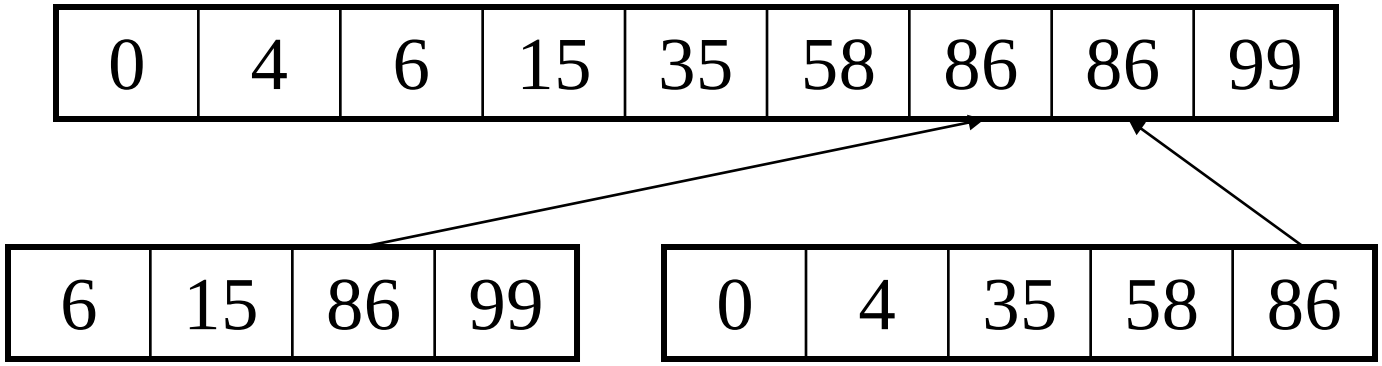
Merge

Merge Sort Example



Merge

Merge Sort Example



Merge

Merge Sort Example

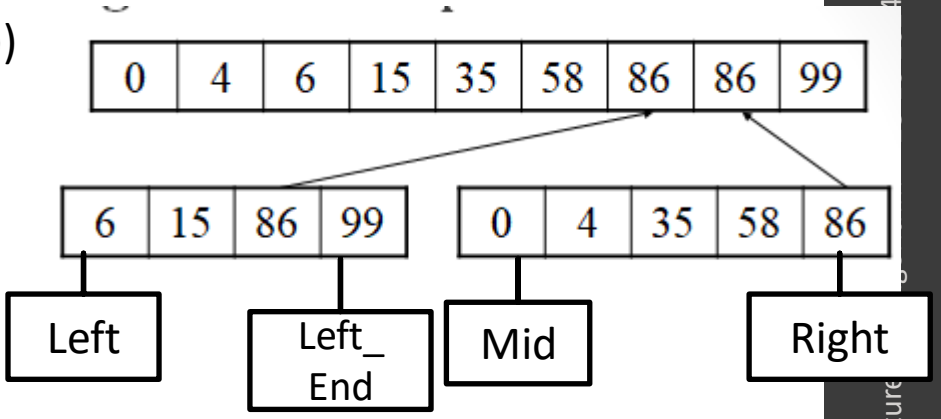
0	4	6	15	35	58	86	86	99

Merge Sort Code

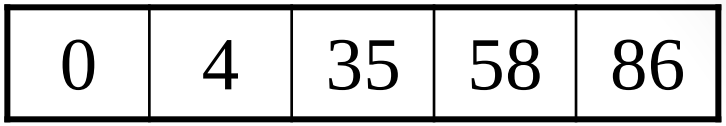
```
// Dividing the array recursively
void m_sort(int numbers[], int temp[], int left, int right)
{
      int mid;
      if (right > left)
      {
         mid = (right + left) / 2;
         m_sort(numbers, temp, left, mid);
         m_sort(numbers, temp, mid+1, right);
         merge(numbers, temp, left, mid+1, right);
      }
```

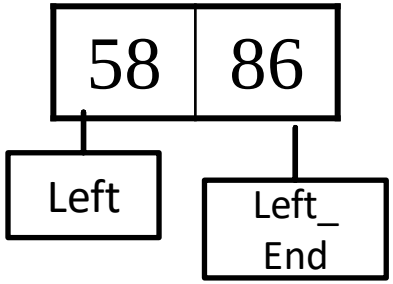
Merge Sort Code

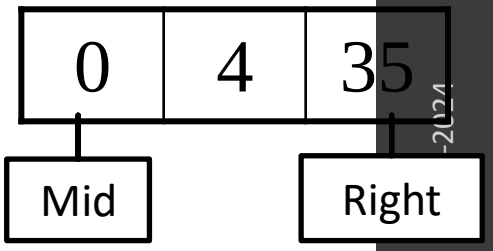
```
void merge(int numbers[], int temp[], int left, int mid, int right)
{       int i, left_end, num_elements, tmp_pos;
        left_end = mid - 1;          tmp_pos = left;
        num_elements = right - left + 1;
      while ((left <= left_end) && (mid <= right))
     {
          if (numbers[left] <= numbers[mid])
          {
               temp[tmp_pos] = numbers[left];
               tmp_pos = tmp_pos + 1;
               left = left +1;
          }
          else
          {
               temp[tmp_pos] = numbers[mid];
               tmp_pos = tmp_pos + 1;
               mid = mid + 1;
          }
}
```



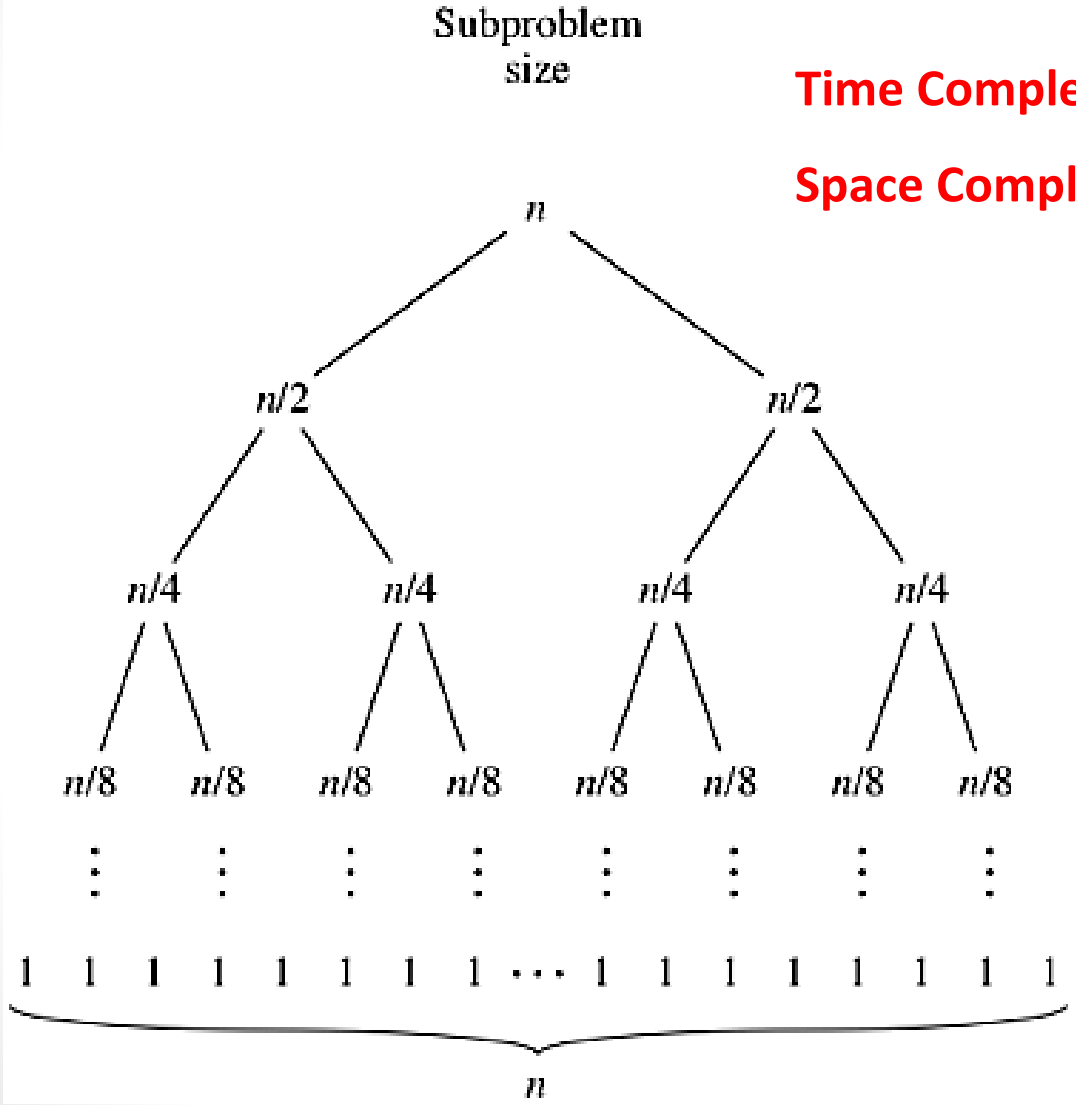
```
while (left <= left_end)
    {
        temp[tmp_pos] = numbers[left];
        left = left + 1;
        tmp_pos = tmp_pos + 1;
    }
while (mid <= right)
    {
        temp[tmp_pos] = numbers[mid];
        mid = mid + 1;
        tmp_pos = tmp_pos + 1;
    }
    for (i = 0; i <= num_elements; i++)
    {
        numbers[right] = temp[right];
        right = right - 1;
    }
}
```







Merge Sort Analysis



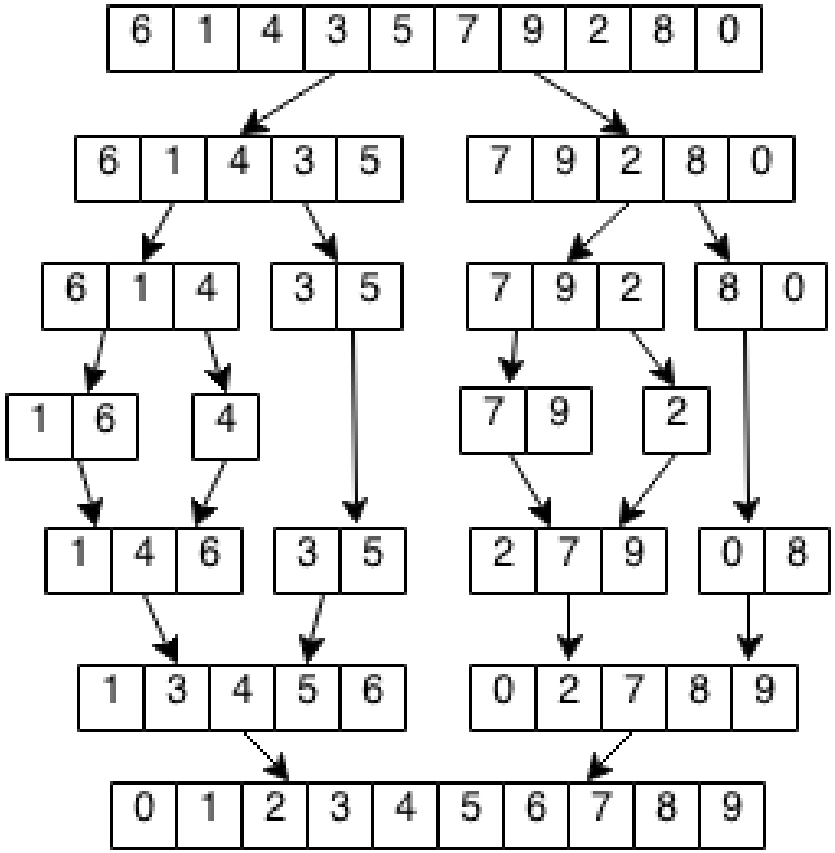
Time Complexity =O(nlog$_2$n)

Space Complexity:- O(n)

Do it Yourself

Sort the array by merge sort

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


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

Quicksort Algorithm

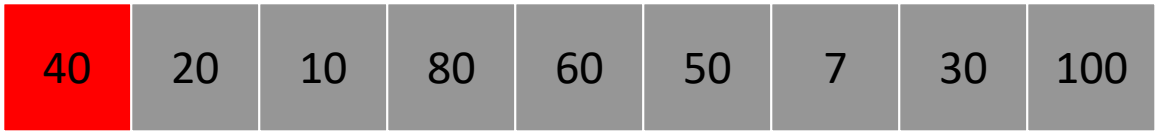
Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

Example

40	20	10	80	60	50	7	30	100

Pick Pivot Element



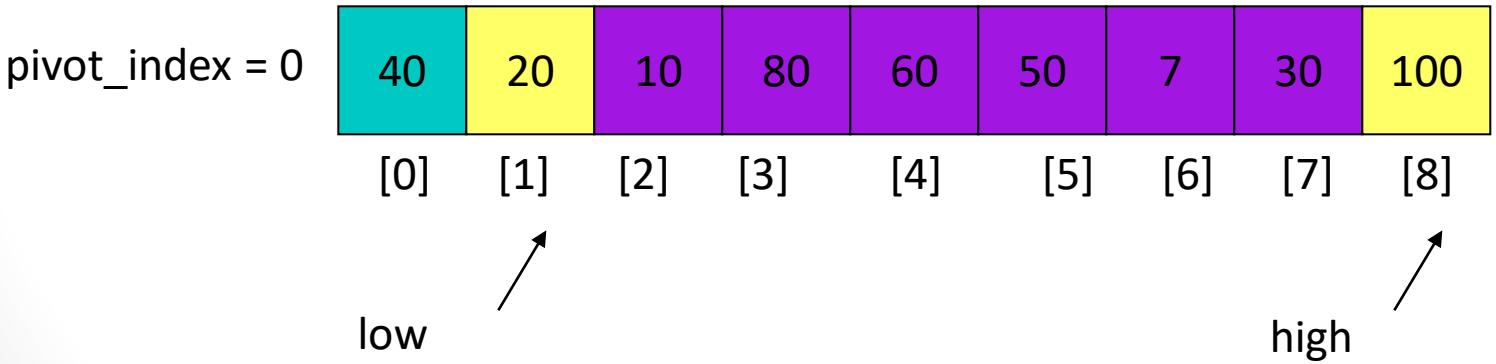
Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

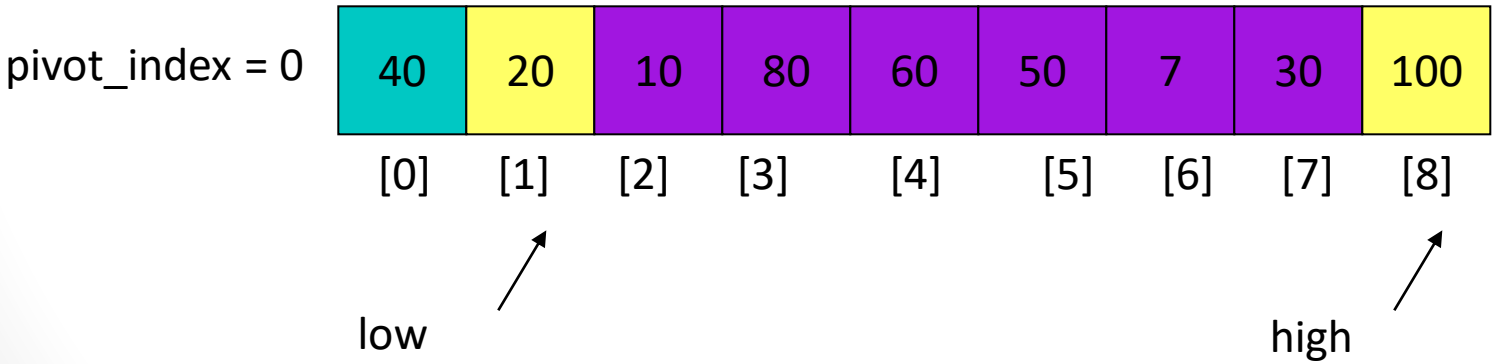
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

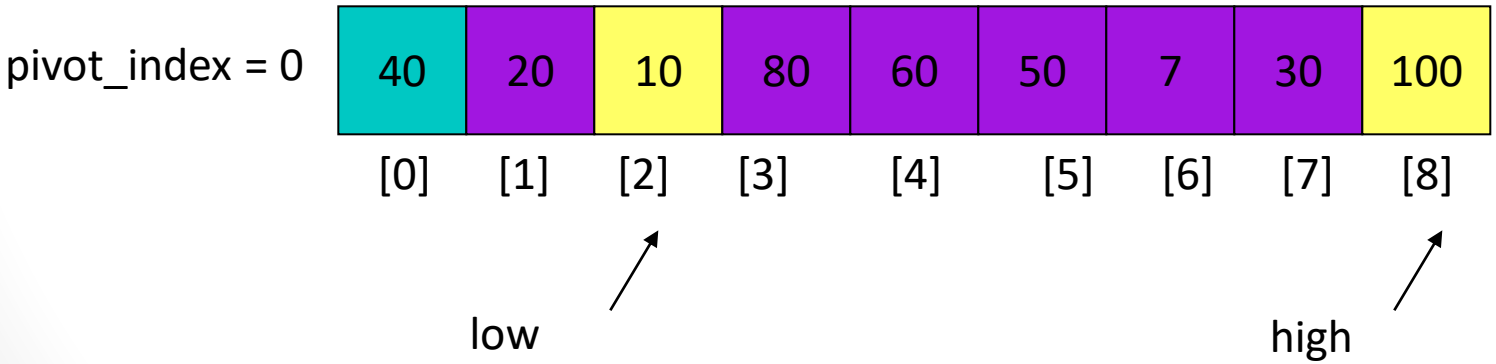
Partitioning loops through, swapping elements below/above pivot.


pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
low
high

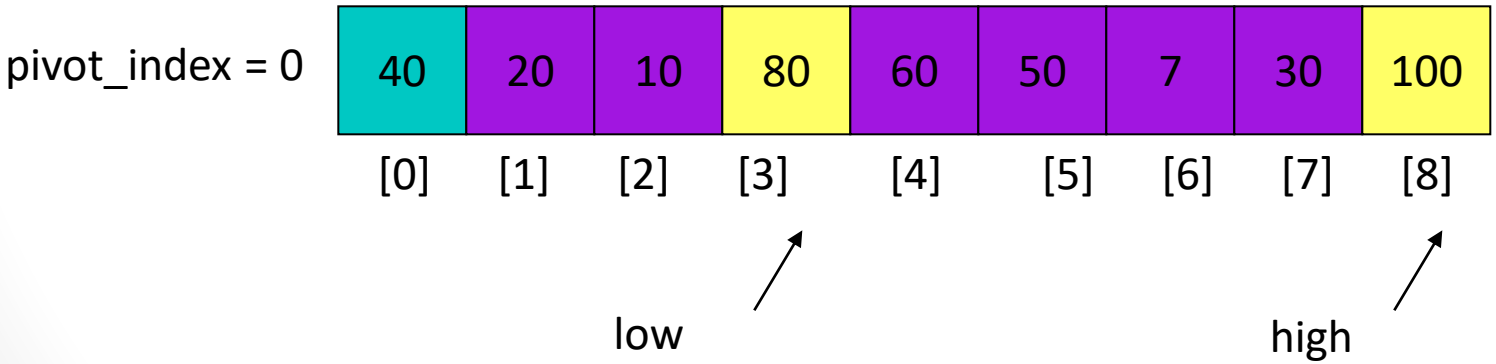
1. While data[low] <= data[pivot]
   ++low



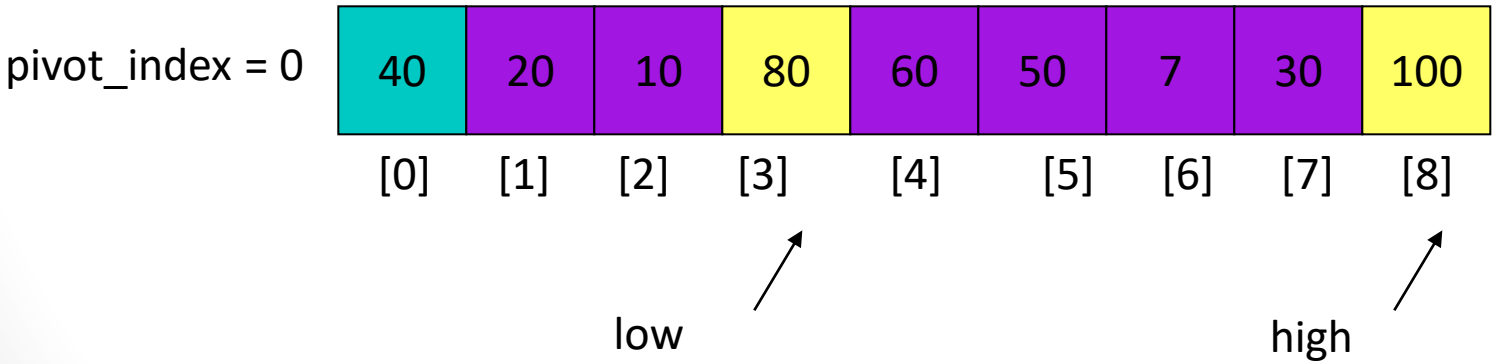
1. While data[low] <= data[pivot]
   ++low



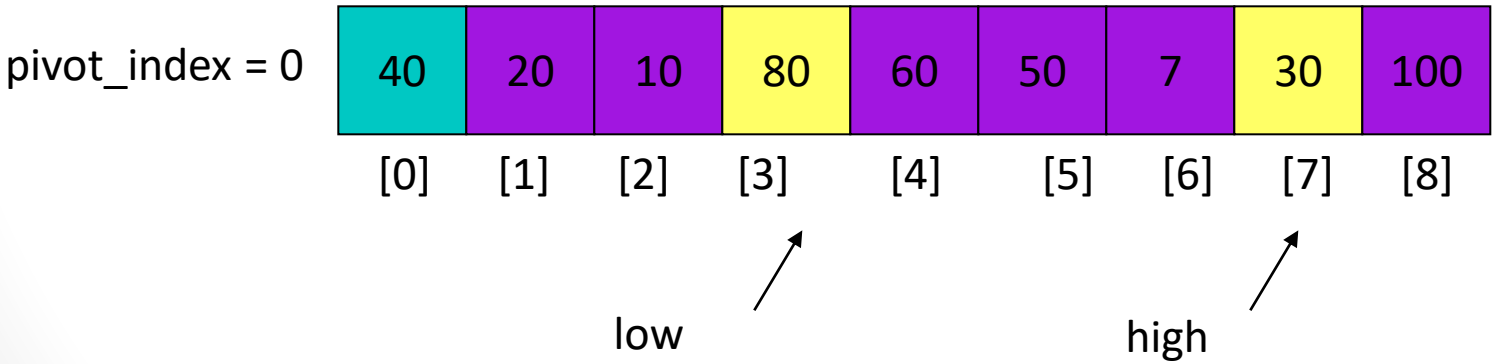
1. While data[low] <= data[pivot]
   ++low



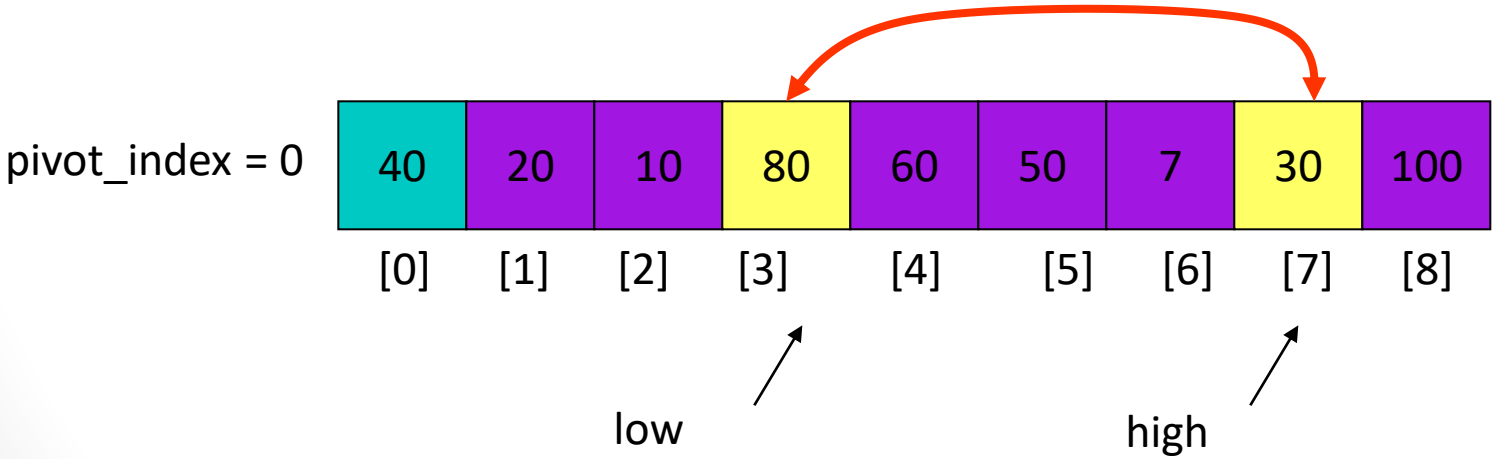
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high



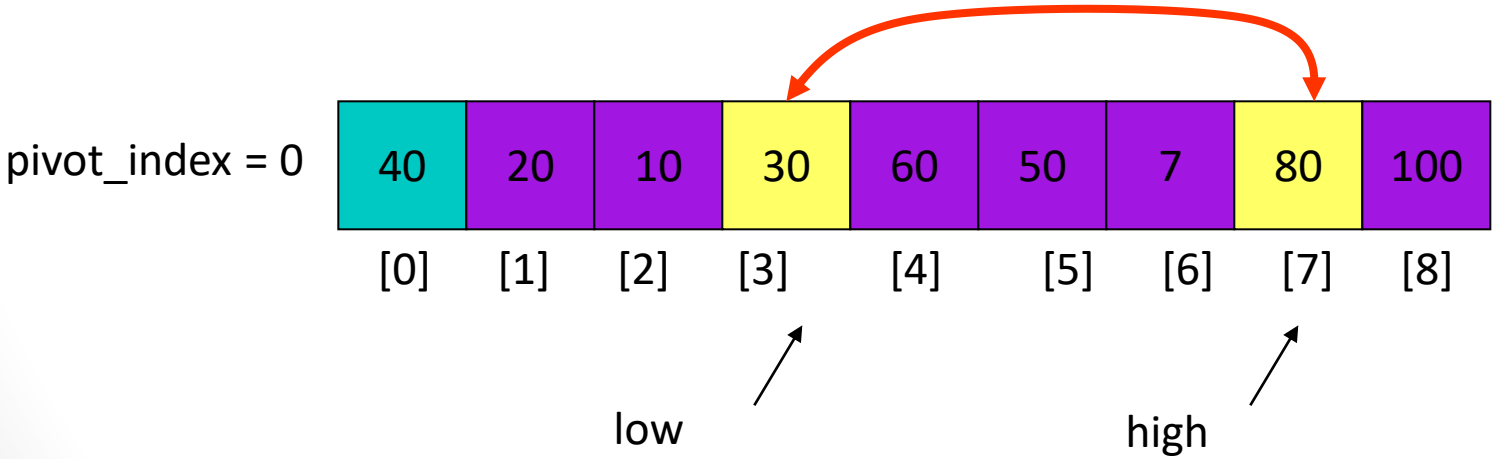
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high



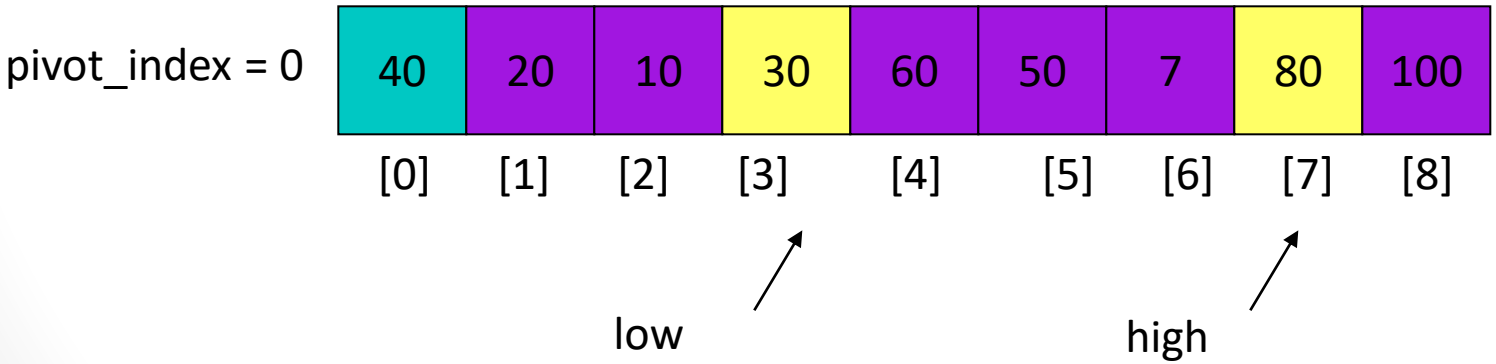
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]



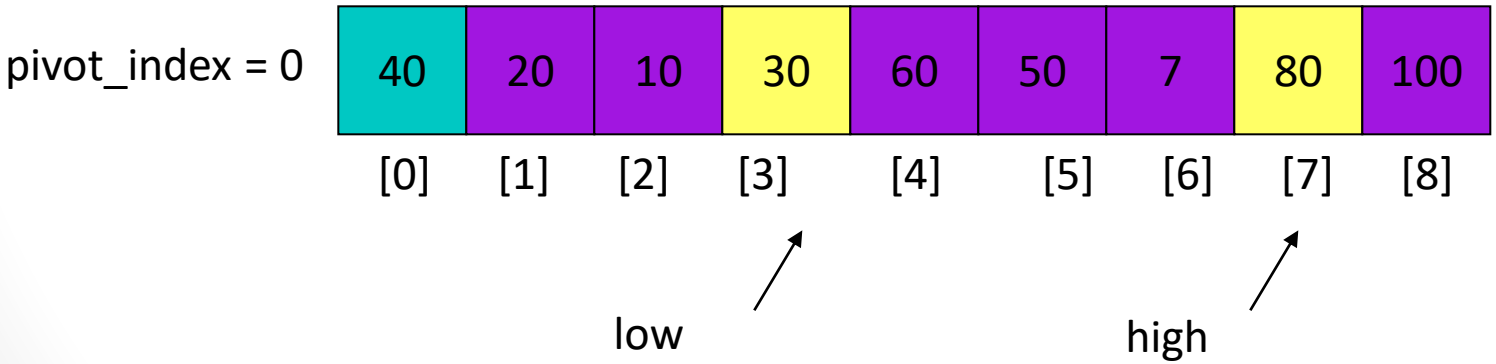
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]



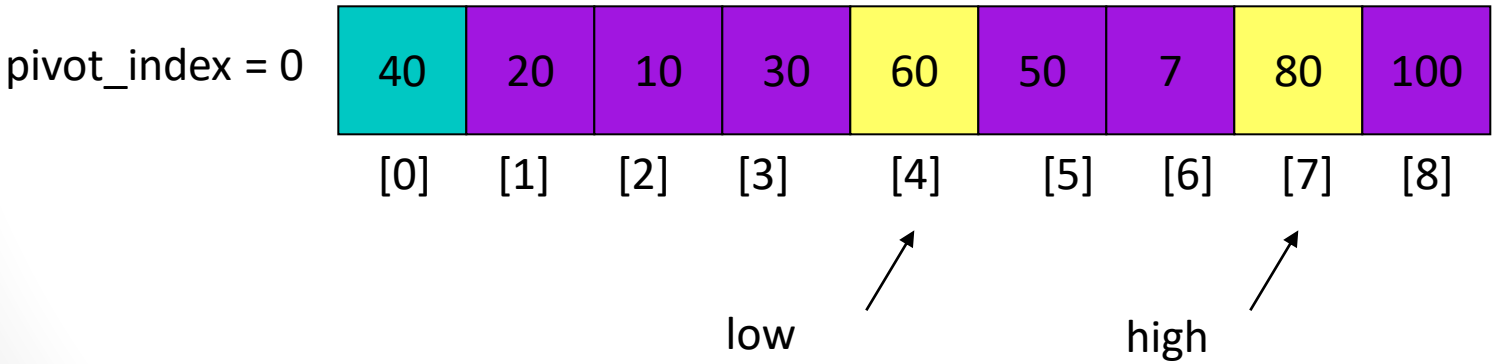
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



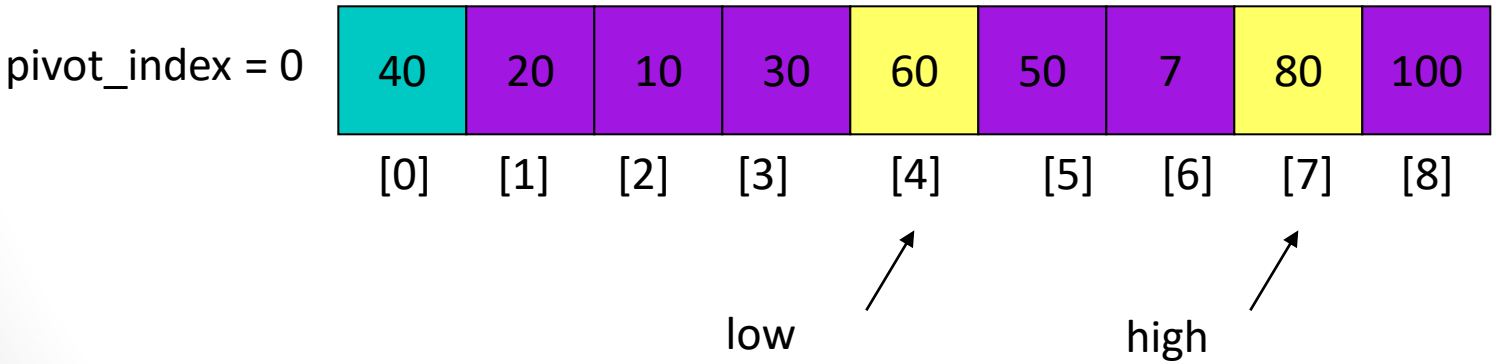
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



