Sorting

Chapter 7

Data Structures: A Programming Approach with C, PHI
Dharmender Singh Kushwaha & A.K.Misra

Introduction

- A sorting algorithm arranges items of a list in a certain order.
- The most-used orders are numerical order and lexicographical or alphabetical order.
- Efficient sorting is important for reducing the time and space complexity.

Sorting Algorithms

- Some important Sorting Techniques are:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Bucket Sort
  - Heap Sort
  - Merge Sort
  - Quick Sort &
  - Radix Sort

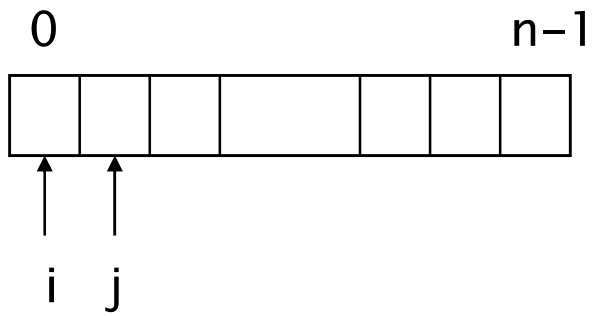
Bubble sort

- Each element is compared with its adjacent element.
- If the element is larger than the second one,
  - then the position of the elements are interchanged.
  - otherwise it is not changed.
- This is repeated until no more elements are left for comparison .

Bubble sort

Algorithm

1. Initialization
   set i = 0
2. Repeat steps 3 to 5 until i<n
3. Set j = i + 1
4. Repeat steps 5 until j < n – i -1
5 .If a[i] > a[j] then
   set temp = a[i]
   set a[i] = a[j]
   set a[j] = temp
   endif
6. Exit



Bubble sort

Sort the sequence (3 7 1 4 9)

- **Ist Pass:**

(**3 7** 1 4 9)	No Swap
(3 **7 1** 4 9)	Swap (3 **1 7** 4 9)
(3 1 **7 4** 9)	Swap (3 1 **4 7** 9)
(3 1 4 **7 9**)	No Swap

- **IInd Pass**

(**3 1** 4 7 9)	Swap (**1 3 4 7** 9)
(**1 3** 4 7 9)	No Swap
(1 **3 4** 7 9)	No Swap
(1 3 **4 7** 9)	No Swap
(1 3 4 **7 9**)	No Swap **sorted sequence is (1 3 4 7 9)**

Analysis of Bubble Sort

- The first pass requires (n-1) comparisons to fix the highest element to its location
- Second pass requires (n-2)
- Kth pass requires (n-k)
- Last pass requires only one comparison
- Total comparison:
  - (n-1) +(n-2)+(n-3)—(n-k)+3+2+1
  - F(n) = n(n-1)/2
  - F(n) = $O(n^2)$

Selection Sort

- This is an in-place sorting algorithm.
- A pass through the array in is made to locate the minimum value.
- This element is placed at the first position.
- The second pass finds out the next smallest element and is placed in second position and so on.
- Once the second last element is compared with the last element, the entire elements are sorted in ascending order.

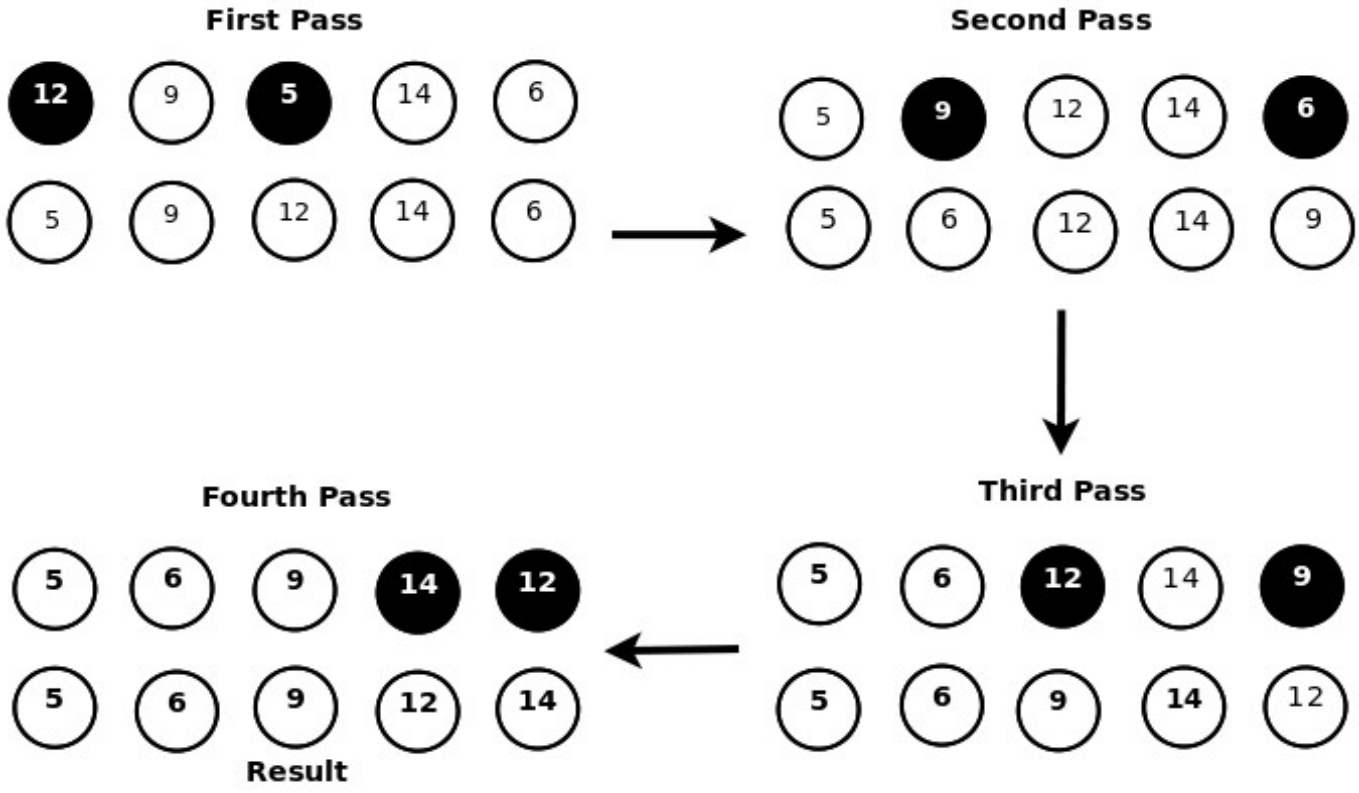
Selection Sort

Algorithm

- Find the minimum value in the list.
- Swap it with the value in the first position.
- Repeat the steps above for remaining elements in the list starting at the second position and so on.

