

CHAPTER 7: SORTING

BIBHA STHAPIT

ASST. PROFESSOR

IoE, PULCHOWK CAMPUS

Introduction

- Arrangement of data in some systematic order is called sorting
- Sorting can be done based on some key reference
- The key reference can be numbers, alphabets etc.
- Sorting is essential part of data management, since it helps in searching the data
- Sorting can be done in two different orders:
 - Ascending Order (Increasing Order)
 - $D1 \leq D2 \leq D3 \leq \dots \leq Dn$
 - Descending Order (Decreasing Order)
 - $D1 \geq D2 \geq D3 \geq \dots \geq Dn$

2,5,4,1,8,6 => Unsorted Data

1,2,4,5,6,8 => Sorted Data (Ascending Order)

8,6,5,4,2,1 => Sorted Data (Descending Order)

Types of sorting

- Sorting can be categorized into different categories:
- **1. Internal and external sorting**
 - Internal Sorting
 - The sorting done within the computer main memory is called internal sorting
 - It is performed on data items small enough to easily fit within the main memory
 - External Sorting
 - The sorting is done in external file disk, i.e. secondary memory
 - It is performed if the data to be sorted does not fit within computer main memory

Types of sorting

- **2. In-place and non-in-place sorting**
- In-place sorting
 - Sorting can be done within array itself
 - E.g. Bubble sort
- Non-in-place sorting
 - It requires more space/array (more than or equal to number of elements)
 - E.g. Merge sort

Types of sorting

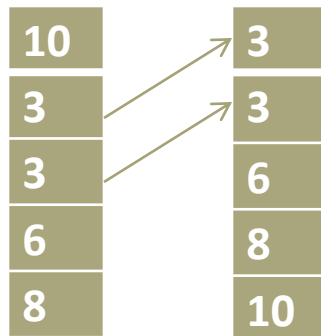
- **3 . Stable and not-stable sorting**

- Stable sorting

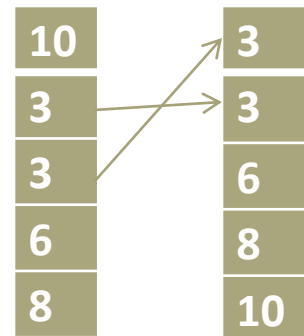
- After sorting, if it doesn't change the sequence of similar content in which they appear.

- Not-stable sorting

- After sorting, if it changes the sequence of similar content in which they appear.



Stable



Not-Stable

Types of sorting

- **4. Adaptive and Non-adaptive sorting**
- Adaptive sorting
 - Takes advantage of already sorted elements
 - Will not try to re-order the previous list
- Non-adaptive sorting
 - Forces every single element to be reordered to confirm their sortedness

Sorting Efficiency

- Different sorting algorithms are known
- Choice of the correct sorting algorithm based on the requirement and data items to be sorted
- Efficiency of the algorithms depend upon mainly two factors
 - Time Complexity:
 - Amount of machine time the algorithm takes to sort the data
 - Depends mainly upon numbers of comparisons and number of moves
 - Space Complexity:
 - Amount of memory space required by the program during sorting
 - Depends mainly upon the programming code for the algorithm and working space required by the program during calculations

Sorting Algorithms


- Some algorithms to be discussed in this chapter are:
 - Bubble Sort
 - Quick Sort
 - Selection Sort
 - Insertion Sort
 - Shell Sort
 - Heap Sort
 - Merge Sort
 - Radix Sort

Bubble Sort

- a simple sorting algorithm which
 - repeatedly steps through the list to be sorted
 - compares each pair of adjacent items
 - swaps them if they are in the wrong order
 - The passing through the list is continued until the swapping is not required (i.e. the list sorted)
- it is a comparison sort
- It is called Bubble Sort because the data gradually bubbles up in its proper position
- In each pass at least one data is bubbled up in its proper position

Bubble sort

Pass	Comparison	Resultant Array
1	<div>18 3 2 33 21</div> <div>3 18 2 33 21</div> <div>3 2 18 33 21</div> <div>3 2 18 33 21</div>	<div>3 2 18 21 33</div>
2	<div>3 2 18 21 33</div> <div>2 3 18 21 33</div> <div>2 3 18 21 33</div>	<div>2 3 18 21 33</div>
3	<div>2 3 18 21 33</div> <div>2 3 18 21 33</div>	<div>2 3 18 21 33</div>
4	<div>2 3 18 21 33</div>	<div>2 3 18 21 33</div>

 denotes the pair of consecutive elements being compared

Bubble Sort

6 5 3 1 8 7 2 4

Bubble Sort (Algorithm-pseudocode)

Declare and Initialize necessary variables

$n \Rightarrow$ number of data items

$a[n] \Rightarrow$ an array holding all the data items to be sorted

BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For $I = 0$ to $N-1$

Step 2: Repeat For $J = 0$ to $N-I$

Step 3: IF $A[J] > A[J + 1]$ // for ascending order

 SWAP $A[J]$ and $A[J+1]$

 [END OF INNER LOOP]


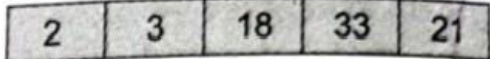
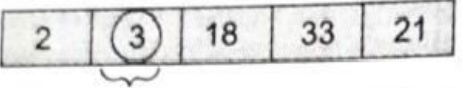
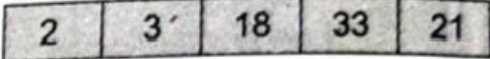
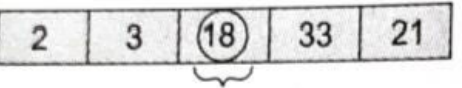
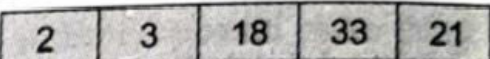
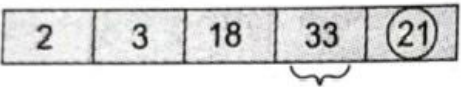
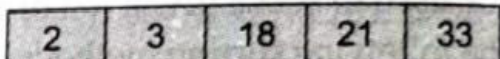
 [END OF OUTER LOOP]