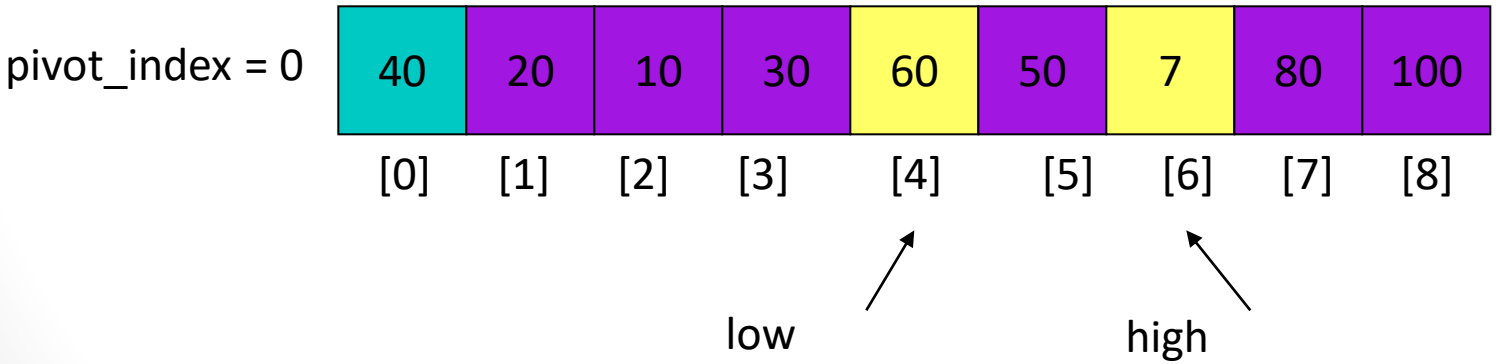
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



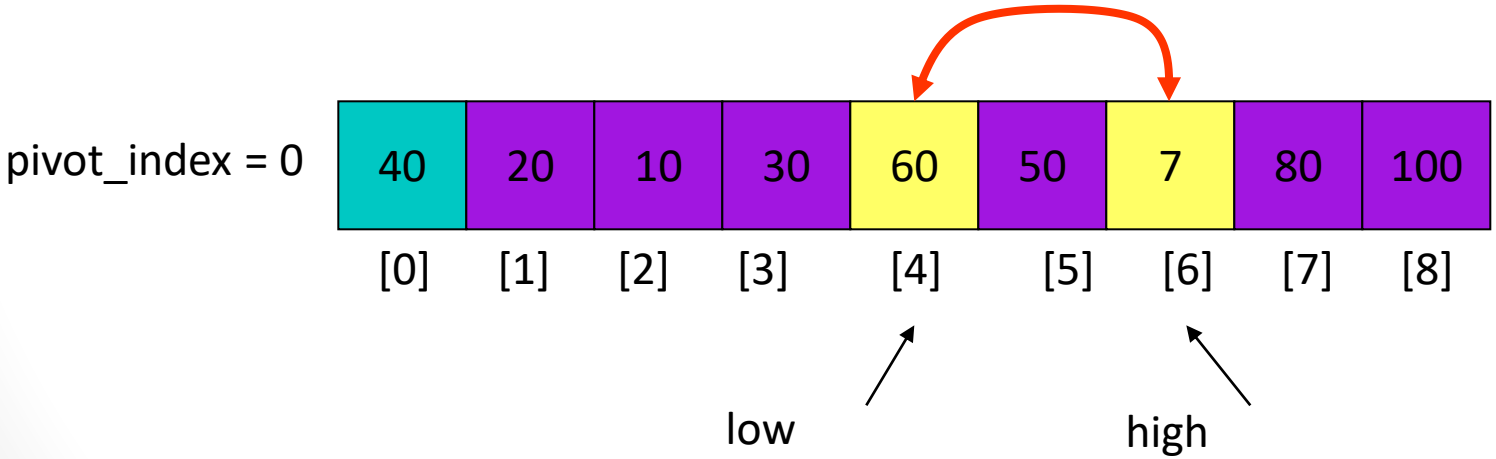
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



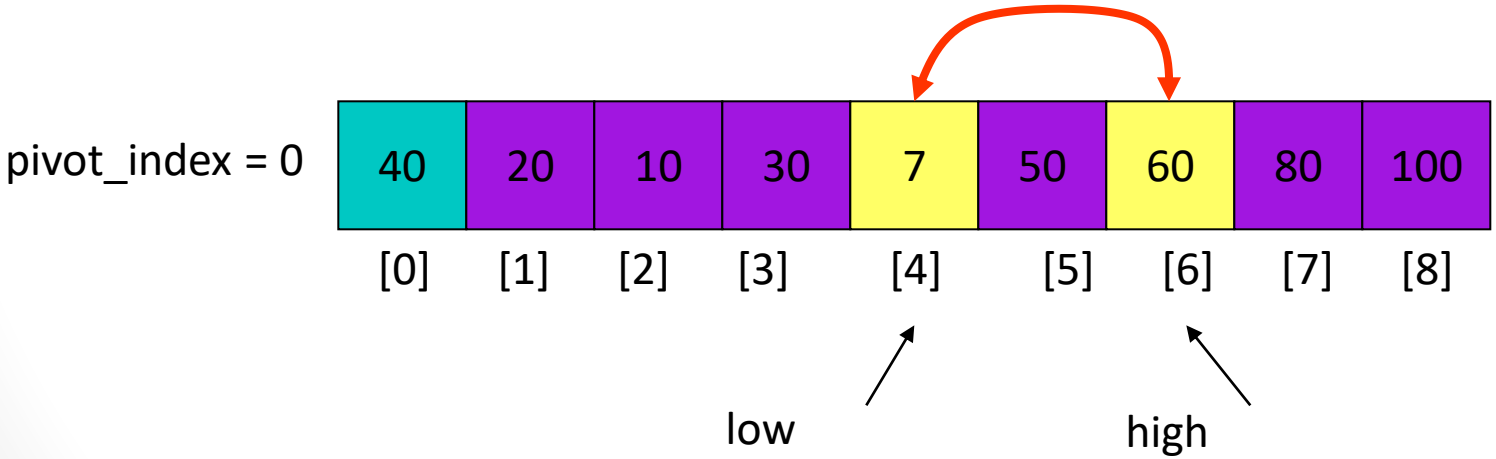
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



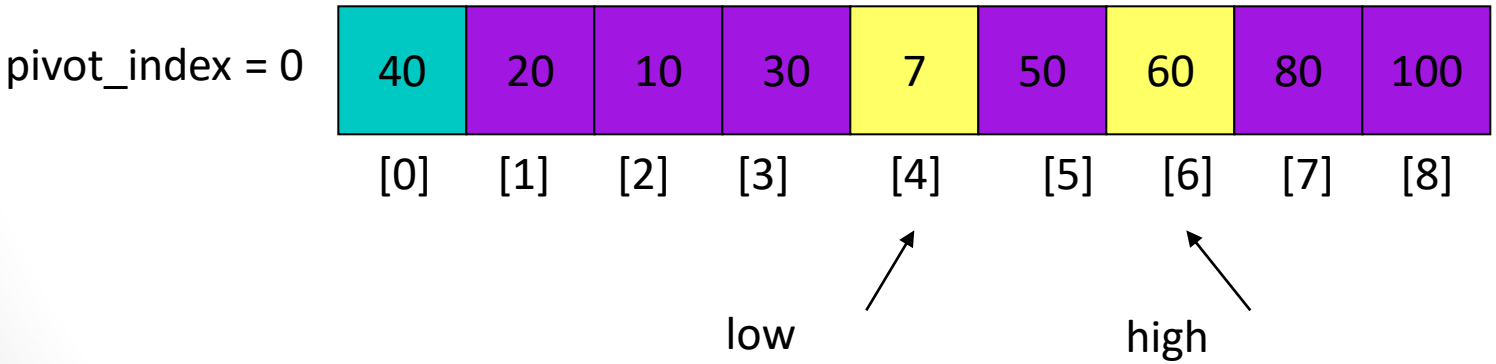
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



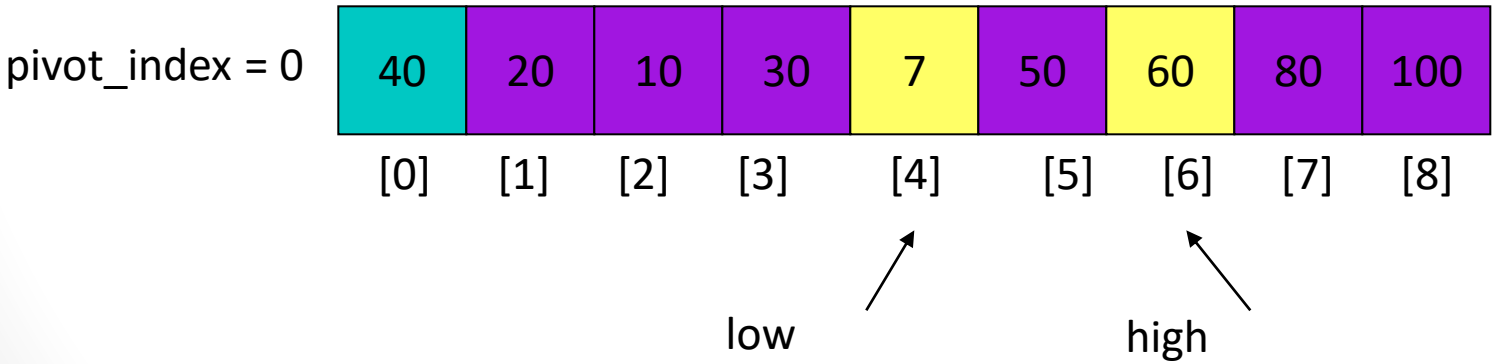
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



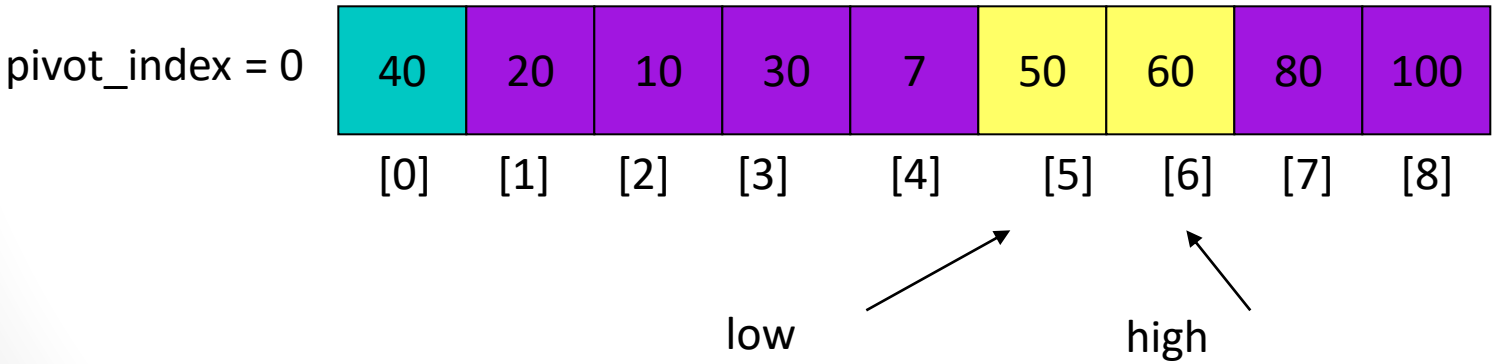
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



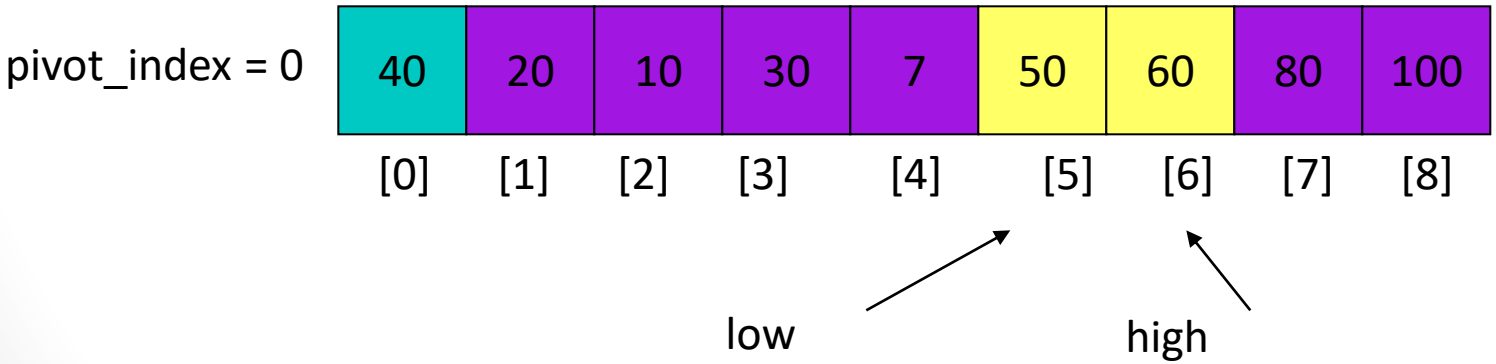
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



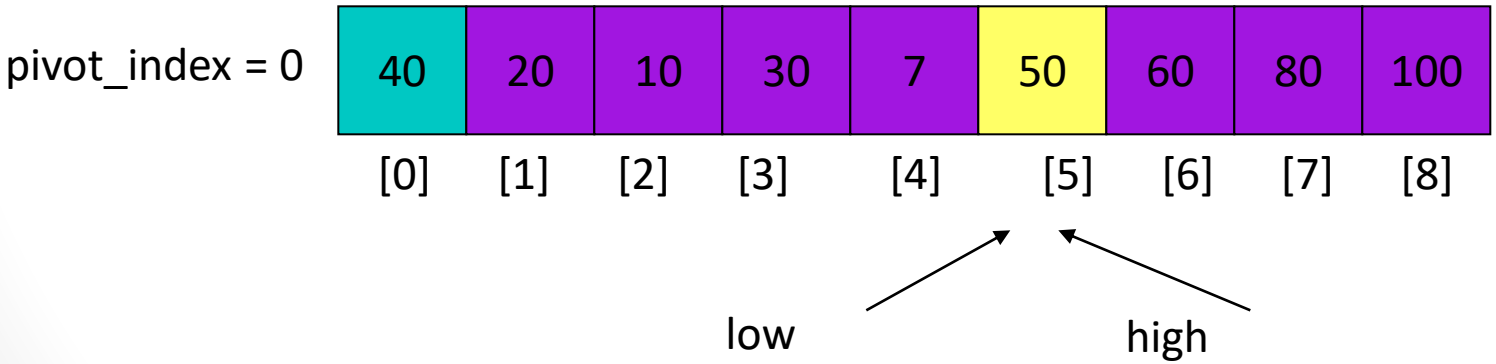
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



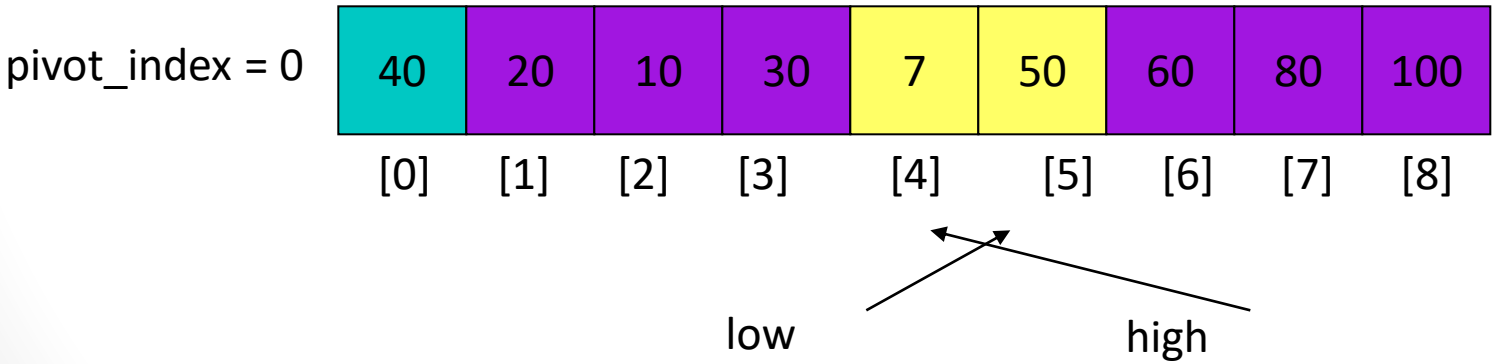
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