Selection Sort

Sort the Sequence 12 9 5 14 6



Data Structures: A Programming Approach with C, PHI
Dharmender Singh Kushwaha & A.K.Misra

Selection Sort

Analysis

- Selecting the lowest element requires scanning all n elements and n-1 comparisons.
- Finding the next lowest element require the remaining n-1 elements and so on.
- Total number of comparisons:

  $=(n-1) + (n-2) +...+ 2 + 1 = n(n - 1) /2$

  $= O(n^2)$

Advantages

- Selection sorting is noted for its simplicity.
- It also has performance advantage over more complicated algorithms in certain situations where the number of elements to be sorted are few in number.
- But it has $O(n^2)$ complexity where 'n' is the number of elements in the list, making it inefficient on large lists, and generally performs worse than the similar insertion sort.

Insertion Sort

- Let $a_0,....a_{n-1}$ be the sequence of elements to be sorted.
- At the beginning and after each iteration of the algorithm, the sequence consists of two parts:

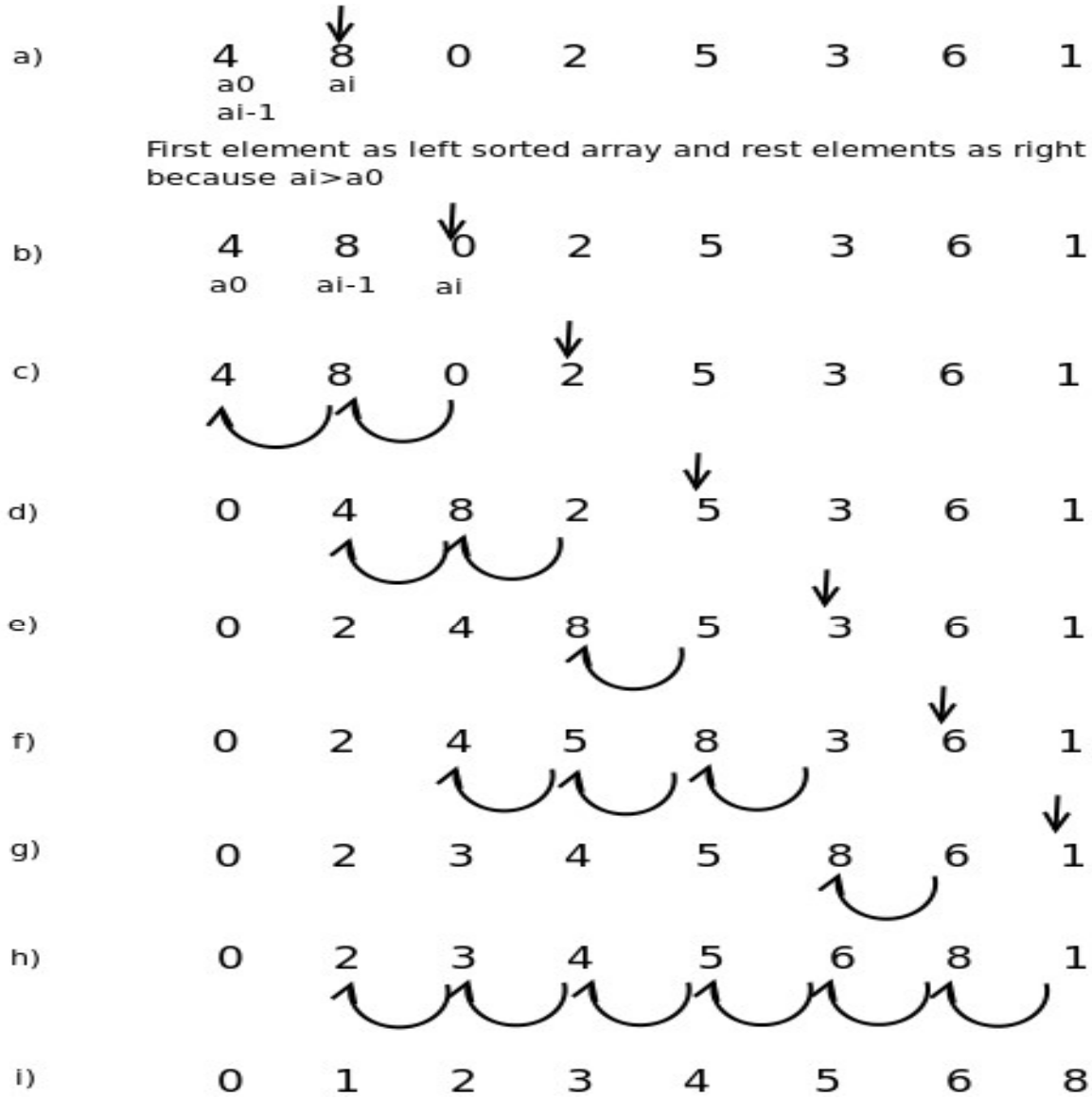
  The first part $a_0,....a_{i-1}$ is already sorted,

  The second part $a_i,...a_{n-1}$ is still unsorted($i \in 0,...,n$)
- To insert element a_i into the sorted part, it is compared with a_{i-1}, a_{i-2} etc.
- When an element a_j with $a_j \leq a_i$ is found, a_i is inserted after it.

Insertion Sort

Insertionsort(Array a)

```
   for i=1 to length[a] -1 do
begin
value = a[i]
j= i-1
while j>= 0 and a[j] > value do
     begin
      a[j + 1] = a[j]
      j = j-1
    end
      a[j + 1] = value
end
```



Data Structures: A Programming Approach with C, PHI
Dharmender Singh Kushwaha & A.K.Misra

Insertion Sort

Analysis

- In the best case, the insertion sorting has θ(n) time complexity.
  - In best case, the input array is already sorted.
  - So, while comparing the first element of rest of the array requires no swapping and only n comparisons are required.
- The worst case occurs when an array to be sorted has all its elements stored in the reverse order.
- Insertion sort takes $O(n^2)$ time in the worst case as well as in the average case.