Step 4: EXIT

Selection Sort

- One of the simplest algorithm
- In each pass the smallest/largest value is selected and placed as the leftmost data item
- Now the selection is done discarding the value just selected and thus selected new data is placed as the second leftmost one
- The list on the left side thus is the sorted list and the one in right side is the unsorted one
- A list of n elements requires $n-1$ passes to completely rearrange the data
- The last pass requires only one comparison between the last two elements remaining

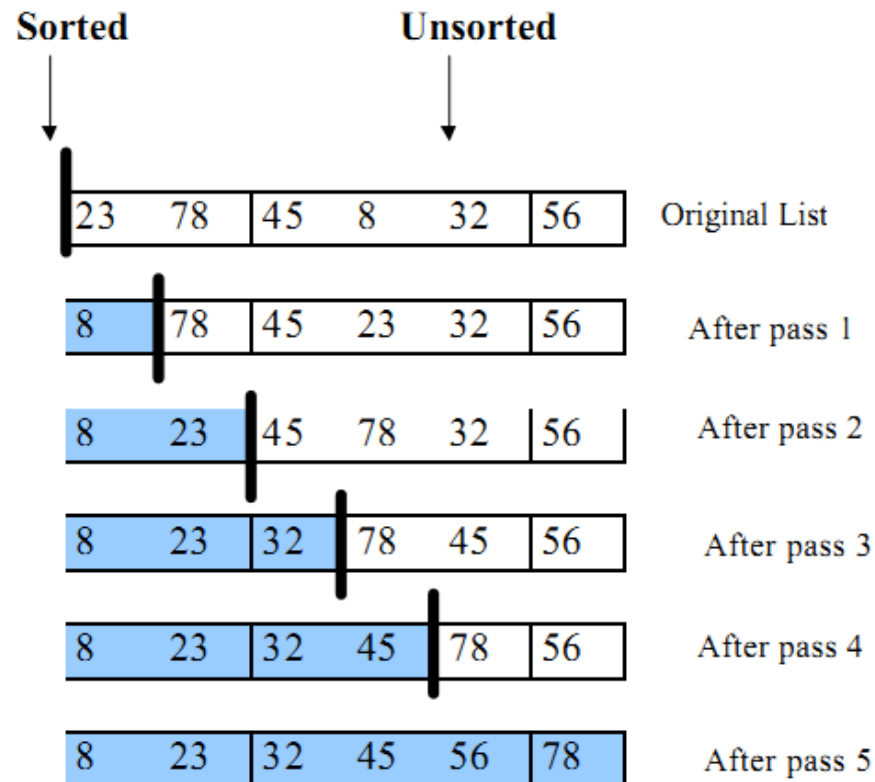
Selection sort

Pass	Comparison	Resultant Array
1		
2		
3		
4		

 → denotes the currently selected element
 → denotes the smallest element identified in the current pass

Selection Sort

- Consider an example:
 - 23 78 45 8 32 56 (six numbers, five passes required)



Selection Sort

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Legend:

Red: Current Minimum

Yellow: Sorted Items

Blue: Current Item

Selection Sort (Algorithm-pseudocode)

SELECTION_SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K=1 to N-1

Step 2: CALL SMALLEST(ARR, K, N, POS)

Step 3: SWAP ARR[K] with ARR[POS]

[END OF LOOP]

Step 4: EXIT

SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]

Step 2: [INITIALIZE] SET POS = K

Step 3: Repeat for J = K+1 to N

 IF SMALL > ARR[J]

 SET SMALL = ARR[J]

 SET POS = J

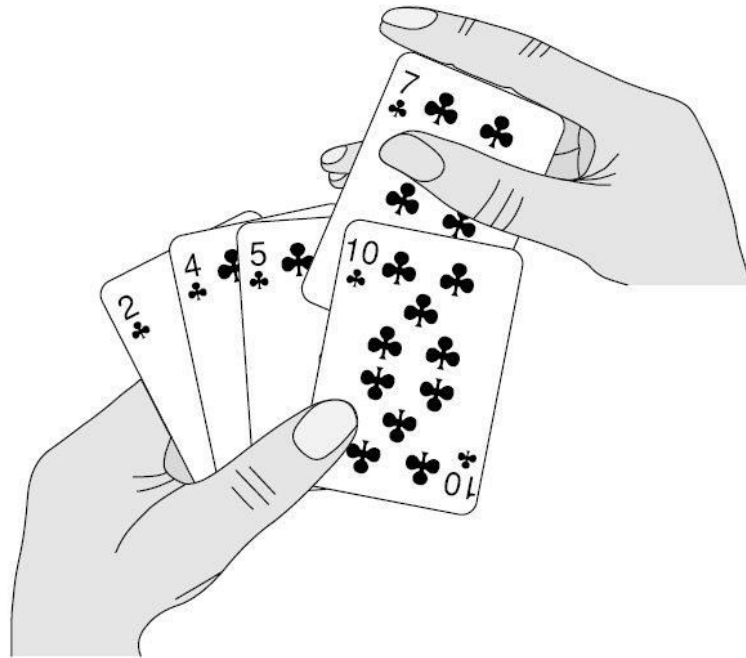
 [END OF IF]

[END OF LOOP]