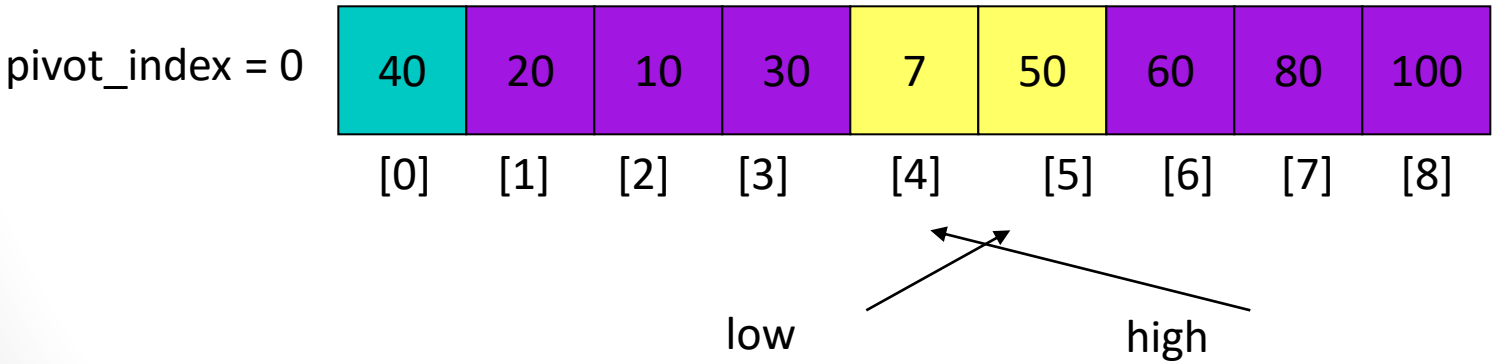
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



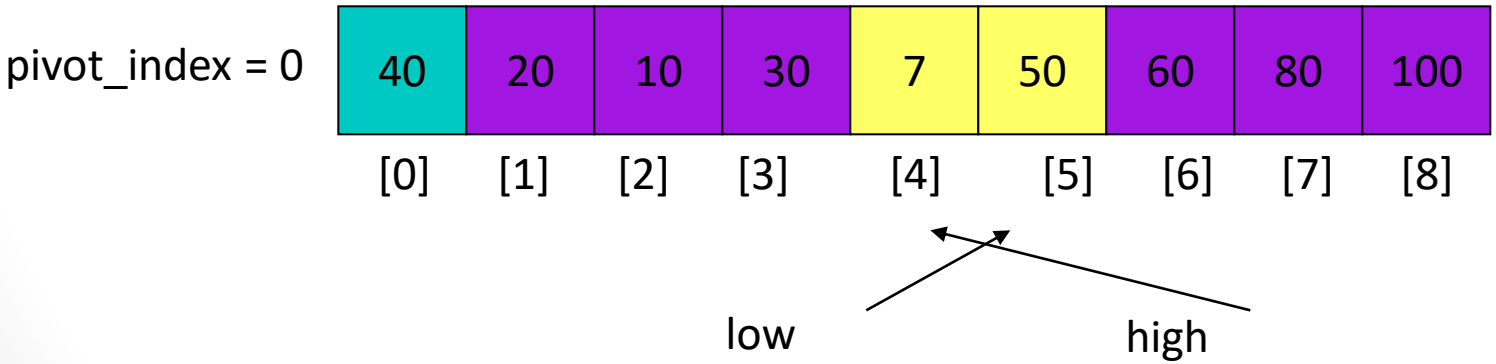
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



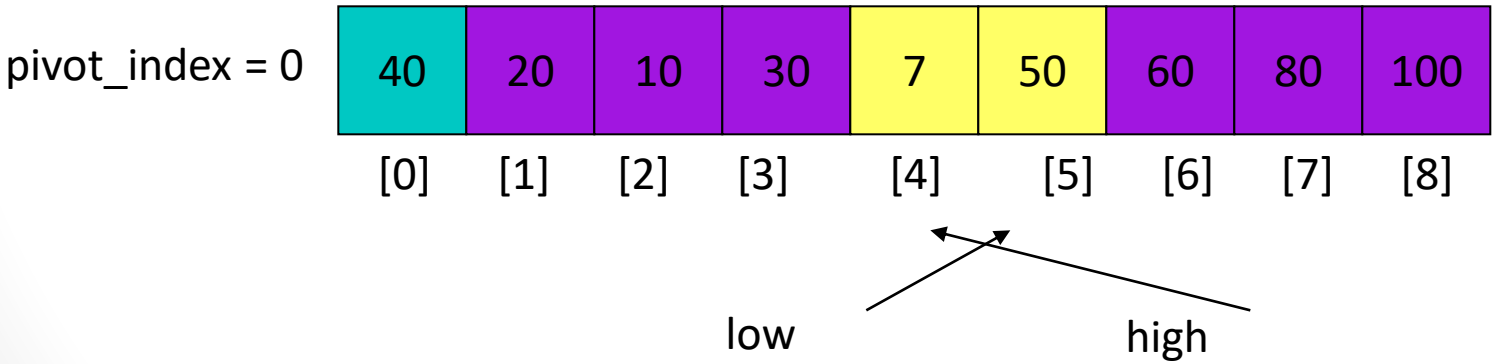
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



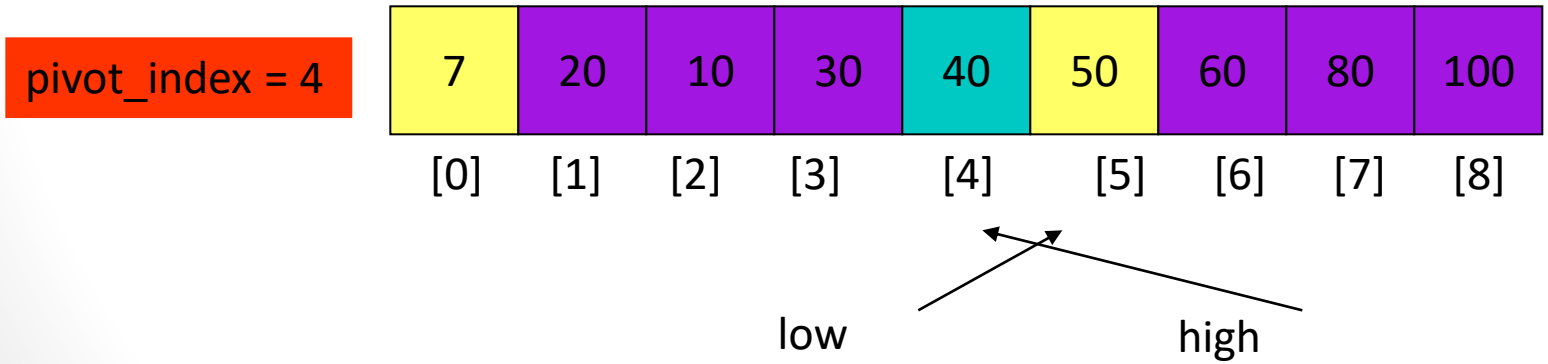
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.



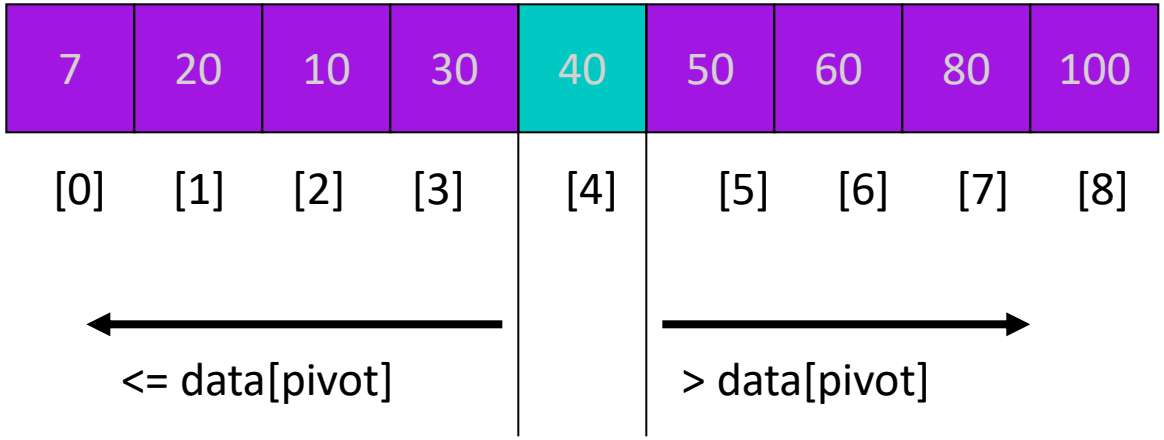
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.
5. Swap data[high] and data[pivot_index]



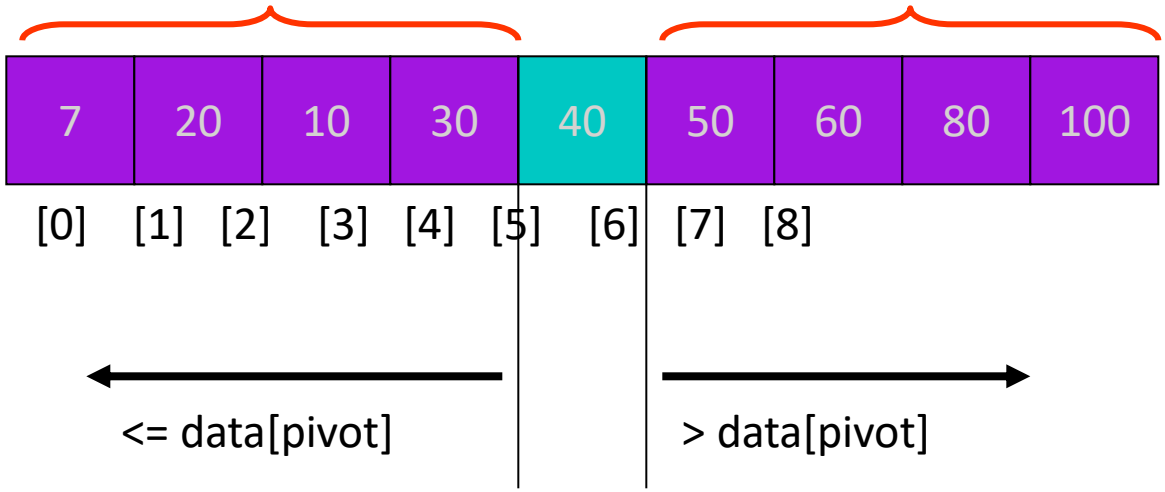
1. While data[low] <= data[pivot]
   ++low
2. While data[high] > data[pivot]
   --high
3. If low < high
   swap data[low] and data[high]
4. While high > low, go to 1.
5. Swap data[high] and data[pivot_index]



Partition Result



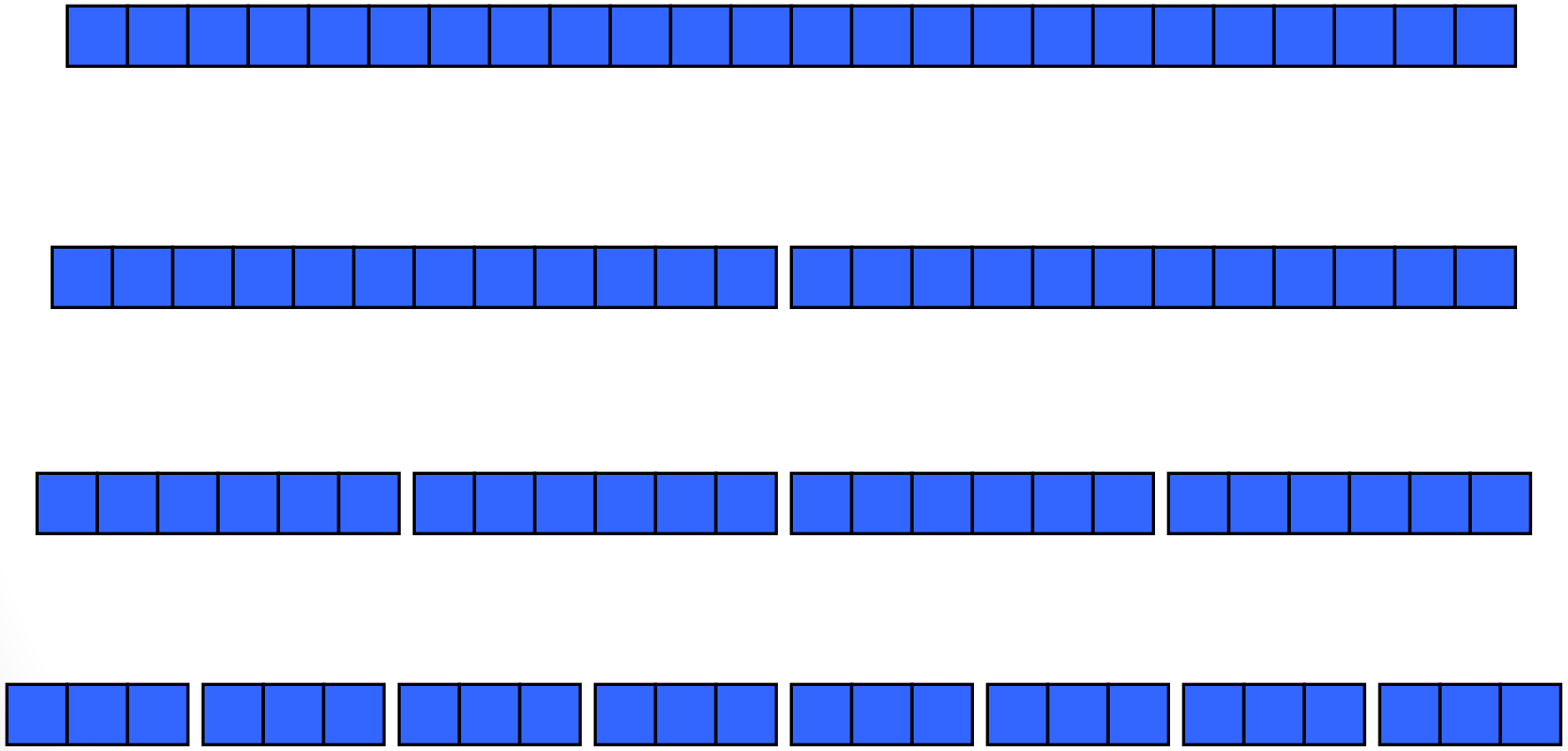
Recursion: Quicksort Sub-arrays



Analysis of quicksort—best case

- Suppose each partition operation divides the array almost exactly in half
- Then the depth of the recursion in $\log_2 n$
  - Because that's how many times we can halve n

Partitioning at various levels



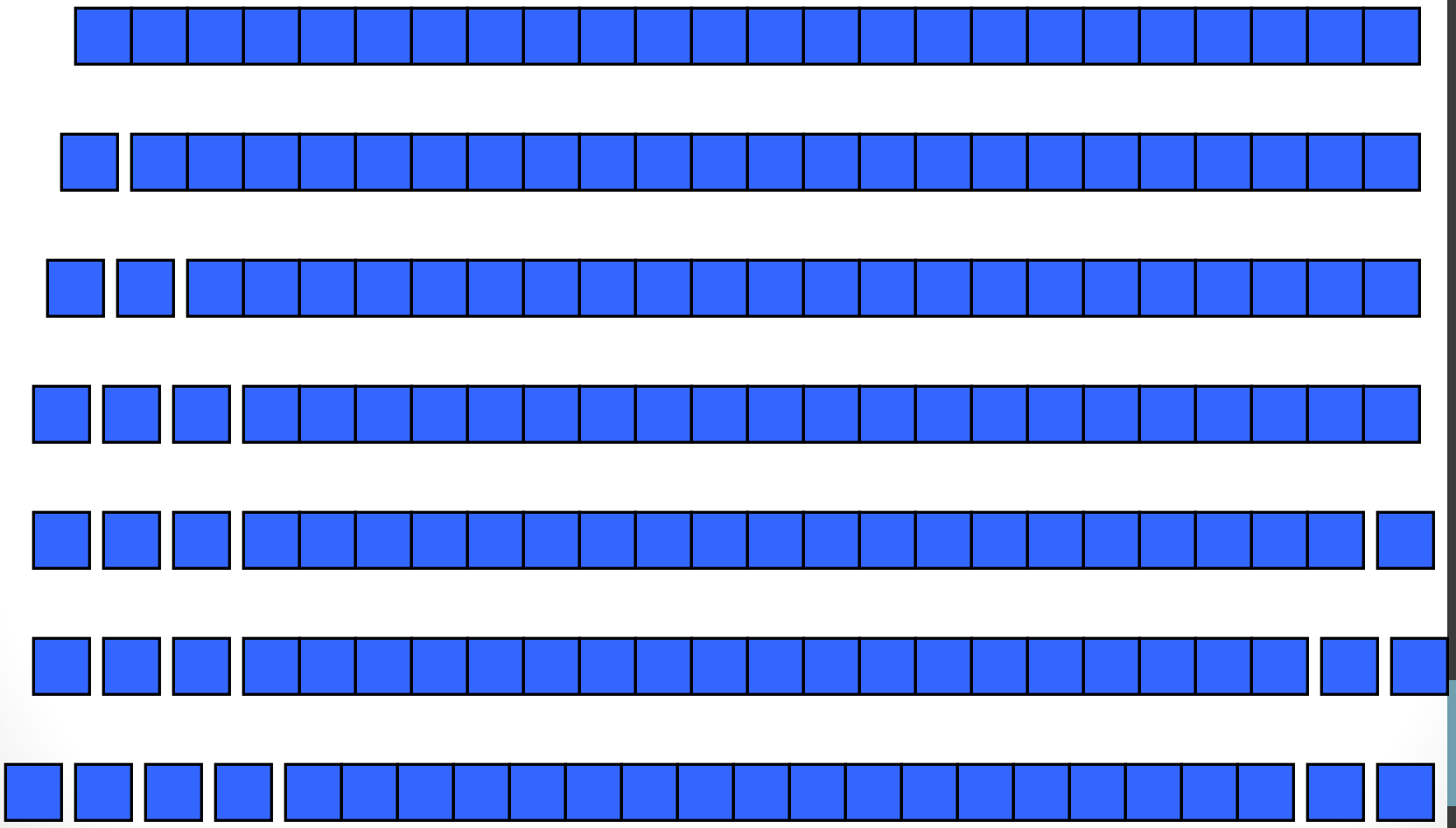
Best case II

- We cut the array size in half each time
- So the depth of the recursion in $\log_2 n$
- At each level of the recursion, all the partitions at that level do work that is linear in n
- $O(\log_2 n) * O(n) = O(n \log_2 n)$
- Hence in the average case, quicksort has time complexity $O(n \log_2 n)$
- What about the worst case?