Comparisons of Insertion Sort with other Sorting Algorithms

- Insertion sort generally makes fewer comparisons than selection sort.
- It requires more write and shift operations because the inner loop can require shifting large section of the sorted position of the array.
- In general, insertion sort will write to the array $O(n^2)$ times while selection sort will write only O(n) times.
- For this reason, selection sort is preferred in those situations where writes to memory are significantly more expensive than reads, such as EEPROM or other variant of flash memories.

Shell sort

- Shell sort shifts out of order elements more than one position at a time.
- The idea is to do something that will get items to roughly the correct position. This allows an element to reach faster toward its expected position.
- It requires relatively lesser amount of memory.
- The last step of Shell sort is a plain insertion sort and the array of the data is sorted.

Shell sort

Sort the sequence [13, 14, 94, 33, 82, 25, 59, 99, 65, 23, 45, 27, 75, 35, 39, 1, 11]

- Arrange the given list into a matrix and sort the columns using an insertion sort.
- Repeat this process, each time with lesser number of longer columns.
- At the end, the matrix has only one column.
- Start with a step-size of 5 and thus we break the list of numbers into a matrix with 5 columns.

Shell sort

Initially the matrix is

13 14 94 33 82
25 59 99 65 23
45 27 75 35 39
1 11

Now we sort each column, which gives us:

1 11 75 33 23
13 14 94 35 39
25 27 99 65 82
45 59

Shell sort

- Read the matrix row wise as a single list of numbers.
- [1, 11, 75, 33, 23, 13, 14, 94, 35, 39, 25, 27, 99, 65, 82, 45, 59]
- List is then again sorted using a step size of 3

1 11 75
33 23 13
14 94 35
39 25 27
99 65 82
45 59

Shell sort

- Now sorting each column, we have:

  1 11 13
  14 23 27
  33 25 35
  39 59 75
  45 65 82
  99 94

Again if we read it back row wise, we have sorted list.

[1 11 13 14 23 27 33 25 35 39 59 75 45 65 82 99 94]

Bucket Sort

- Input elements are initially uniformly distributed to the finite number of fixed size buckets.
- Sort each bucket either recursively or using a different sorting algorithm.
- Finally, the contents of each bucket are concatenated.

Bucket Sort

Bucketsort (a[i], n)

Corresponding_buckets: array of n buckets

for i = 0 to (n-1)

Step-I: now place each item from a[i] into respective buckets

Place a[i] into corresponding_buckets based on most significant bit (m_s_bits(a[i], k))

k is the no of items in each corresponding_buckets ( )

for i = 0 to (n–1)

Step-II: sort items within each bucket either recursively or by using a different sorting algorithm

Sort (corresponding_buckets[i])

Step-III: concatenate all the individual buckets

Concatenate corresponding_buckets[0], ..., corresponding_buckets[n-1]

Example

- Illustrate sorting of the given list using bucket sort.
  - 39, 19, 16, 27, 23, 8, 44, 5, 31, 41
- Since all the elements are two digit numbers less than 50, we create 5 buckets.
- The first bucket holds element from 1 – 10, second from 11 – 20 and so on. Thus the array is partitioned.
- Elements in each bucket are sorted either recursively or using other simpler sorting algorithm.
- It is a generalization of pigeonhole sort.

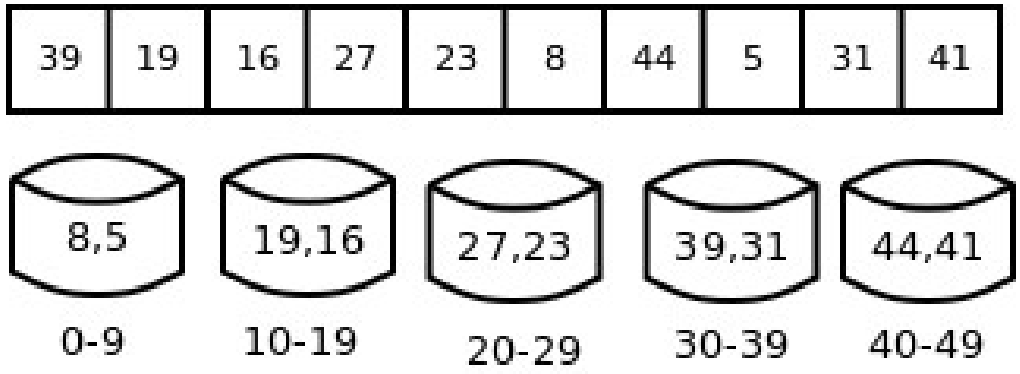
Bucket Sort

- Sort the sequence

  [39, 19, 16, 27, 23, 8, 44, 5, 31, 41]

Step-I

- Items are placed in different buckets according to their range without changing their order.



Bucket Sort

Step-II

- Items are sorted within each bucket either recursively or using any other sorting algorithm.

- Concatenate the items from all buckets into a single array.

5	8	16	19	20	29	30	39	40	49

Bucket Sort

Complexity

- Bucket sort works well for data set where the possible key values are known and relatively small.
- If there are few elements/buckets, then the cost of sorting the elements can be significantly reduced.
- Worst case complexity of bucket sort is of order O (n^2) where n is the number of buckets.
- Can achieve O(*n log n).*
- Best and Average case complexity is O (*m + n*) where 'm' is the number of items to be sorted and 'n', num of buckets.

Heap sort

- Heap Sort inserts the input list elements into a heap data structure.
- We start either by the smallest or the largest value of the node and create the heap.
- The root node is guaranteed to be the smallest or the largest element.
- If this element is deleted, the heap gets rearranged and the second largest or the second smallest element occupies the root.
- Heap is a complete binary tree.