Step 4: RETURN POS

Insertion Sort

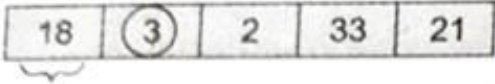
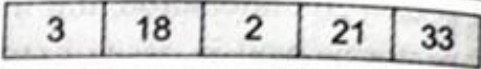
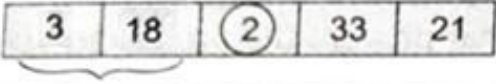
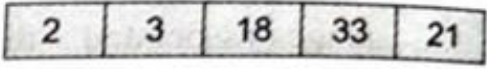
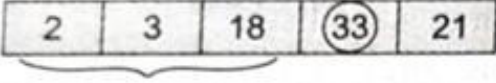
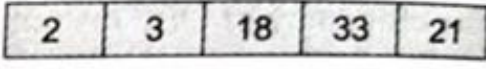
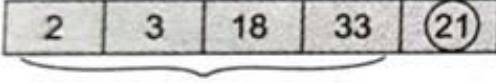
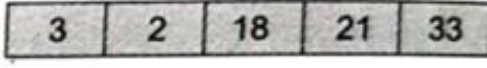
- Its similar to arrangement of cards during card game





Insertion Sort

- Insertion sort is implemented by inserting a particular data item in its proper position
- Any unsorted data item is kept on swapping with its previous data items until its proper position is not found
- The number of swapping makes the previous data items to shift for the new data item to take its position in order
- Once the new data item is inserted, the next data item after it is chosen for next insertion
- The process continues until all data items are sorted
- This method is highly efficient if the list is almost in sorted form

Insertion sort

Pass	Comparison	Resultant Array
1		
2		
3		
4		

 → denotes the previously sorted sub array
 → denotes the current selection

Insertion Sort

40 (15) 30 5 25 10 20 35
↑

15 40 (30) 5 25 10 20 35
↑

15 30 40 (5) 25 10 20 35
↑

5 15 30 40 (25) 10 20 35
↑

5 15 25 30 40 (10) 20 35
↑

5 10 15 25 30 40 (20) 35
↑

5 10 15 20 25 30 40 (35)
↑

5 10 15 20 25 30 35 40

Insertion Sort

6 5 3 1 8 7 2 4

Insertion Sort (Algorithm-pseudocode)

INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for $K = 1$ to $N-1$

Step 2: SET TEMP = ARR[K]

Step 3: SET $J = K - 1$

Step 4: Repeat while TEMP \leq ARR[J]

 SET ARR[J + 1] = ARR[J] //SWAP

 SET $J = J - 1$

[END OF INNER LOOP]

Step 5: SET ARR[J + 1] = TEMP

[END OF LOOP]

Step 6: EXIT

Quick Sort

- Also called partition-exchange sort
- Uses divide and conquer algorithm
- One pivot element is chosen from within the list
- The list is divided into two partition
 - All values less than the pivot are placed on left side of pivot
 - All greater values are placed on right side of the pivot
- After a single pass, the pivot is in its proper position
- The left and right partitions are sorted recursively using the same method joining the left sorted, pivot and right sorted results with the list in sorted order

Quick Sort

6 5 3 1 8 7 2 4

Choose pivot

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

start from left and right values

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

move left until left > pivot
move right until right < pivot

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

swap left and right value

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

repeat same steps until left > right

Chapter 7

4	2	6	5	3	9
---	---	---	---	---	---

4	2	6	5	3	9
↑ L					↑ R

4	2	6	5	3	9
---	---	---	---	---	---

L

R

4	2	6	5	3	9
---	---	---	---	---	---

L

R

4	2	6	5	3	9
---	---	---	---	---	---

L

R

A diagram showing a 6-element array: [4, 2, 3, 5, 6, 9]. The elements are in boxes. The boxes for 3 and 6 are blue, while the others are white. Below the array, two arrows point upwards: 'L' points to the box containing 3, and 'R' points to the box containing 6.

4	2	3	5	6	9
---	---	---	---	---	---

↑ ↑
L R

4	2	3	5	6	9
---	---	---	---	---	---

↑ ↑
R L

Quick Sort

6 5 3 1 8 7 2 4

Quick Sort(Algorithm-pseudocode)

PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0

Step 2: Repeat Steps 3 to 6 while FLAG = 0

Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT

 SET RIGHT = RIGHT - 1

 [END OF LOOP]

Step 4: IF LOC = RIGHT

 SET FLAG = 1

 ELSE IF ARR[LOC] > ARR[RIGHT]

 SWAP ARR[LOC] with ARR[RIGHT]

 SET LOC = RIGHT

 [END OF IF]

Step 5: IF FLAG = 0

 Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT

 SET LEFT = LEFT + 1

 [END OF LOOP]

Step 6: IF LOC = LEFT

 SET FLAG = 1

 ELSE IF ARR[LOC] < ARR[LEFT]

 SWAP ARR[LOC] with ARR[LEFT]

 SET LOC = LEFT

 [END OF IF]

[END OF IF]

Step 7: [END OF LOOP]

Step 8: END

Quick Sort(Algorithm-pseudocode)

QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)

 CALL PARTITION (ARR, BEG, END, LOC)

 CALL QUICKSORT(ARR, BEG, LOC - 1)

 CALL QUICKSORT(ARR, LOC + 1, END)

[END OF IF]

Step 2: END

Shell Sort

- Generalization of insertion sort
- Better form of insertion sort
- Also called diminishing increment sort
 - A value for a gap(or increment) is chosen (convention: $\text{gap} = n/2$)
 - Sub-lists are made from the original list taking items from every gap values
 - The sub-lists thus formed are sorted first
 - In second pass smaller value for the gap is chosen ($\text{gap} = \text{gap}/2$)
 - The next sub-lists are formed and sorted
 - The process is repeated until gap value is less than 1; the list is sorted by the end of this pass

Shell Sort (Example)

- Lets see an example
 - 25, 57, 48, 37, 12, 92, 86, 33
 - We choose gap size of 5, 3, and 1