Worst case

- In the worst case, partitioning always divides the size n array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length n-1 part, containing everything else
- We don't recur on the zero-length part
- Recurring on the length n-1 part requires (in the worst case) recurring to depth n-1

Worst case partitioning

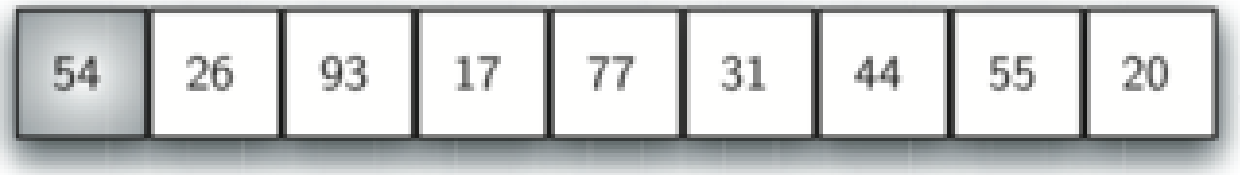


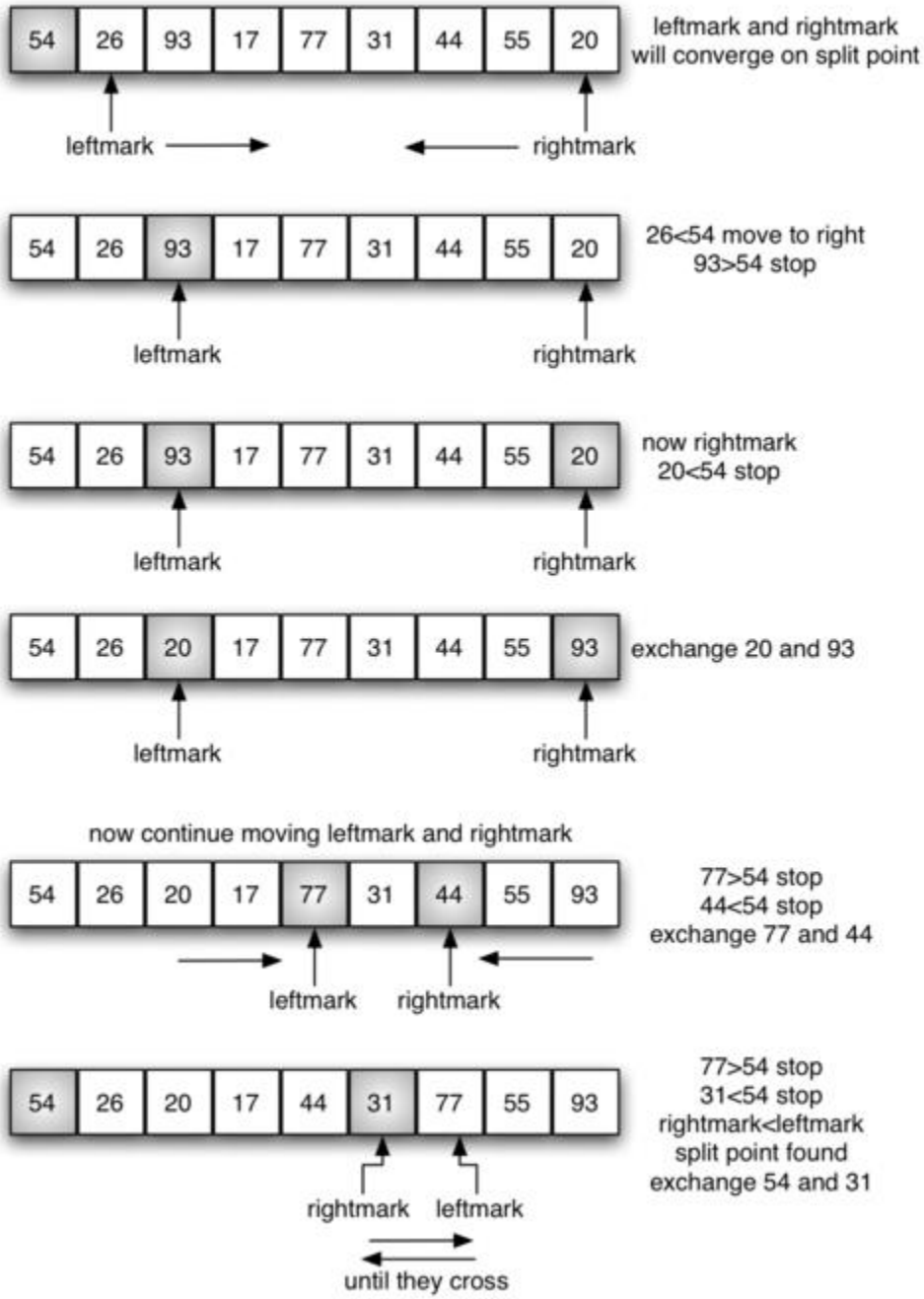
Worst case for quicksort

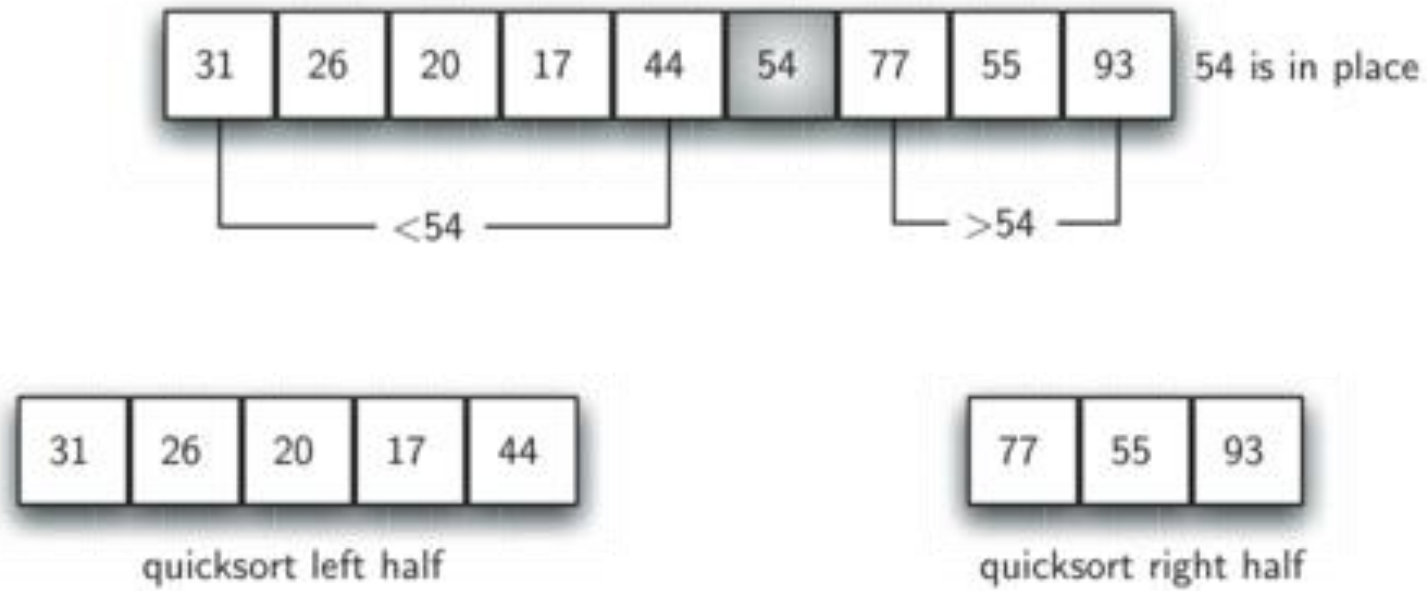
- In the worst case, recursion may be n levels deep (for an array of size n)
- But the partitioning work done at each level is still n
- $O(n) * O(n) = O(n^2)$
- So worst case for Quicksort is $O(n^2)$
- When does this happen?
  - When the array is sorted to begin with!

Do it Yourself

- Sort the Number using Quicksort




54 26 93 17 77 31 44 55 20
leftmark and rightmark will converge on split point
leftmark
rightmark
54 26 93 17 77 31 44 55 20
26<54 move to right
93>54 stop
leftmark
rightmark
54 26 93 17 77 31 44 55 20
now rightmark
20<54 stop
leftmark
rightmark
54 26 20 17 77 31 44 55 93
exchange 20 and 93
leftmark
rightmark
now continue moving leftmark and rightmark
54 26 20 17 77 31 44 55 93
77>54 stop
44<54 stop
exchange 77 and 44
leftmark
rightmark
54 26 20 17 44 31 77 55 93
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31
rightmark
leftmark
until they cross


31
26
20
17
44
54
77
55
93
54 is in place
<54
>54
31
26
20
17
44
quicksort left half
77
55
93
quicksort right half

```
void quicksort( int a[], int low, int high )
{
 int pivot,pivot_item,i,j,t;
 if(high<=low)
        return;
 if(high > low)
 {
        pivot = i = low;
        pivot_item = a[low];
        j = high;
        while ( i < j )
        {
                while( a[i] <= pivot_item && i<=high )
                        i++;
                while( a[j] > pivot_item && j>=low)
                        j--;
```

```
if ( i < j )
                {
                        t=a[i];
                        a[i]=a[j];
                        a[j]=t;
                }
        }
 }
//position for pivot found
 a[pivot] = a[j];
 a[j] = pivot_item;
 quicksort( a, low, j-1 );
 quicksort( a, j+1, high );
}
```

Heap Sort

Heapsort

- It is a well-known, traditional sorting algorithm you will be expected to know
- Heapsort is *always* O(n log n)
  - Quicksort is usually O(n log n) but in the worst case slows to $O(n^2)$
  - Quicksort is generally faster, but Heapsort is better in time-critical applications

What is a “heap”?

Definitions of heap:

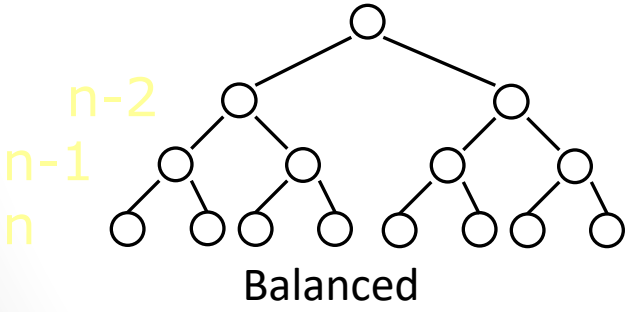
1. A large area of memory from which the programmer can allocate blocks as needed, and deallocate them (or allow them to be garbage collected) when no longer needed
2. A balanced, left-justified binary tree in which no node has a value greater than the value in its parent

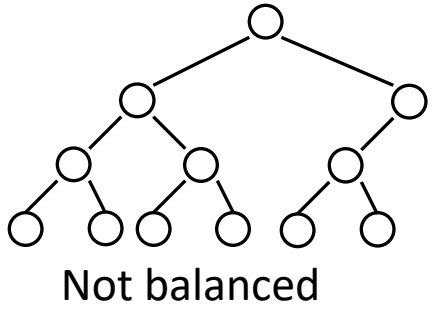
These two definitions have little in common

➔Heapsort uses the second definition

Balanced binary trees

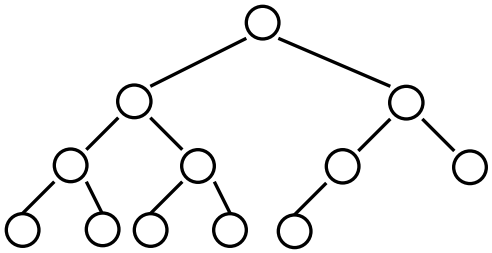
- Concept of Depth:
  - The depth of a node is its distance from the root
  - The depth of a tree is the depth of the deepest node
- A binary tree of depth n is balanced if all the nodes at depths 0 through n-2 have two children



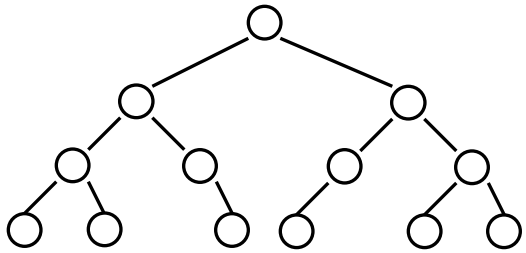


Left-justified binary trees

- A balanced binary tree is left-justified if:
  - all the leaves are at the same depth, or
  - all the leaves at depth n+1 are to the left of all the nodes at depth n



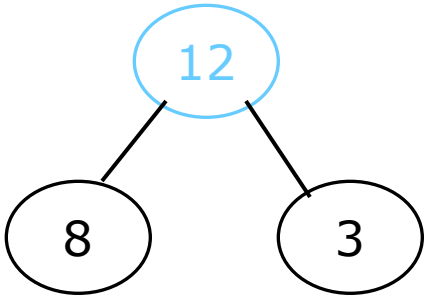
Left-justified



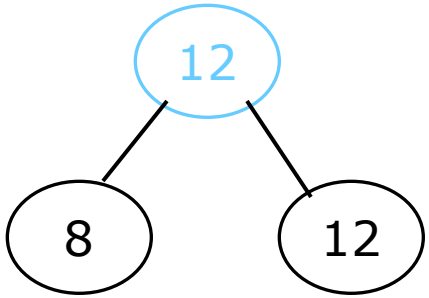
Not left-justified

The heap property

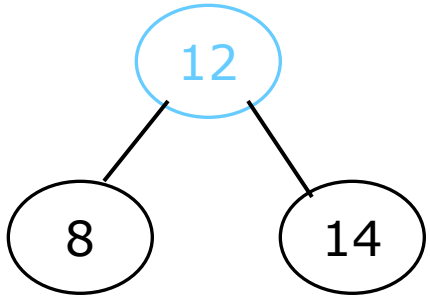
- A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has heap property



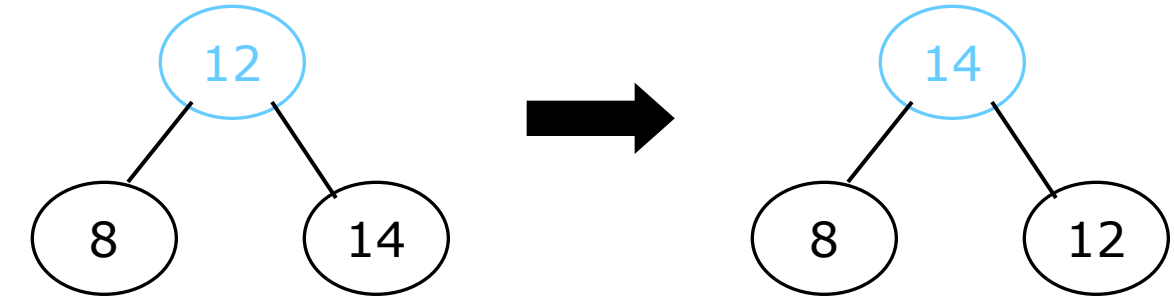
Blue node has heap property



Blue node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



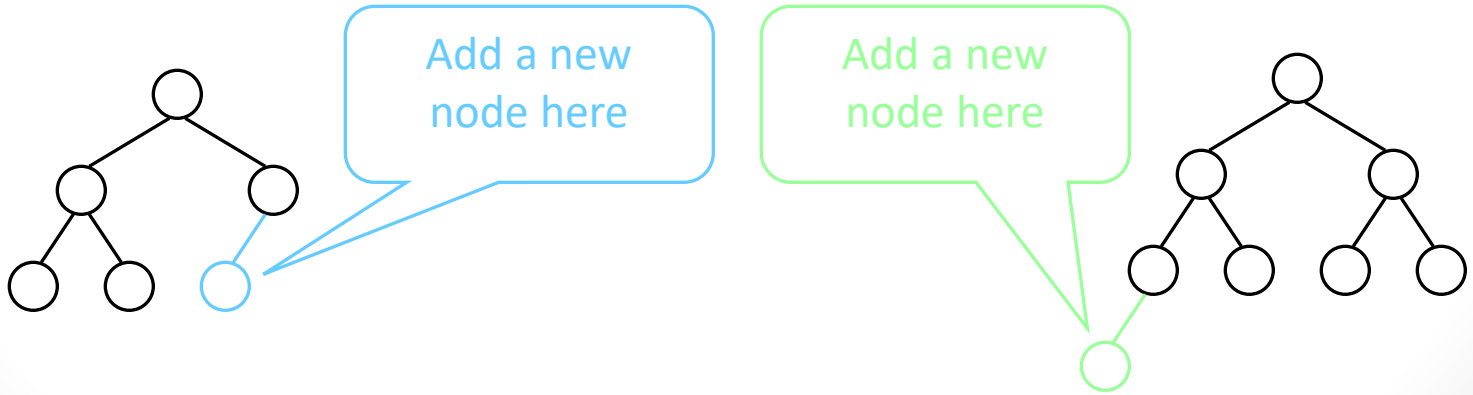
Blue node does not have heap property

Blue node has heap property

- This is sometimes called sifting up
- Notice that the child may have *lost* the heap property