Heap sort

Algorithm

- Array elements are mapped into a heap.
- Converting the input array into a heap requires re-arranging the elements of the array, so that it possesses the heap property.
- Let the elements to be sorted be:

24 85 48 76 57 2 5 32 12

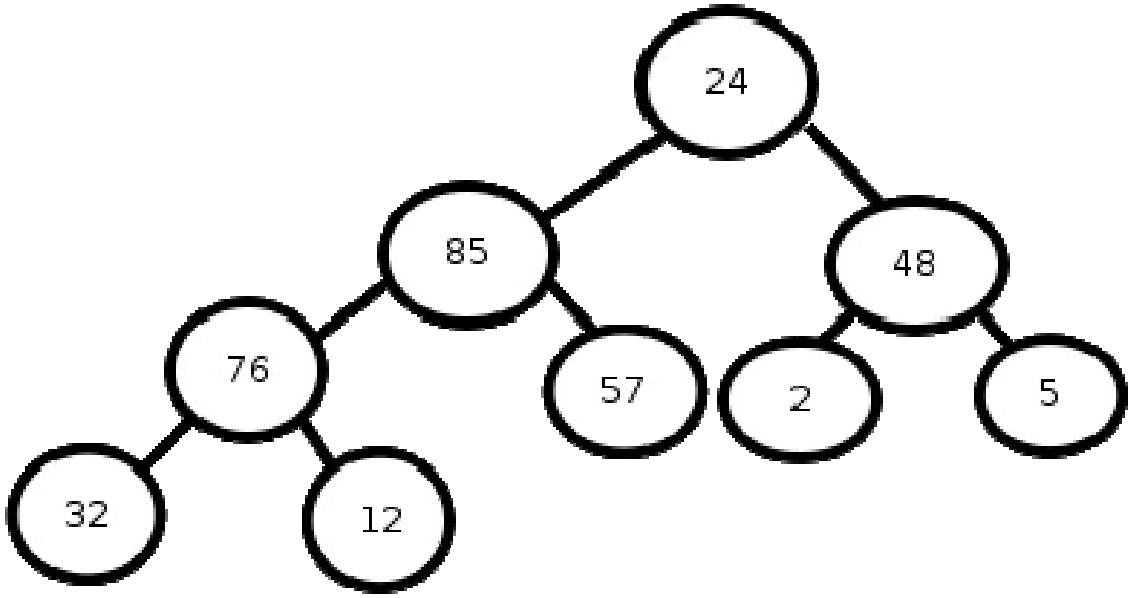
Step1: Arrange the elements in an array.

24	85	48	76	57	2	5	32	12

Heap sort

Step 2: Now heapify this array. If may not resemble an exact heap structure

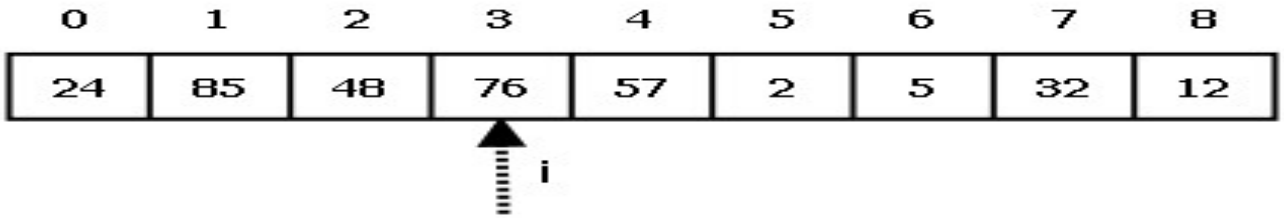
24	85	48	76	57	2	5	32	12



Data Structures: A Programming Approach with C, PHI
Dharmender Singh Kushwaha & A.K.Misra

Heap sort

Step 3: Find the last non-leaf node. This is “76” and indexed by ‘i’.



- We find the children of this last non leaf node at index 3. So left child is at (2*3+1) = 7,and the element at index 7 happens to be is 32. Right child is at (2*3+2) = 8 which is 12.
- **Step 4:** Compare value of last non-leaf node with maximum of the two children. Is 76 > 32 => Yes. Since it is greater, no swapping is carried out.

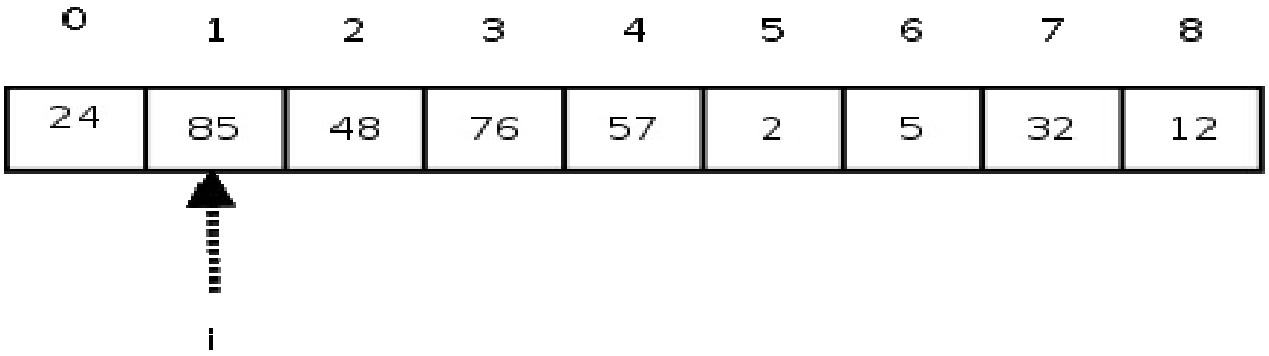
Heap sort

Step 5: Decrement the index " i " by 1 and find the children of the element of the node i-1 (2^{nd} position).

- Element at index 2 is 48. Its left children is at (2*2+1) = 5 which is 2 and its right child is at (2*2+1) = 6 which is element 5.
- Now compare 48 with larger of two children which is 6. 48 > 6 => Yes, So no swapping of elements is required.

Step 6: Go to step 5, so again decrement counter i by 1, so that it now points to 85.