Original list 25, 57, 48, 37, 12, 92, 86, 33

Pass 1
Gap=5

25, 57, 48, 37, 12, 92, 86, 33



Pass 2
Gap=3

25, 57, 33, 37, 12, 92, 86, 48



Pass 3
Gap=1

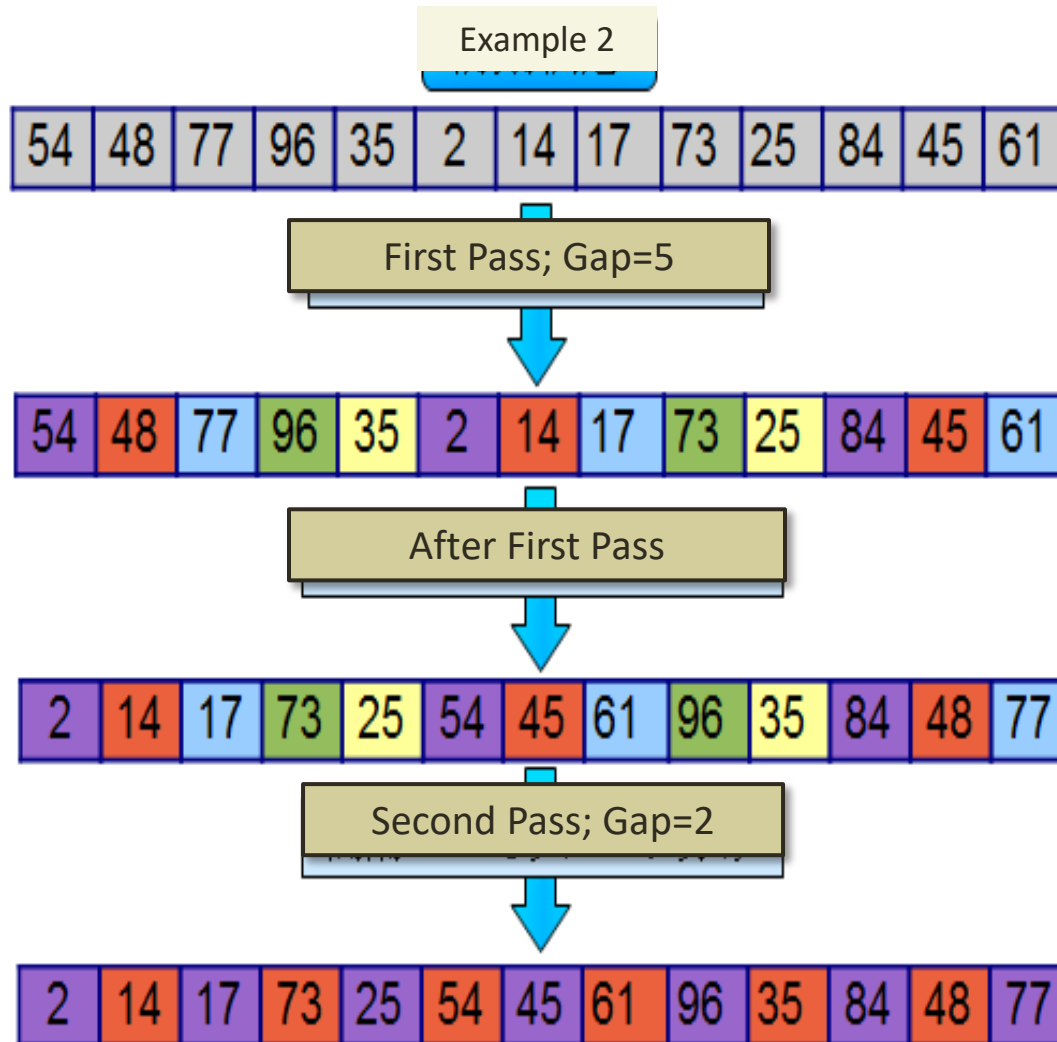
25, 12, 33, 37, 48, 92, 86, 57



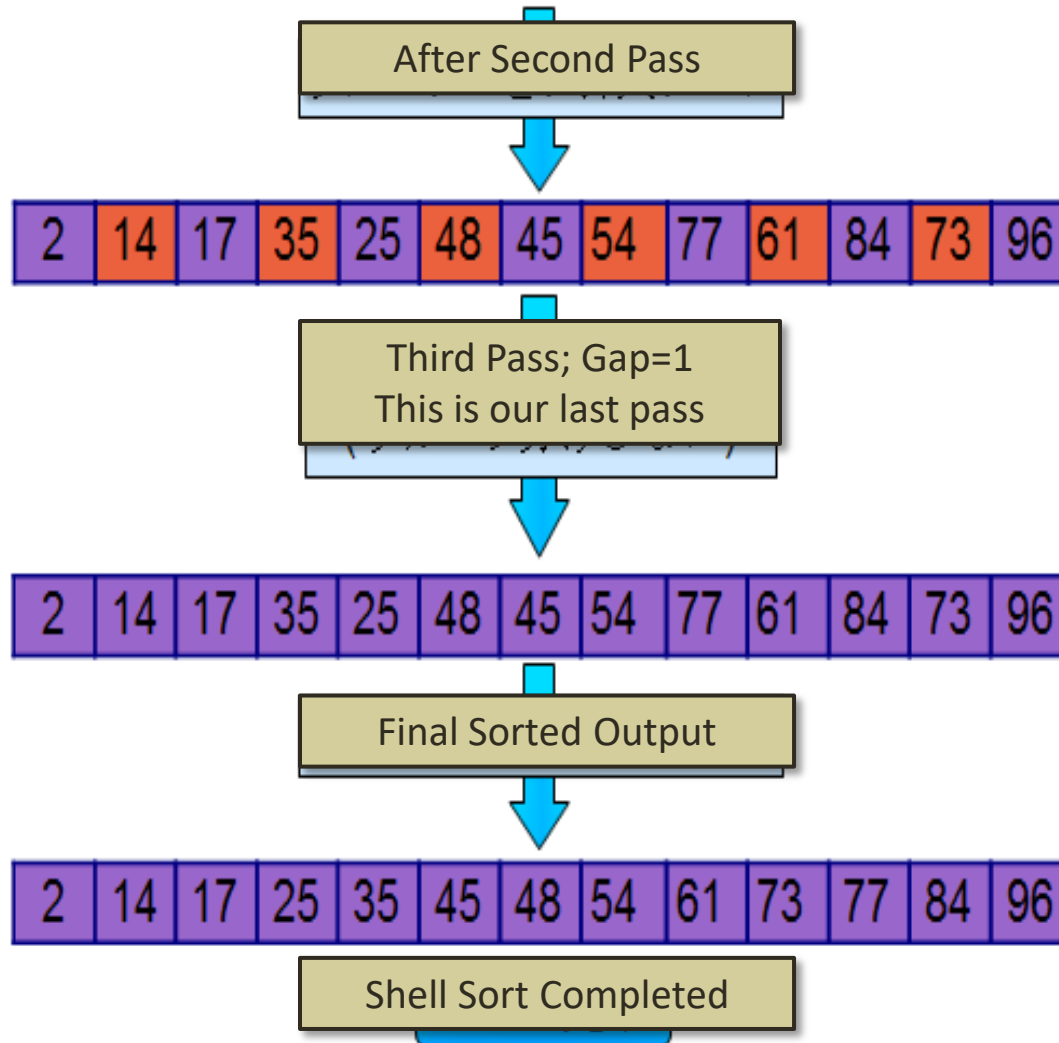
Sorted
result

12, 25, 33, 37, 48, 57, 86, 92

Shell Sort (Example)



Shell Sort (Example- contd..)



Shell Sort (Algorithm-Pseudocode)

Shell_Sort(Arr, n)

Step 1: SET FLAG = 1, GAP_SIZE = N

Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1

Step 3: SET FLAG = 0

Step 4: SET GAP_SIZE = (GAP_SIZE + 1) / 2

Step 5: Repeat Step 6 for I = 0 to (N-GAP_SIZE-1)

Step 6: IF Arr[I + GAP_SIZE] > Arr[I]