Constructing a heap I

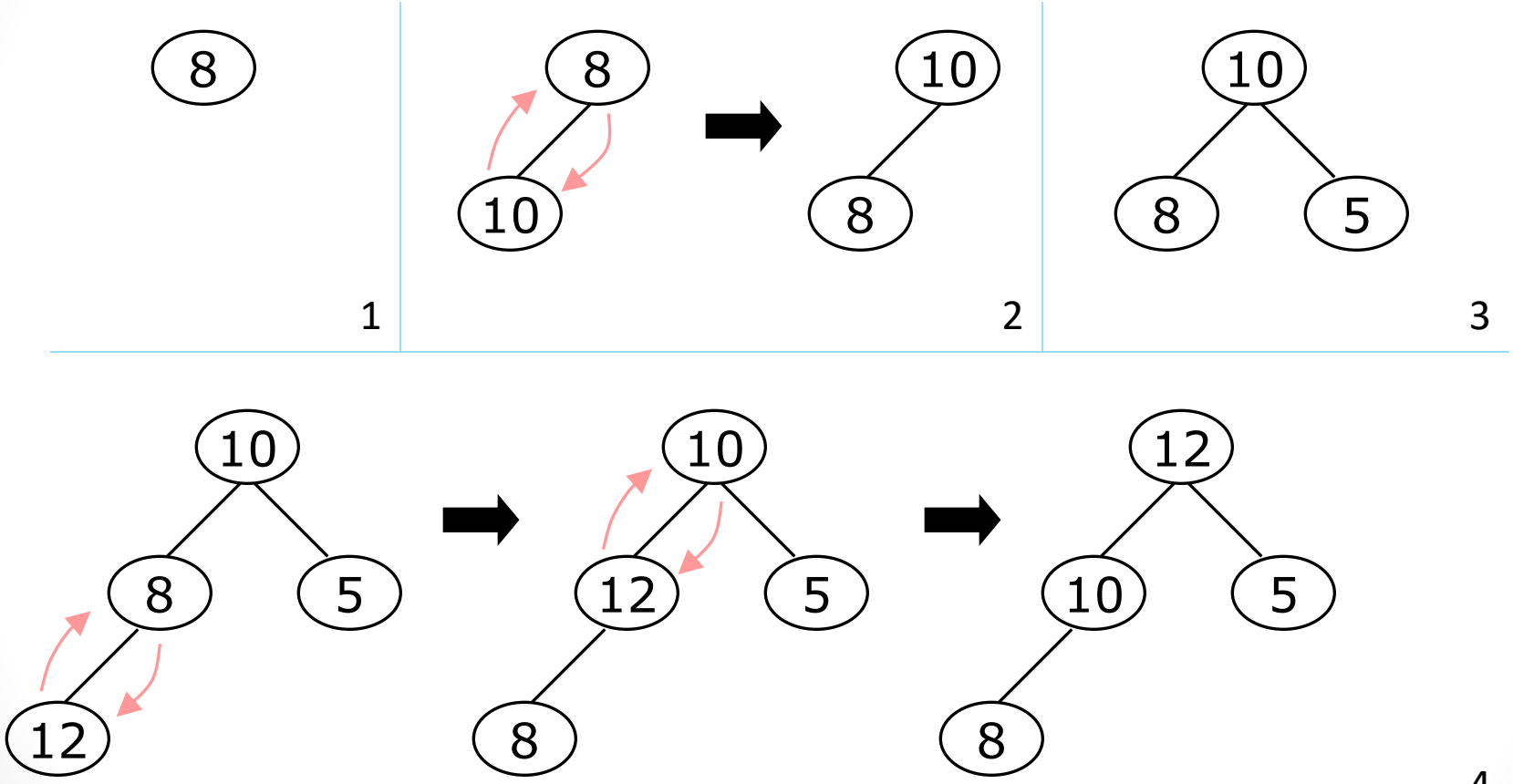
- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node in the last level; it should be filled **from left to right** without any gaps.
  - If the last level is full, start a new level
- Examples:



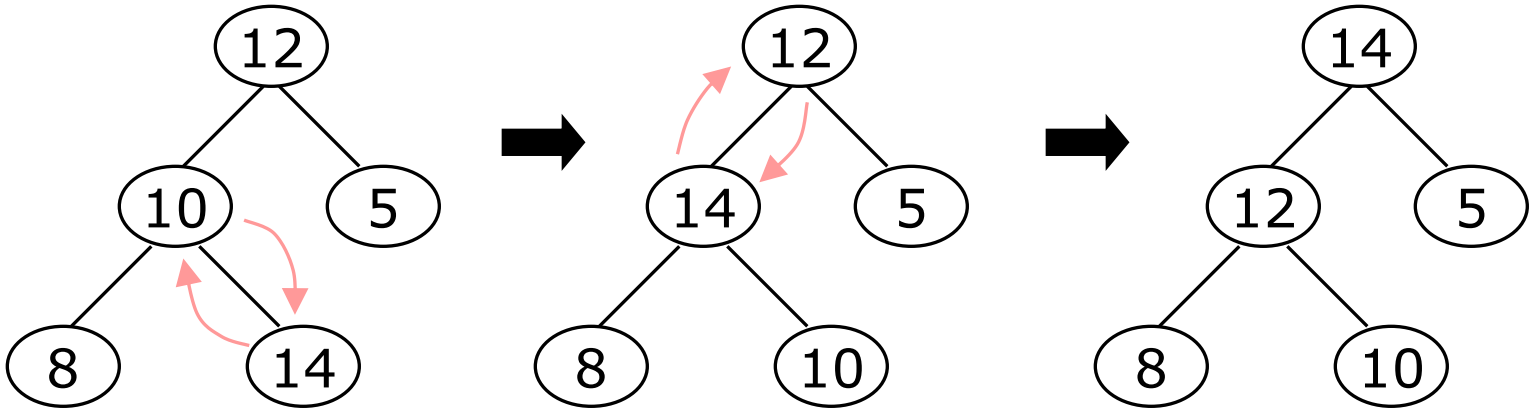
Constructing a heap II

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we sift up
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node
- We repeat the sifting up process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root

Constructing a heap III



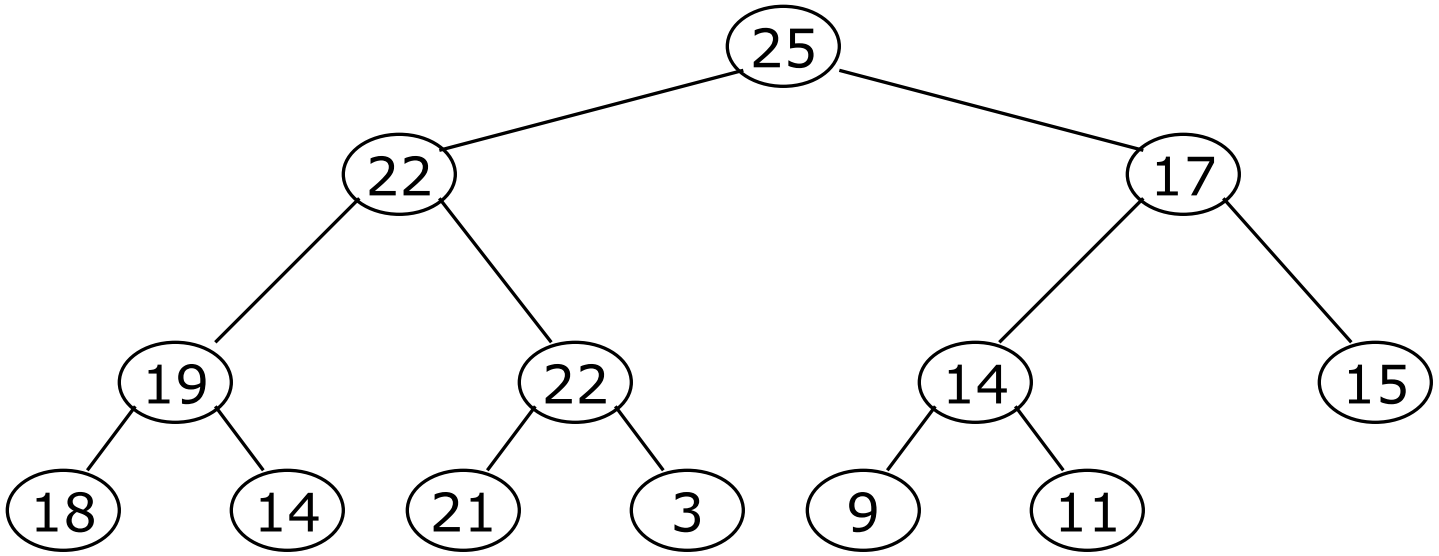
Other children are not affected



- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

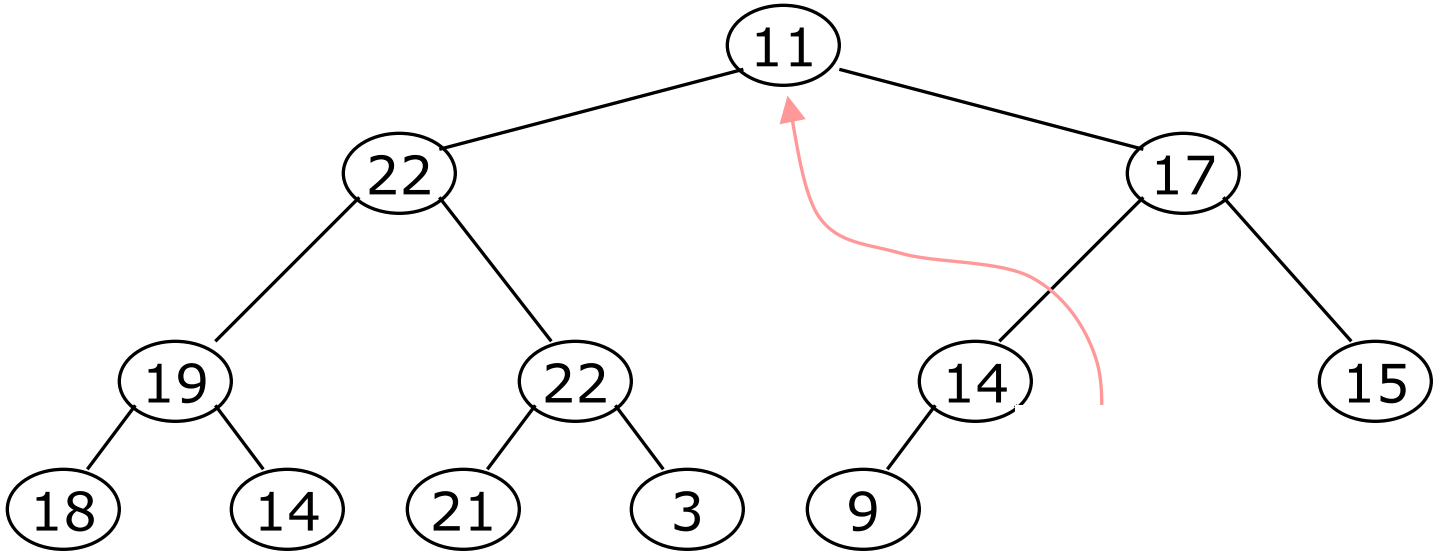
- Here's a sample binary tree after it has been heapified



- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Removing the root

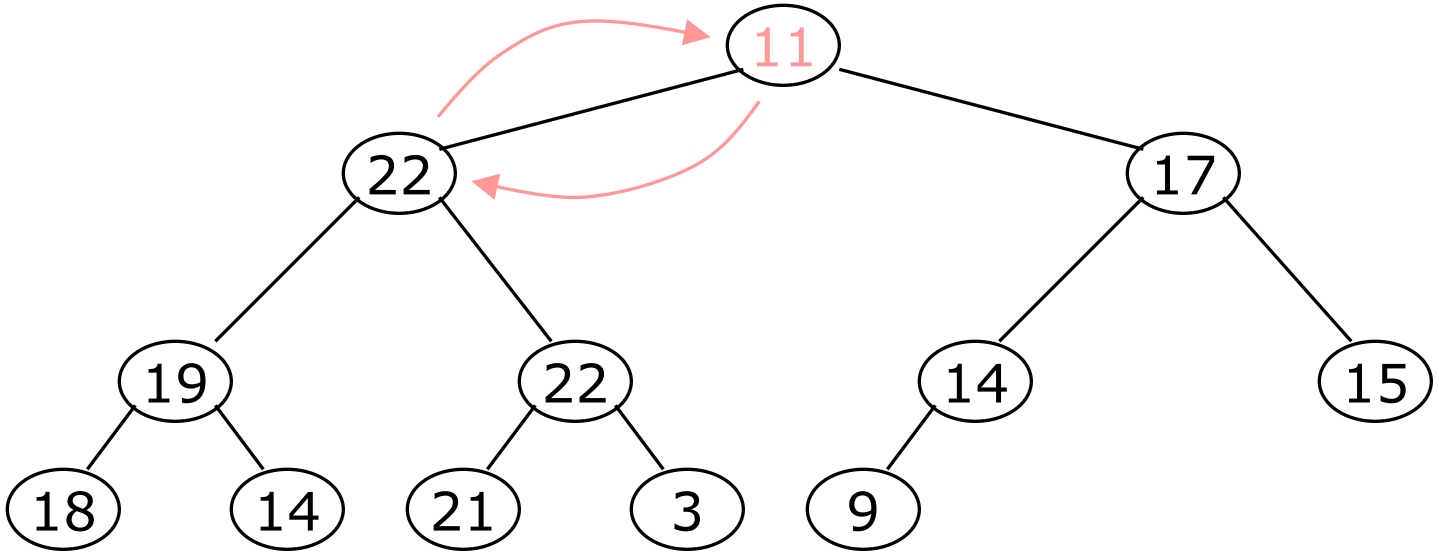
- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- How can we fix the binary tree so it is once again *balanced and left-justified?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

The reHeap method

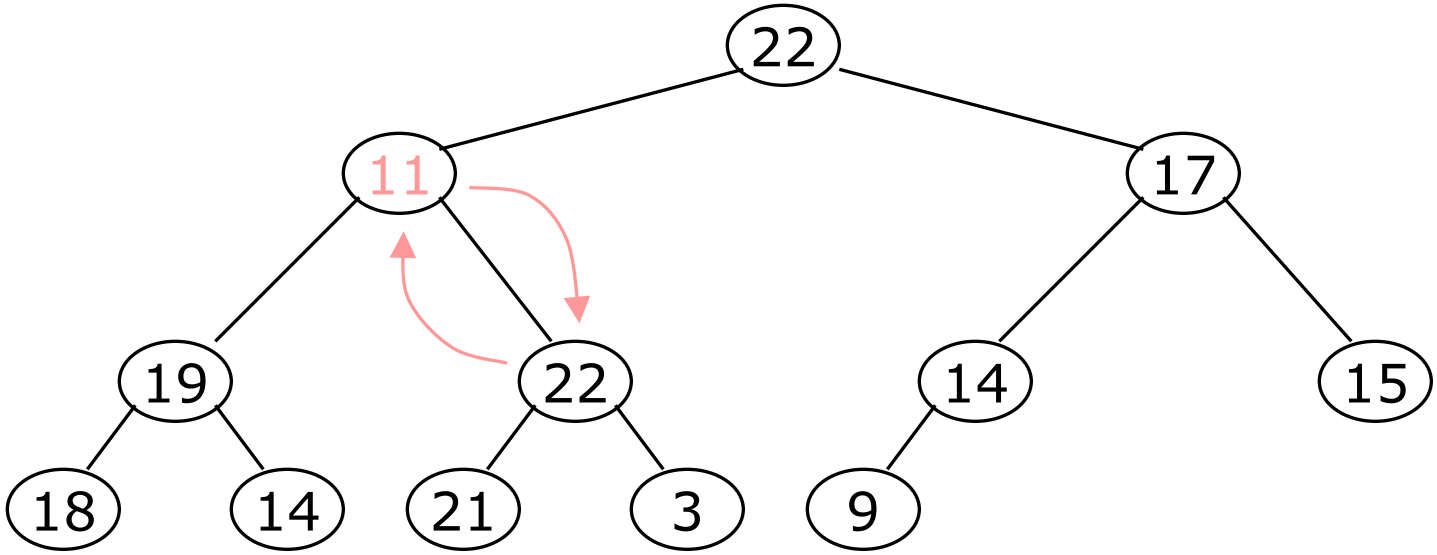
- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property