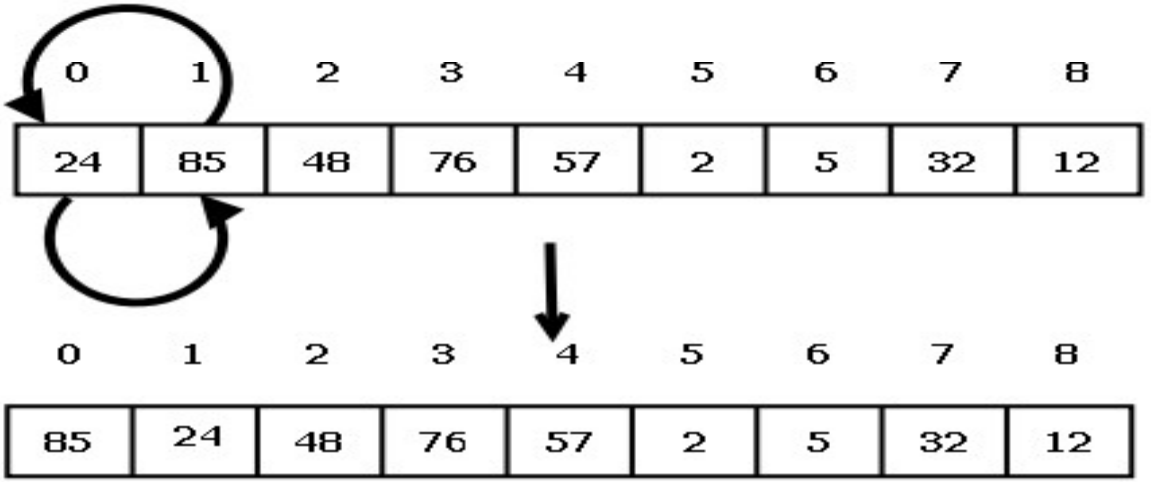
Heap sort



- Its left child is at 2*1+1= 3, which is element 76 and its right child is at 2*2+1= 4 which is element 57.
- Since 85 is greater than the two children, so no swapping is done.
- Decrement i by 1. Now it becomes 0.

Heap sort

- At 0^{th} position the element is 24. Its left and right child is 85 and 48.
- Now compare 24 with larger of its two children. i.e. 24 > 48 => No, so we perform the swap operation.



Heap sort

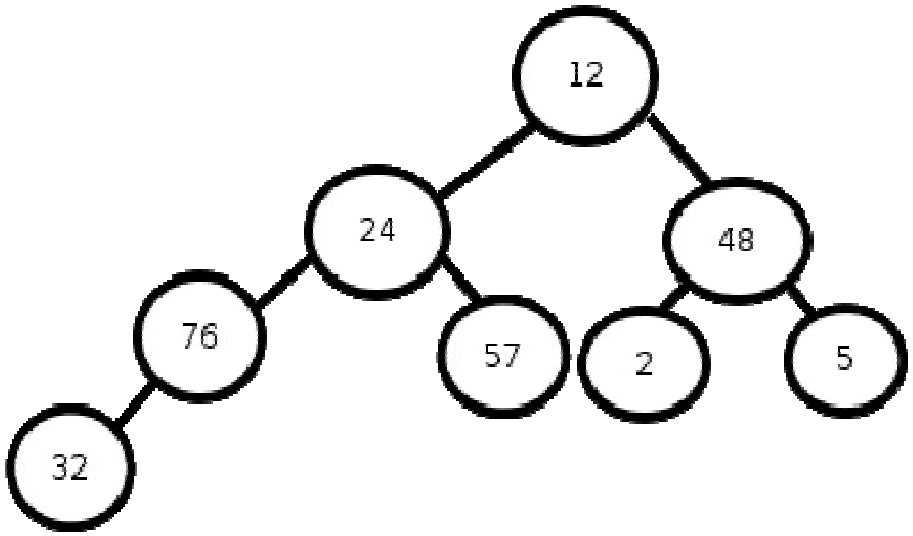
- We can see that the largest element is at index position '0‘.
- If we are sorting it in ascending order , then we can swap the first element at index 0 with the last element.

12	24	48	76	57	2	5	32	85

- Since 85 reaches its final position, so we no longer include it in the further process.
- Repeat the same process starting at Step 1, till all elements reach their proper position.

Heap sort

- Store back the elements in a heap structure.



- After performing a heapify operation, we shall get the second largest element at last position and so on.

Heap sort

Analysis

- Heap is a Complete binary tree with n vertices has a depth of at most log (n).
- Downheap operation requires at most (log n) steps.
- Buildheap procedure calls downheap for each vertex, therefore it requires at most (n log n) steps.
- Thus, the time complexity of heapsort is O (n log (n)).

Merge Sort

- It is based on divide-and-conquer paradigm.
- Merge sort takes the advantage of the ease of merging already sorted lists into a new sorted list.
- It starts by comparing every two elements and swapping them if the first element should come after the second.
- It then merges each of the resulting lists of two elements in the first pass.
- Then merges the lists of adjacent four elements, and this continues until at last two sublists are merged into one final sorted list.