 SWAP Arr[I + GAP_SIZE], Arr[I]

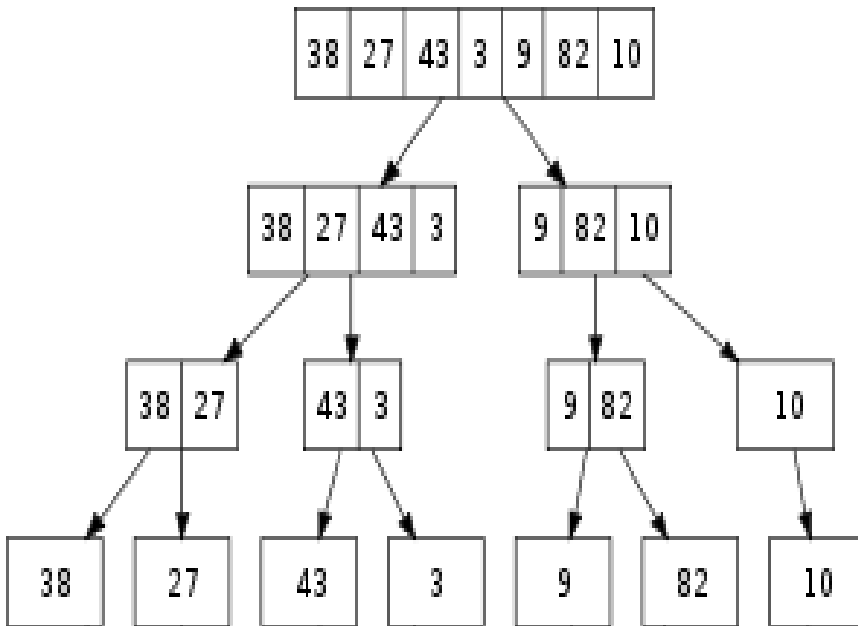
 SET FLAG = 1

Step 7: END

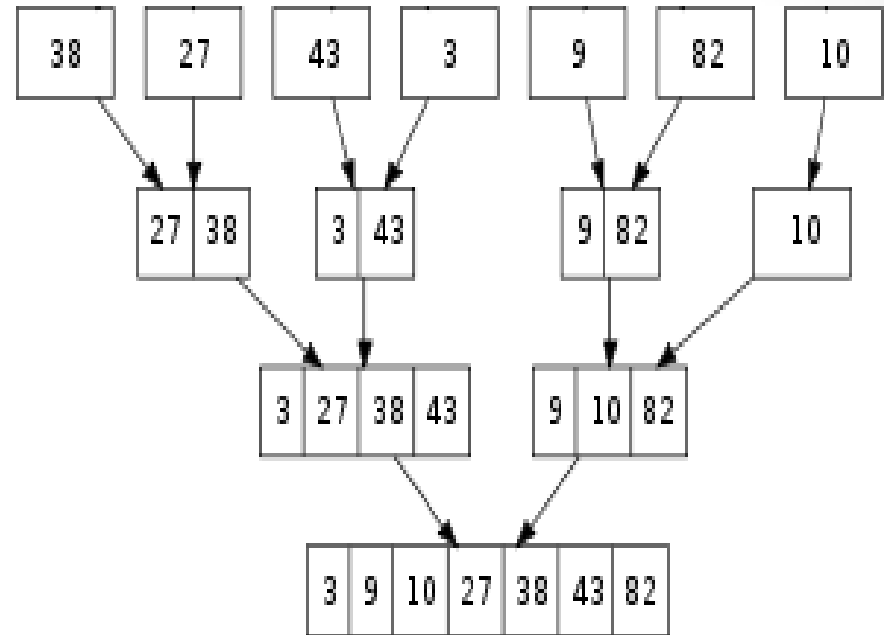
Merge Sort

- It is a divide and conquer algorithm
- At first we divide the given list of item
 - list is divided into two parts from middle
 - The process is repeated until each sub-list contain exactly 1 item
- Now is the turn for sort and combine (conquer)
 - A list with a single element is considered sorted automatically
 - Pair of list is sorted and merged into one (i.e. approx. $n/2$ sublists of size 2)
 - The sort and merge is keep on repeated until a single list of size n is found
- The overall dividing and conquering is done recursively