- We can siftUp() the root
- After doing this, one and only one of its children may have lost the heap property

The reHeap method

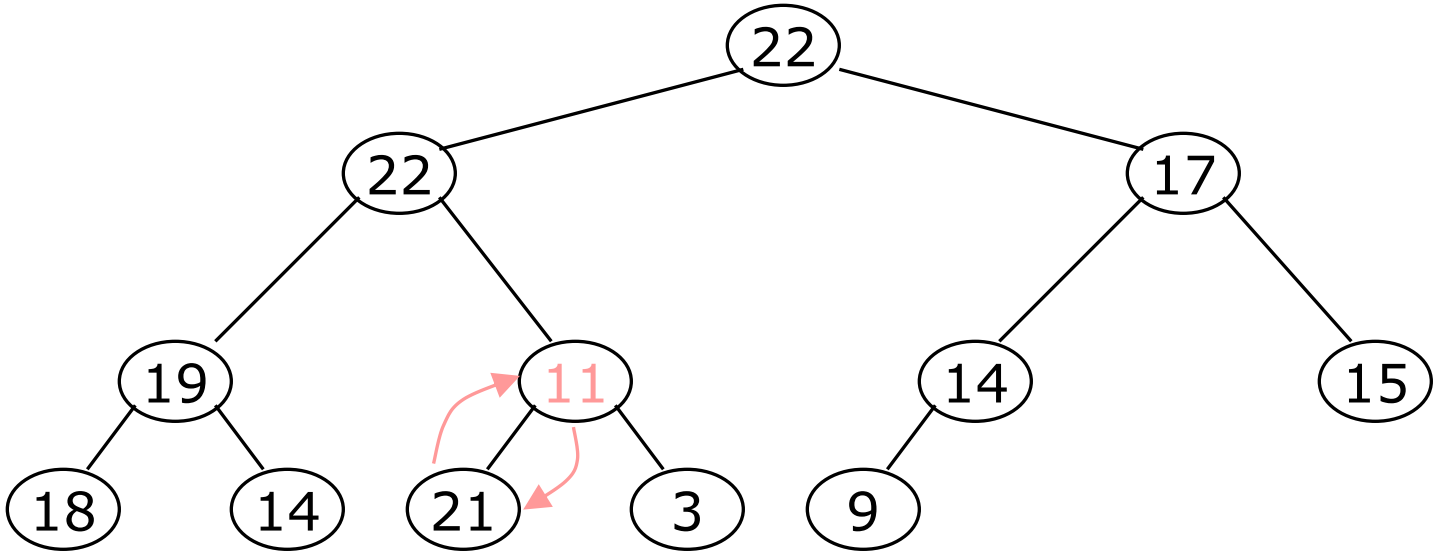
- Now the left child of the root (still the number 11) lacks the heap property



- We can siftUp() this node
- After doing this, one and only one of its children may have lost the heap property

The reHeap method

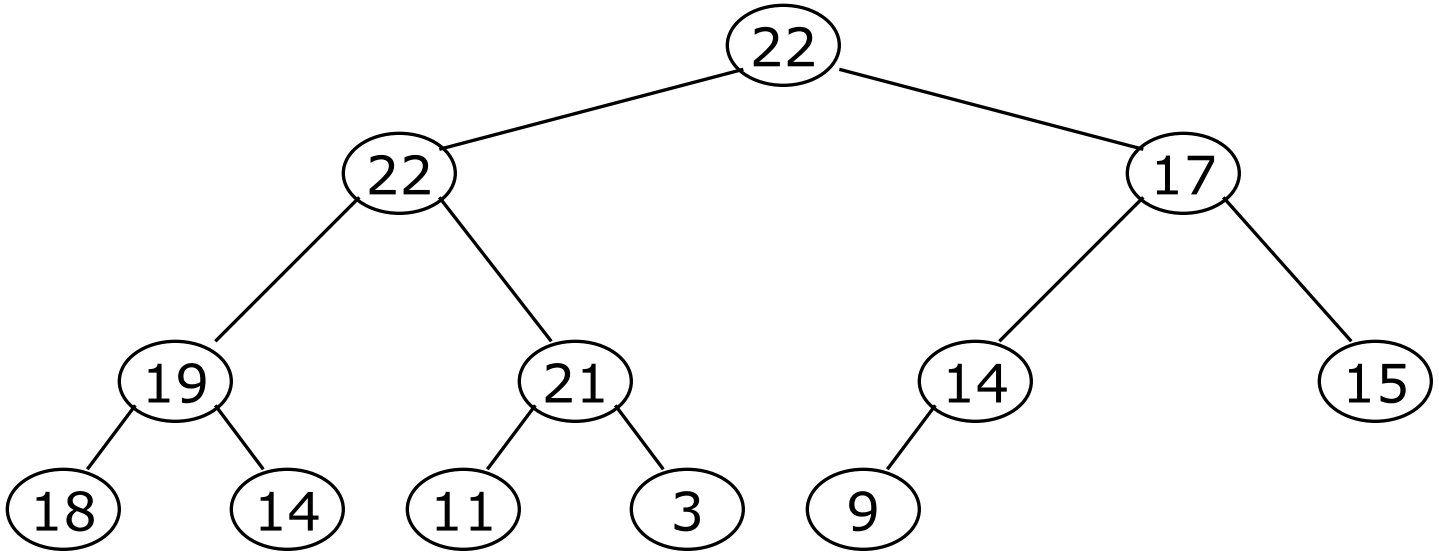
- Now the right child of the left child of the root (still the number 11) lacks the heap property:



- We can siftUp() this node
- After doing this, one and only one of its children may have lost the heap property —but it doesn't, because it's a leaf

The reHeap method

- Our tree is once again a heap, because every node in it has the heap property



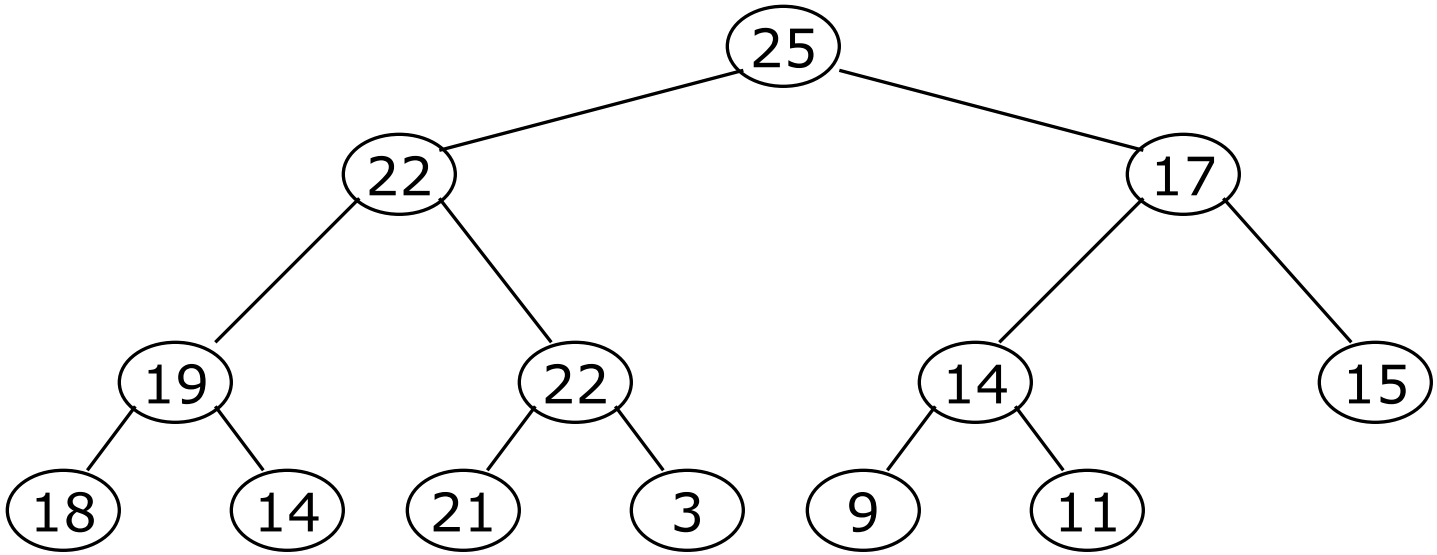
- Once again, the largest (or *a* largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest

Sorting

- What do heaps have to do with sorting an array?
- Here's the neat part:
  - Because the binary tree is *balanced* and *left justified,* it can be represented as an array
  - All our operations on binary trees can be represented as operations on *arrays*
  - To sort:

```
heapify
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

Mapping into an array

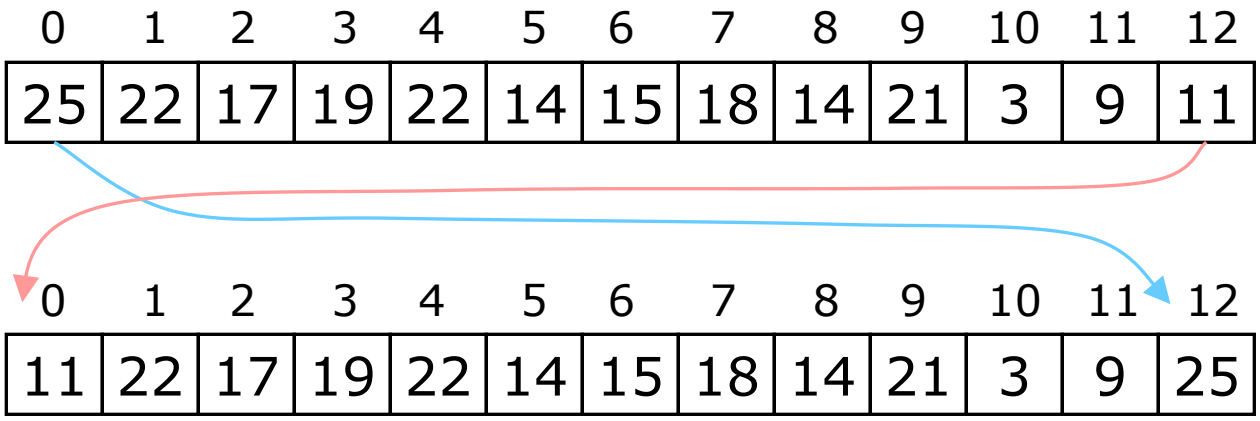


0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

- Notice:
  - The left child of index i is at index 2*i+1
  - The right child of index i is at index 2*i+2
  - Example: the children of node 3 (19) are 7 (18) and 8 (14)

Removing and replacing the root

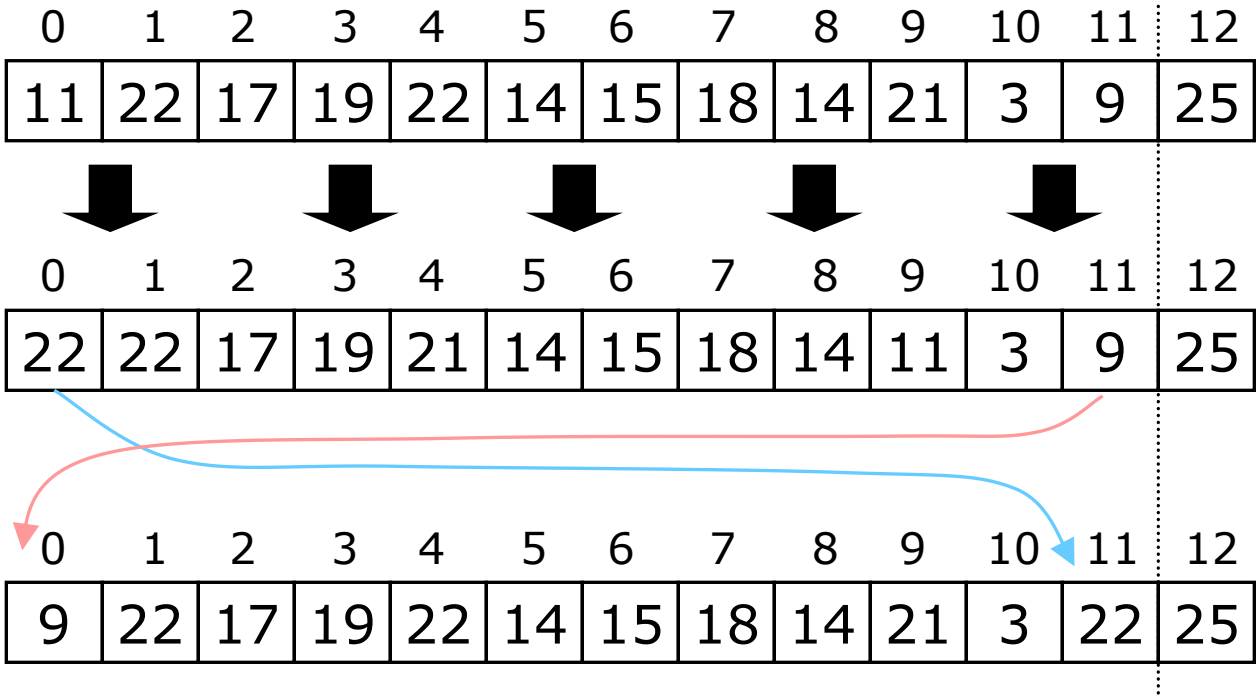
- The “root” is the first element in the array
- The “rightmost node at the deepest level” is the last element
- Swap them...



- ...And pretend that the last element in the array no longer exists—that is, the “last index” is 11 (9)

Reheap and repeat

- Reheap the root node (index 0, containing 11)...



- ...And again, remove and replace the root node
- Remember, though, that the "last" array index is changed
- Repeat until the last becomes first, and the array is sorted!

Analysis I

- Here's how the algorithm starts:

  heapify the array ;

- Heapifying the array: we add each of n nodes
  - Each node has to be sifted up, possibly as far as the root
    - Since the binary tree is perfectly balanced, sifting up a single node takes O(log n) time
  - Since we do this n times, insertion takes n*O(log n) time, that is, O(n log n) time

Analysis II

- Here's the rest of the algorithm:

```
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

- We do the while loop n times (actually, n-1 times), because we remove one of the n nodes each time
- Removing and replacing the root takes O(1) time
- Therefore, the total time is n times however long it takes the reheap method

Analysis III

- To reheap the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
- The binary tree is perfectly balanced
- Therefore, this path is $O(\log n)$ long
  - And we only do $O(1)$ operations at each node
  - Therefore, reheaping takes $O(\log n)$ times
- Since we reheap inside a while loop that we do n times, the total time for the while loop is $n*O(\log n)$, or $O(n \log n)$