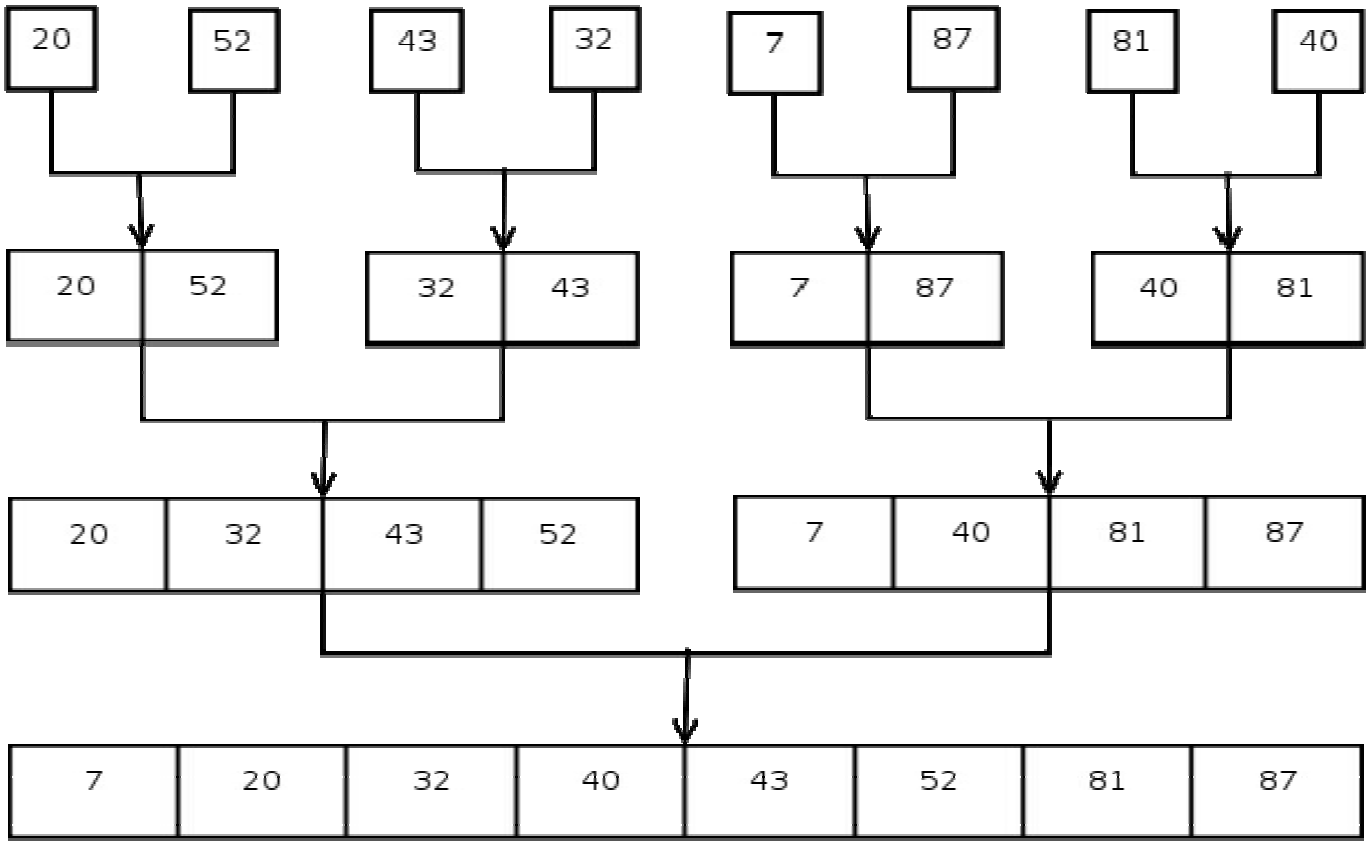
Merge Sort

The operation steps are as follows:

- If the list to be sorted is of length 0 or 1, then it is already sorted, else
- Divide the unsorted list into two sub-lists of about half the size.
- Sort each sub-list recursively by re-applying merge sort.
- Merge the two sub-lists back into one sorted list.

Merge Sort

Elements to be sorted be 20,52,43,32,7,87,81,40



Data Structures: A Programming Approach with C, PHI
Dharmender Singh Kushwaha & A.K.Misra

Merge Sort

Analysis

- It took 3 passes to sort a file of 8 elements.
- In general we never need more than $\log_2 n$ passes.
- Each pass does fewer than "n" comparisons.
- Thus total number of comparisons required are always less than $(n*\log_2 n)$.
- Total number of comparisons required are always less than $(n*\log_2 n)$.

Quick Sort

- Quick sort is well-known sorting algorithm developed by C.A.R.Hoare, that employs a divide and conquer strategy.
- It stores all the elements to be sorted in an array and then partitions it by choosing an element, called a pivot.
- Elements that are to left side to this pivot element are lesser than it and element at right side are larger than it.
- These two parts, left array and right array, are sorted separately.
- A separate pivot element is chosen and those parts are further divided in two parts.

Quick Sort

Recursive Algorithm

i. Select an element, called a pivot, from the elements stored in the array.

ii. Re-arrange the array so that all elements which are less than the pivot are placed before the pivot and all elements greater than the pivot after it.

- Equal values can be placed on either of the two sides of the list.

Contd…

iii. We have the pivot in the final sorted position.

This partition operation completes the first pass of the sort operation.

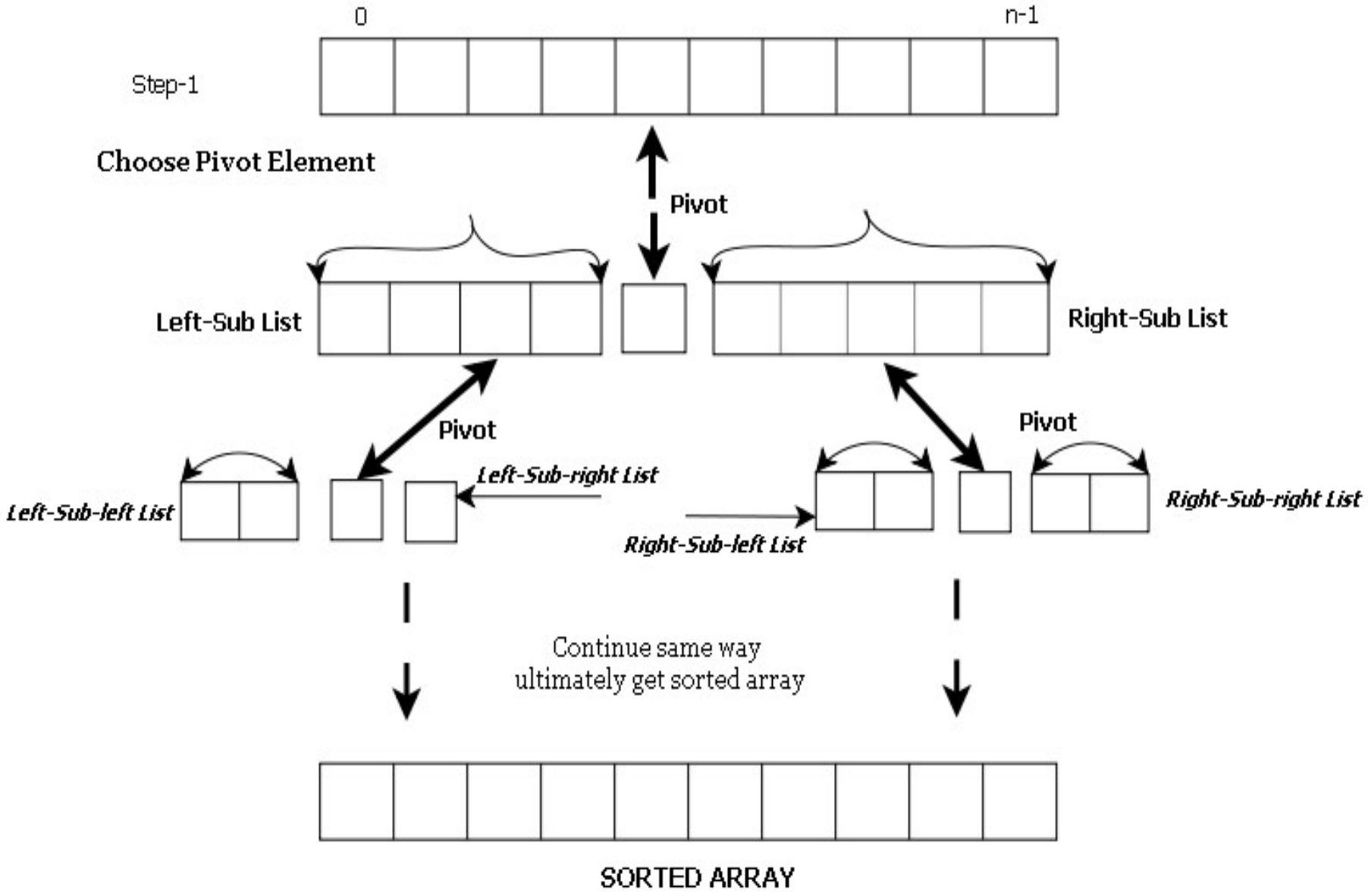
iv. Repeat step 1 to step 3 to recursively sort the two sub arrays obtained after the first partition.