Merge Sort



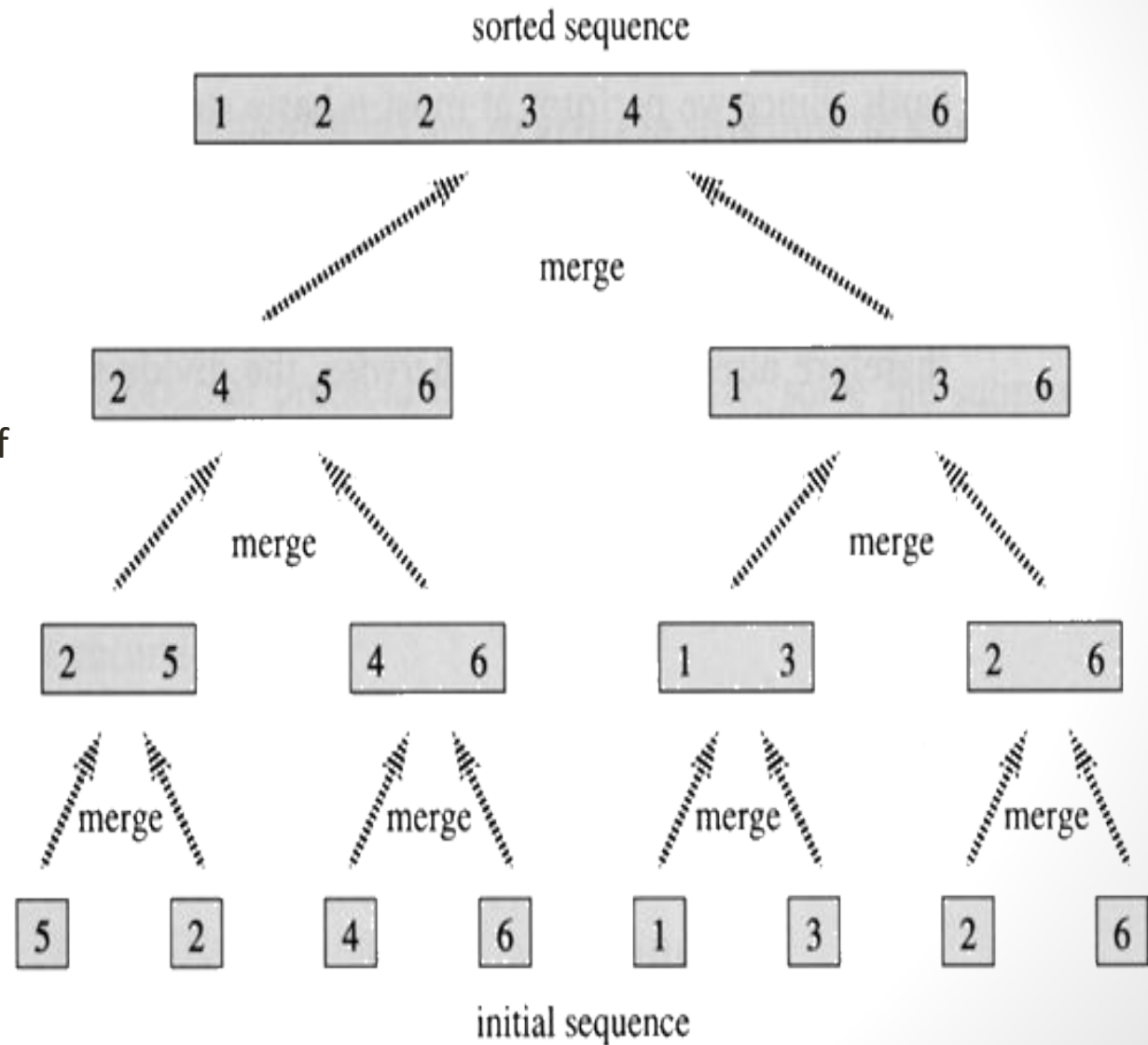
Divide



Conquer

Merge Sort

Bottom-Up approach of
merge after
completion of division



Merge Sort

6 5 3 1 8 7 2 4

Merge Sort

- To sort $A[p .. r]$:

1. Divide Step

- If a given array A has zero or one element, simply return; it is already sorted.
- Otherwise, split $A[p .. r]$ into two sub-arrays $A[p .. q]$ and $A[q + 1 .. r]$, each containing about half of the elements of $A[p .. r]$. That is, q is the halfway point of $A[p .. r]$.

2. Conquer Step

- Conquer by recursively sorting two subarrays $A[p .. q]$ and $A[q + 1 .. r]$.

3. Combine Step

- Combine the elements back in $A[p .. r]$ by merging the two sorted subarrays $A[p .. q]$ and $A[q + 1 .. r]$ into a sorted sequence.
- To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Merge Sort (Algorithm)

Declare and initialize necessary variables

n-total number of elements in an array

a[n]-array containing data

p=0, r=n-1; first and last index of the array

MERGE-SORT (A, p, r)

1. IF $p < r$ // Check for base case
2. THEN $q = (p + r)/2$ // Divide step
3. MERGE-SORT (A, p, q) // Conquer step.
4. MERGE-SORT ($A, q + 1, r$) // Conquer step.
5. MERGER (A, p, q, r) // combine step.