Analysis IV

- Here's the algorithm again:

```
heapify the array;
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

- We have seen that heapifying takes $O(n \log n)$ time
- The while loop takes $O(n \log n)$ time
- The total time is therefore $O(n \log n) + O(n \log n)$
- This is the same as $O(n \log n)$ time

Do it Yourself

- Sort the number using Heap Sort

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

Counting Sort

Counting Sort is a **non-comparison-based** sorting algorithm that works well when there is limited range of input values.

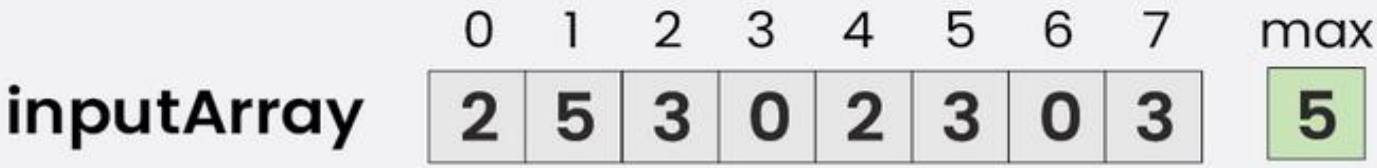
It is particularly efficient when the range of input values is small compared to the number of elements to be sorted.

The basic idea behind Counting Sort is to count the frequency of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

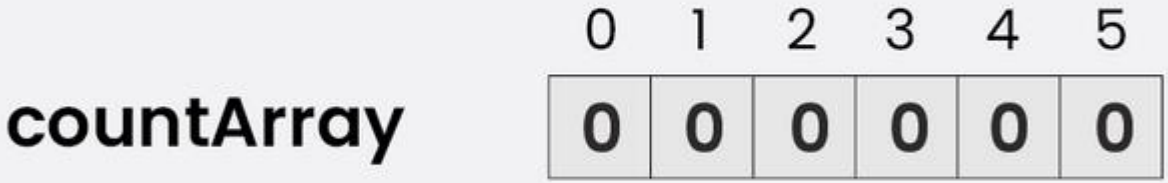
Counting Sort Algorithm

- Declare an auxiliary array countArray[] of size max(inputArray[])+1 and initialize it with 0.
- Traverse array inputArray[] and map each element of inputArray[] as an index of countArray[] array, i.e., execute countArray[inputArray[i]]++ for 0 <= i < N.
- Calculate the prefix sum at every index of array inputArray[].
- Create an array outputArray[] of size N.
- Traverse array inputArray[] from end and update outputArray[countArray[inputArray[i]] – 1] = inputArray[i]. Also, update countArray[inputArray[i]] = countArray[inputArray[i]]–.

Step 1 :

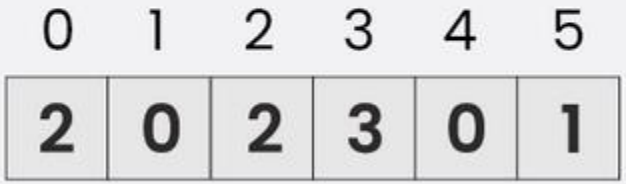

0 1 2 3 4 5 6 7
max
inputArray
2 5 3 0 2 3 0 3
5

Step 2 :


0 1 2 3 4 5
countArray
0 0 0 0 0 0

Step 3 :

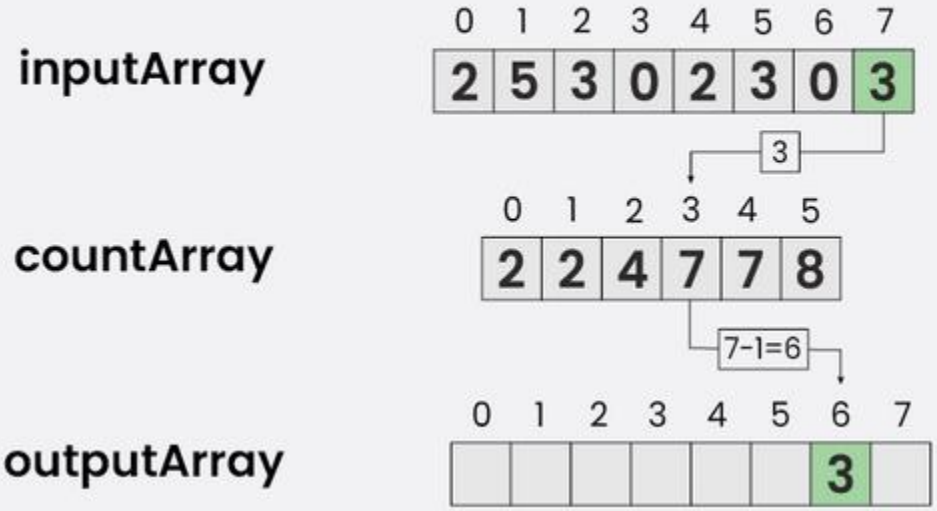
countArray



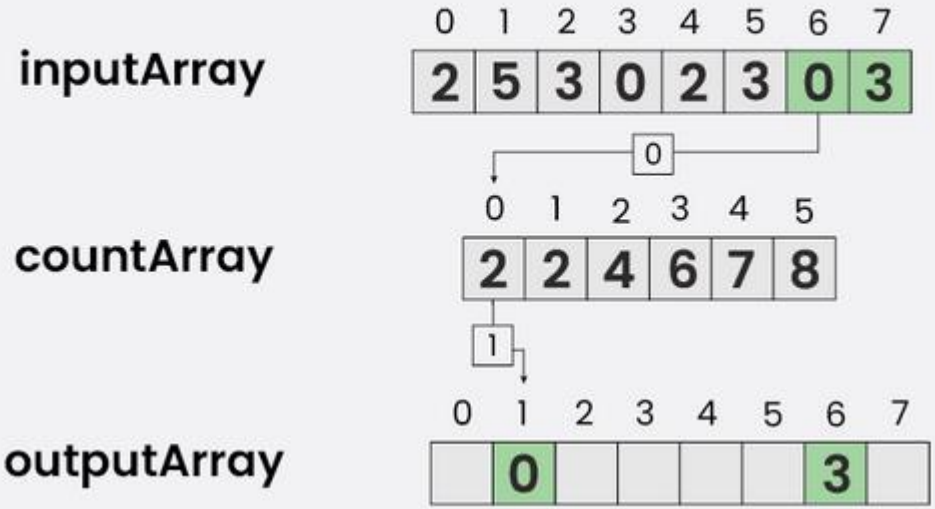
Step 4 :

countArray

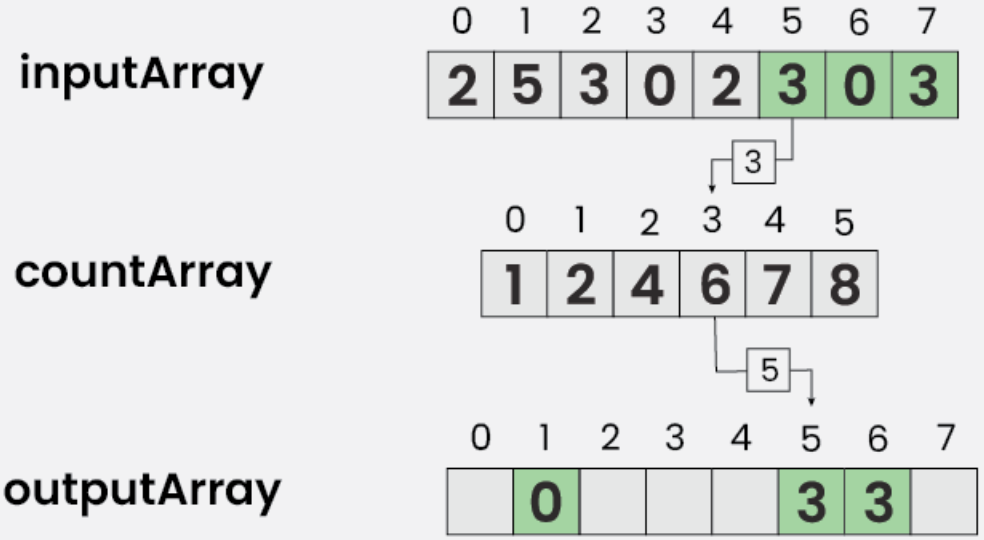
Step 5 :


inputArray
0 1 2 3 4 5 6 7
2 5 3 0 2 3 0 3
3
countArray
0 1 2 3 4 5
2 2 4 7 7 8
7-1=6
outputArray
0 1 2 3 4 5 6 7
3

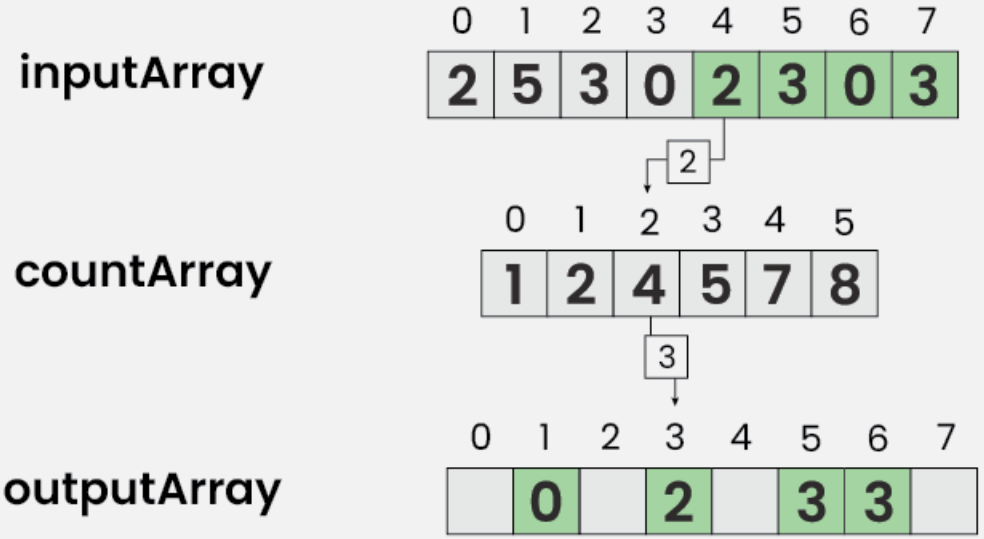
Step 6 :


inputArray
0 1 2 3 4 5 6 7
2 5 3 0 2 3 0 3
0
countArray
0 1 2 3 4 5
2 2 4 6 7 8
1
outputArray
0 1 2 3 4 5 6 7
0 3

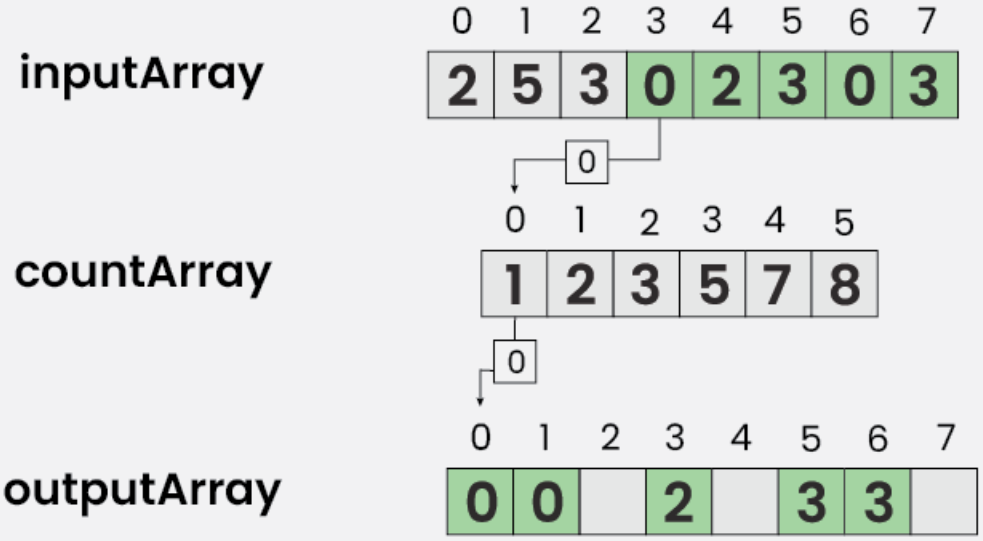
Step 7 :


inputArray
0 1 2 3 4 5 6 7
2 5 3 0 2 3 0 3
3
countArray
0 1 2 3 4 5
1 2 4 6 7 8
5
outputArray
0 1 2 3 4 5 6 7
0 3 3

Step 8 :


inputArray
0 1 2 3 4 5 6 7
2 5 3 0 2 3 0 3
2
countArray
0 1 2 3 4 5
1 2 4 5 7 8
3
outputArray
0 1 2 3 4 5 6 7
0 2 3 3

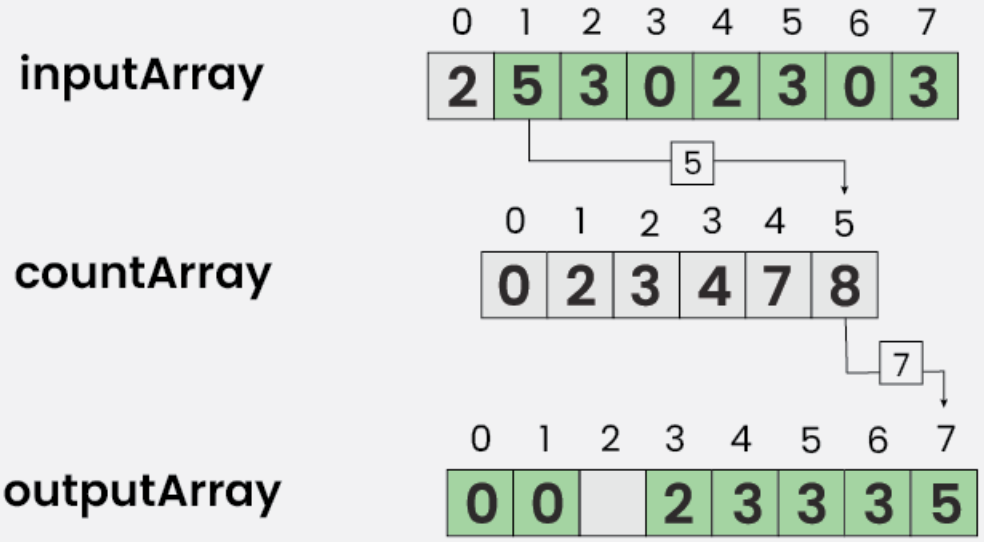
Step 9 :


inputArray
0 1 2 3 4 5 6 7
2 5 3 0 2 3 0 3
0
countArray
0 1 2 3 4 5
1 2 3 5 7 8
0
outputArray
0 1 2 3 4 5 6 7
0 0 2 3 3

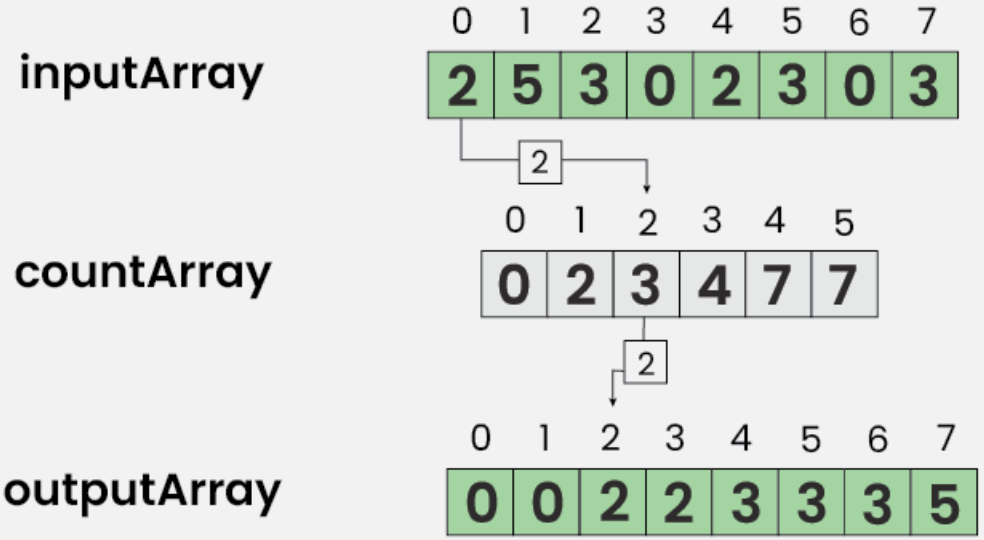
Step 10 :


inputArray
0 1 2 3 4 5 6 7
2 5 3 0 2 3 0 3
3
countArray
0 1 2 3 4 5
0 2 3 5 7 8
4
outputArray
0 1 2 3 4 5 6 7
0 0 2 3 3 3

Step 11 :


inputArray
0 1 2 3 4 5 6 7
2 5 3 0 2 3 0 3
5
countArray
0 1 2 3 4 5
0 2 3 4 7 8
7
outputArray
0 1 2 3 4 5 6 7
0 0 2 3 3 3 5

Step 12 :


inputArray
0 1 2 3 4 5 6 7
2 5 3 0 2 3 0 3
2
countArray
0 1 2 3 4 5
0 2 3 4 7 7
2
outputArray
0 1 2 3 4 5 6 7
0 0 2 2 3 3 3 5

Complexity

- Time Complexity:
  - O(n+k)
    - n is number of elements
    - k is maximum key
- Space Complexity:
  - O(k)

Disadvantages of Counting Sort

- Counting sort doesn't work on decimal values
- Counting sort is inefficient if the range of values to be sorted is very large

Which Sort to use when

- **Selection Sort**

•When the list is small. As the time complexity of the selection sort is **$O(N^2)$** which makes it inefficient for a large list.
•When memory space is limited because it makes the minimum possible number of swaps during sorting.

•**Bubble Sort**

•It works well with large datasets where the items are almost sorted because it takes only one iteration to detect whether the list is sorted or not. But if the list is unsorted to a large extend then this algorithm holds good for small datasets or lists.
•This algorithm is fastest on an extremely small or nearly sorted set of data.

Which Sort to use when

- **Insertion Sort**

•If the data is nearly sorted or when the list is small as it has a complexity of **$O(N^2)$** and if the list is sorted a minimum number of elements will slide over to insert the element at its correct location.
•This algorithm is stable and it has fast running case when the list is nearly sorted.
•The usage of memory is a constraint as it has space complexity of O(1).

- **Merge Sort**

1. Merge sort is used when the data structure **doesn't support random access** since it works with pure sequential access that is forward iterators, rather than random access iterators.
2. It is widely used for external sorting, where random access can be very, very expensive compared to sequential access.
3. It is used where it is known that the data is similar data.
4. Merge sort is fast in the case of a linked list.
5. It is used in the case of a linked list as in a linked list for accessing any data at some index we need to traverse from the head to that index and merge sort accesses data sequentially and the need of random access is low.
6. The main advantage of the merge sort is its stability, the elements compared equally retain their original order.

- **Quick Sort**

Quick sort is the fastest, but it is not always O(N*log N), as there are worst cases where it becomes $O(N^2)$.

Quicksort is probably more effective for datasets that fit in memory. For larger data sets it proves to be inefficient so algorithms like merge sort are preferred in that case.

Quick Sort is an in-place sort (i.e. it doesn't require any extra storage) so it is appropriate to use it for arrays.

External Sorting

- External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted does not fit into the main memory of a computing device (usually RAM) and instead, must reside in the slower external memory (usually a hard drive).
- External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in the main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

When We do External Sorting?

- When the unsorted data is too large to perform sorting in computer internal memory then we use external sorting.
- In external sorting we use the secondary device. in a secondary storage device, we use the tape disk array.
- when data is large like in merge sort and quick sort.
- Quick Sort: best average runtime.
- Merge Sort: Best Worse case time.
- To perform sort-merge, join operation on data.
- To perform order by the query.
- To select duplicate element.
- Where we need to take large input from the user.