Contd…

- The most complex issue in this algorithm is choosing a good pivot element.
- A poor choices of pivot element can result in drastically slower $O(n^2)$ performance.
- If at each step we choose the median as the pivot then it works in O(n log n).

Algorithm

- We require two iterators say ‘i’ and ‘j’ to iterate over the list.
- ‘i’ starts from first element and moves forward in the list.
- ‘j’ is positioned at last element and moves backwards.
- To begin with, we place the pivot at one end of the list.
- Move ‘i’ to the right till we find a number greater than pivot.
- Move ‘j’ to the left till we find a number smaller than pivot.
- Swap numbers at these two positions.
- **If (i < j),** we swap the pivot with element at i^{th} position.



Data Structures: A Programming Approach with C, PHI
Dharmender Singh Kushwaha & A.K.Misra

Example

Sort the array [19, 12, 15, 29, 11, 14, 16, 13, 18 17].

Choosing the pivot element.

Left-most element is at position = 0, so L[left-most] = 19.

Right-most element is at position = 9, so L[right-most] = 17.

Centre = (left-most + right-most) / 2 = (0 + 9) = 4.

The element at index 4 is 11.

Pivot = Median of left-most, right-most, and centre.

Median of 19, 17 & 11 is 17.

This is chosen as pivot element.

	19 i	12	15	29	11	14	16	13	18 j	17 pivot	
move	19 i	12	15	29	11	14	16	13 j	18	17 pivot	
swap	13 i	12	15	29	11	14	16	19 j	18	17 pivot	
move	13	12	15	29 i	11	14	16 j	19	18	17 pivot	
swap	13	12	15	16 i	11	14	29 j	19	18	17 pivot	
move	13	12	15	16	11	14 j	29 i	19	18	17 pivot	i and j have crossed
swap S[i] with pivot	13	12	15	16	11	14	17 j	19 i → pivot	18	29	

Data Structures: A Programming Approach with C, PHI
Dharmender Singh Kushwaha & A.K.Misra

Quick Sort Analysis

- The other sorting technique that is comparable to quick sort is heapsort.
- Heapsort is somewhat slower than quicksort, but the worst case running time is always O(n logn).
- Quicksort is usually faster, though there are chances of worst case performance.
  - This happens when the list is sorted and the left-most element is chosen as the pivot.
  - The worst-case efficiency of the quick sort $O(n^2)$ is encountered in this kind of a scenario.

Radix Sort

- It sorts items by scanning individual digits.
- Does not involve comparison between the items being sorted.
- It makes use of key to shuffle the items.
- For integer data items, the key is each individual digit at each decimal place in that item for a specific pass.
- We start with the least significant digit (LSD) to the most significant digit (MSD).
- Since integers can represent strings of characters it is not limited to integers.

Radix Sort

Pseudo code

1.Consider digits at different decimal place (i.e. at units, tens, hundreds, and thousand and so on) in different passes.

- i.e. for Pass-I consider only, digits at units place of each item.
- i.e. for Pass-II consider only, digits at tens(10^{th}) place of each item and so on.

2. In each pass, put all the items with same index value at that decimal place, in a single bin, without changing their order.

3. Repeat procedure in step 2 next decimal place.

3. Continue with these steps till all the decimal places are covered.

Radix Sort

169	499	711	158	535	507	849	388

PASS-I

1. Considers item at units place.

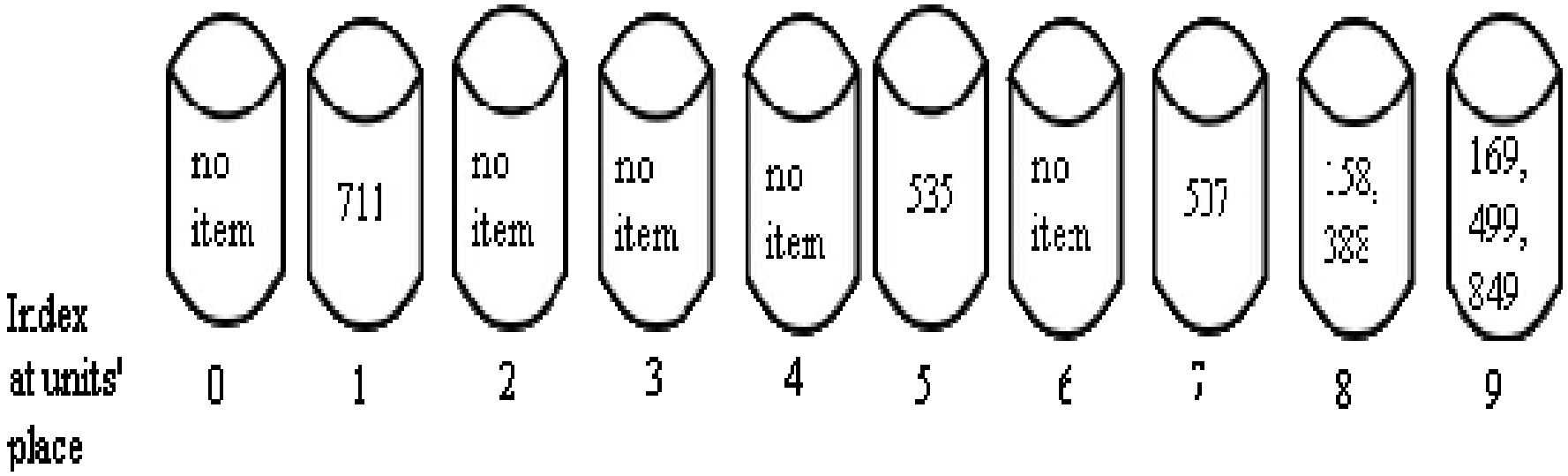
16**9**	49**9**	71**1**	15**8**	53**5**	50**7**	84**9**	38**8**