Merge Sort (Algorithm-pseudocode)

MERGE_SORT(ARR, BEG, END)

Step 1: IF BEG < END

 SET MID = (BEG + END)/2

 CALL MERGE_SORT (ARR, BEG, MID)

 CALL MERGE_SORT (ARR, MID + 1, END)

 MERGE (ARR, BEG, MID, END)

 [END OF IF]

Step 2: END

MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0

Step 2: Repeat while (I <= MID) AND (J <= END)

IF ARR[I] < ARR[J]

SET TEMP[INDEX] = ARR[I] , SET I = I + 1

ELSE

SET TEMP[INDEX] = ARR[J] , SET J = J + 1

[END OF IF]

SET INDEX = INDEX + 1

[END OF LOOP]

Step 3: [Copy the remaining elements of right sub-array, if any]

IF I > MID

Repeat while J <= END

SET TEMP[INDEX] = ARR[J] , SET INDEX = INDEX + 1, SET J = J + 1

[END OF LOOP]

[Copy the remaining elements of left sub-array, if any]

ELSE

Repeat while I <= MID

SET TEMP[INDEX] = ARR[I] , SET INDEX = INDEX + 1, SET I = I + 1

[END OF LOOP]

[END OF IF]

Step 4: [Copy the contents of TEMP back to ARR] SET K = 0

Step 5: Repeat while K < INDEX

SET ARR[K] = TEMP[K] , SET K = K + 1

[END OF LOOP]

Step 6: END

Radix Sort

- Non-comparative integer sorting algorithm
- Queues are made for representing different key values
- In each queue sub-lists from the original list are formed depending on the key values
- The data to be sorted in the given list is checked starting from any of the position(LSD/MSD) in data
- If the starting position is LSD – Least Significant Digit, then the sorting is called LSD Radix Sort
- If the starting position is MSD – Most Significant Digit, then the sorting is called MSD Radix Sort
- The corresponding LSD or MSD value determines the queue for the data items

Radix Sort

- All the values are made of same length by adding 0s as MSDs if any of the values is shorter than others
- If LSD is being used, one's position are checked first and sub-lists are formed
- Sub-lists are now gathered into main list in order of the key values in queues
- Thus formed list is now again checked for ten's place and sub-lists are formed on queues
- The gathering and forming of sub-lists is continued for other higher placed digit until every queue contains at most one data items
- Now the final gathering results with the sorted list

Radix Sort

- LSD Radix Sort is mainly used for sorting Numeral values
- MSD Radix Sort is mainly used for sorting Alphabetical values
- Its main advantage is that the computers represent data in binary form which can be sorted easily by using radix sort
- Its main disadvantage is that it requires more space than other sorting algorithms

Radix Sort (Example)

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits:

Digit	Sublist
0	710 340
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

710 340 812 582 493 715 195 385 437

Radix Sort (Example)

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

710 340 812 582 493 715 195 385 437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Radix Sort (Example)

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

Radix Sort (Algorithm-pseudocode)

RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE

Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE

Step 3: SET PASS = 0

Step 4: Repeat Step 5 while PASS \leq NOP-1

Step 5: SET I= and INITIALIZE buckets

Step 6: Repeat Steps 7 to 9 while I $<$ N-1

Step 7: SET DIGIT = digit at PASSth place in A[I]

Step 8: Add A[I] to the bucket numbered DIGIT

Step 9: INCREMENT bucket count for bucket numbered DIGIT

[END OF LOOP]

Step 10: Collect the numbers in the bucket

[END OF LOOP]

Step 11: END

Heap

- Heaps are almost complete binary tree, where
 - Content of each node is less than or equal to that of its father
 - All levels are full except the lowest level
 - If lowest level is not full, then nodes must be packed to left
 - The operations performed in a heap tree are: insertion and deletion
 - Insertion always occurs in empty left child node
 - Deletion occurs from root node
- Note: There should not be any gap in a heap tree

Operations in Heap

- **Insertion in the heap**
 - Add new element to the lowest available level
 - Check the newly inserted value with the value in its parent node
 - If the parent value is less than the child node
 - Then interchange the values of child and parent
 - Repeat the comparison and interchange with other ascending nodes for the condition above
 - Keep on repeating the above processes until all data are inserted
- Note: the process of changing the node with its parent is called **heapup**

Operations in Heap

- **Deletion from heap**
 - Remove the root element
 - Move the last node onto the root node
 - Check the value with child node and swap until it gets its proper position
 - During swapping the parent node is swapped with the greater child among the two children
 - Note: Process of changing the node with its child is called **heapdown**

Heap Sort

- Comparison based sorting algorithm
- One of the types of selection sort
- It is an in-place algorithm
- It is not a stable sort
- Heap sort algorithm is divided into two-parts
 - Building of heap
 - Sorting of data in an array from the heap

Heap Sort

- The two steps for sorting are:
 - In the first step,
 - a heap is built out of the data.
 - In the second step,
 - a sorted array is created by using following technique
 - repeatedly remove the largest element from the heap,
 - Insert it into the array.
 - Reconstruct the heap after each removal.
 - Once all objects have been removed from the heap, we have a sorted array.