2. Take 10 bins for each of the digit.

3. There is no item at units place with index value 0, 2,3,4,6. →

3. Items with index 0 at units place are pushed in bin with index 0 and so on till 9.

Radix Sort


no item
711
no item
no item
no item
535
no item
537
358, 388
169, 499, 849
Index at units' place
0
1
2
3
4
5
6
7
8
9

Radix Sort

Index at units place(i)→	0	1	2	3	4	5	6	7	8	9
Count(C)→	0	1	0	0	0	1	0	1	2	3

- Sorted list of items according to units place after Pass-I:

71**1**	53**5**	50**7**	15**8**	38**8**	16**9**	49**9**	84**9**

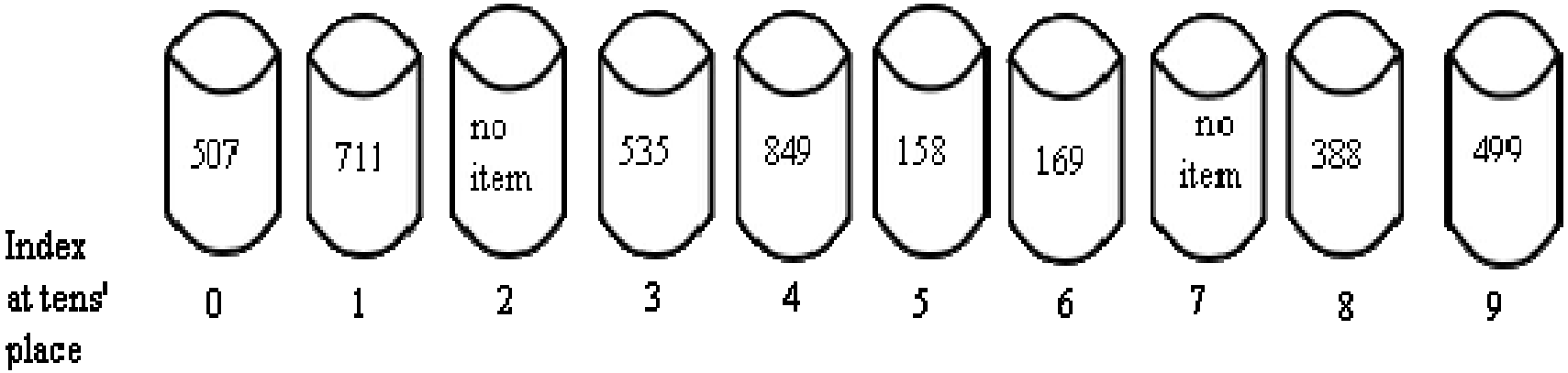
PASS-II

Considers item at tens place.

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

Radix Sort

- After second pass we have



- Item after pass-II sorted according to tens (10^{th}) place

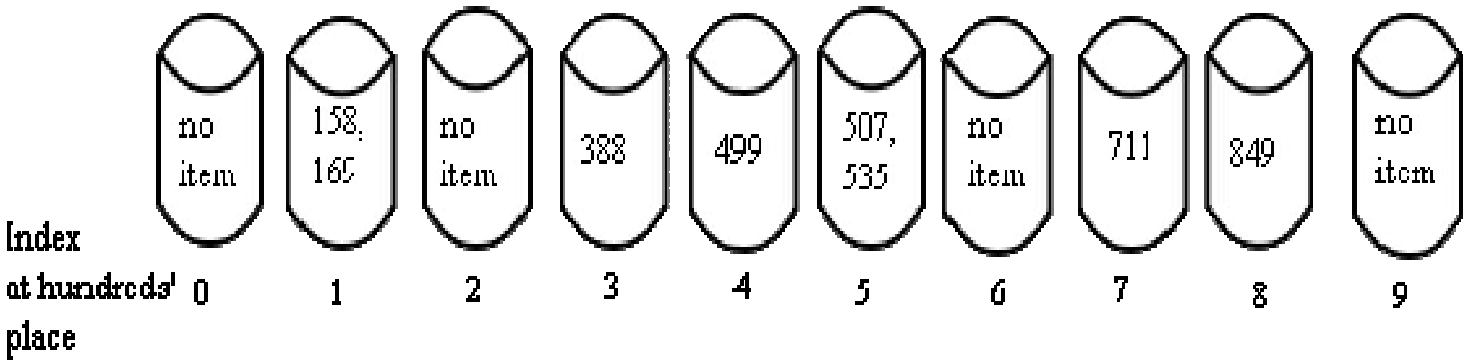
5**0**7	7**1**1	5**3**5	8**4**9	1**5**8	1**6**9	3**8**8	4**9**9

Radix Sort

PASS-III

Considers item at hundreds place.

507	711	535	849	158	169	388	499



Radix Sort

- Item after pass-III sorted according to hundreds (100^{th}) place

158	**1**69	**3**88	**4**99	**5**07	**5**35	**7**11	**8**49

Complexity

- Complexity of radix sort is O (n), where n is the number of items i.e. linear time complexity.

Summary

- Bubble sort works by repeatedly heading through the stored item to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.
- Selection sort is designed to sort the elements in such a way that after every pass one element should reach its suitable position.
- Insertion sort inserts an element one by one and help that number to reach its desired position according to the order sorting.

Summary

- Heap sort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the partially sorted array.
- Quick sort works by dividing the list into two sublists.
- Merge sort is merging two sorted arrays in a third array not violating the order of sorting.
- Bucket sort works by partitioning an array into a number of buckets. Each bucket is then sorted individually using a different sorting algorithm recursively.
- Radix sort is a sorting algorithm that sorts items by scanning individual digits. It does not involve comparison between the items being sorted.