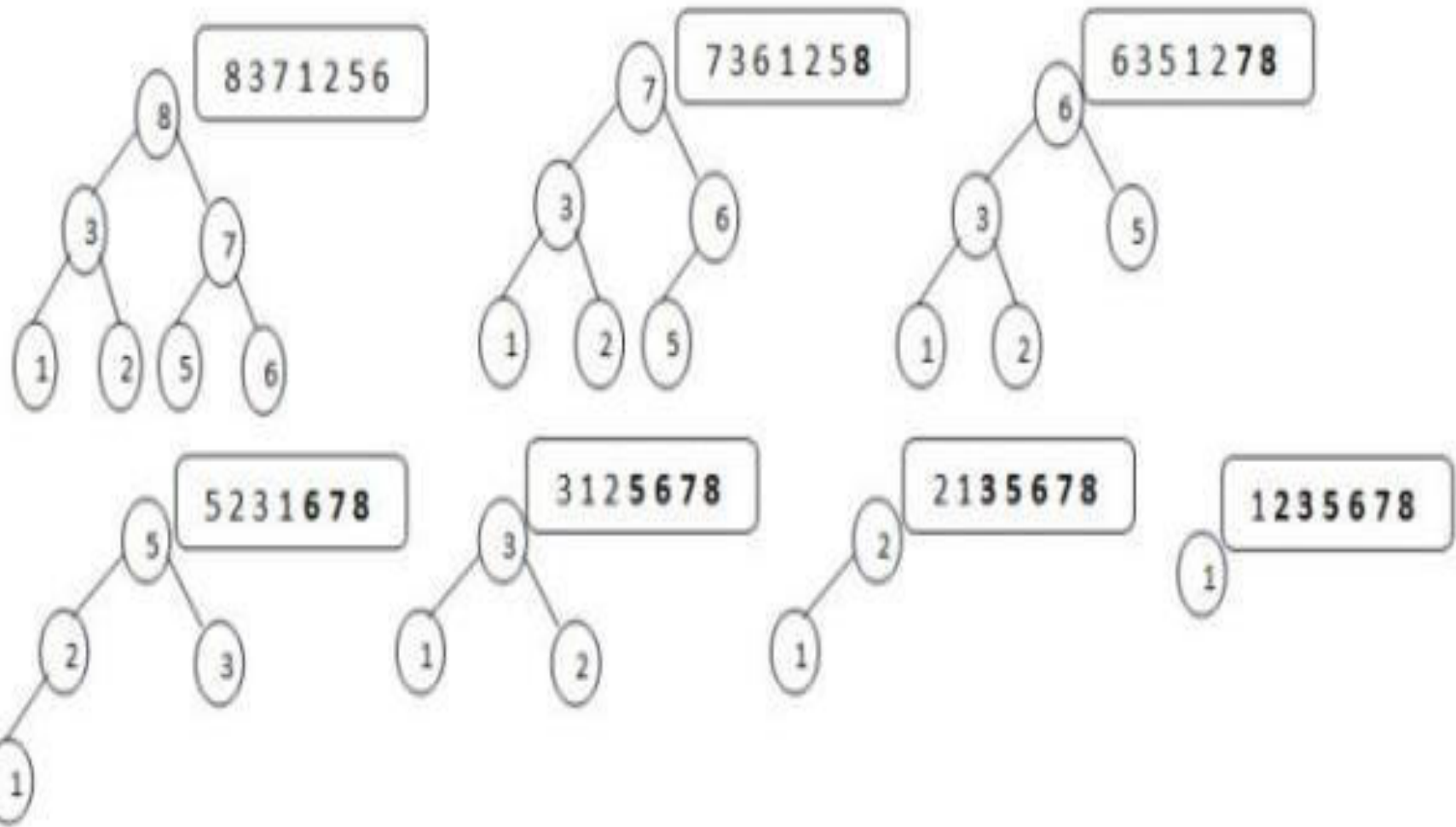
Heap Sort (Algorithm)

1. Construct a heap using the data from the list by inserting every data sequentially
2. After the completion of heap construction, create an array with its bound equal to number of elements in the heap
3. Remove root value from heap and insert it into array
4. Insert the value of last node into the place of root
5. Reconstruct the heap by arranging the data (reheapdown)
6. Continue the above steps until all nodes are deleted from heap and inserted into the array
7. Now the array contains the result in sorted form

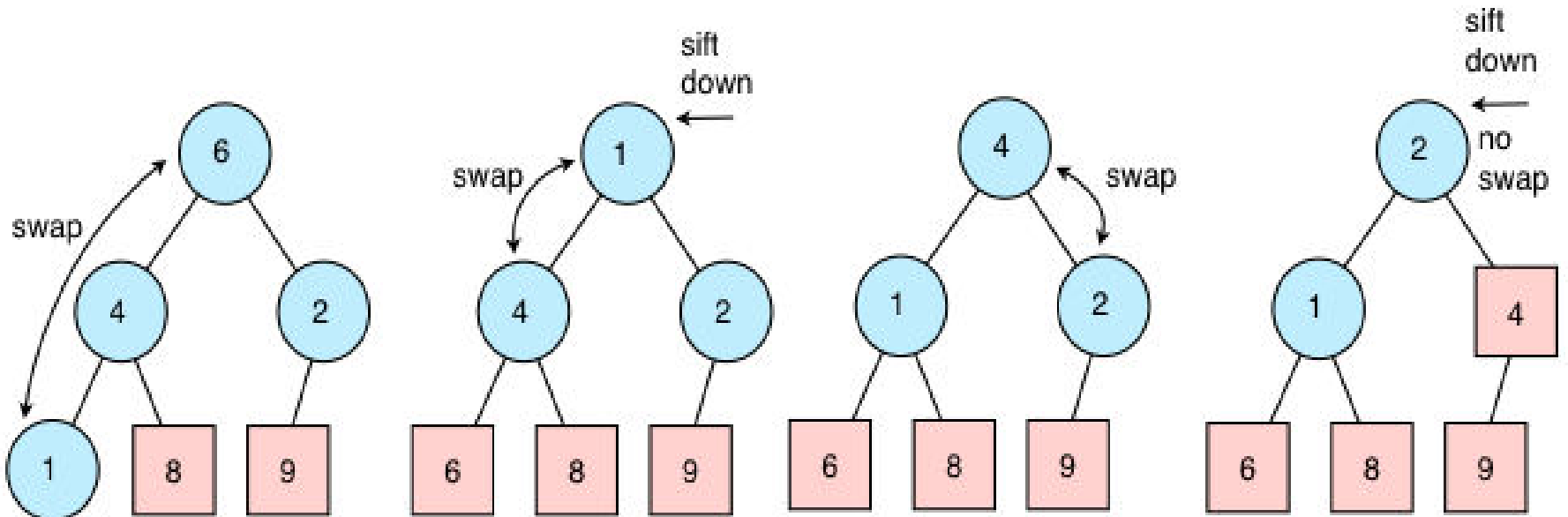
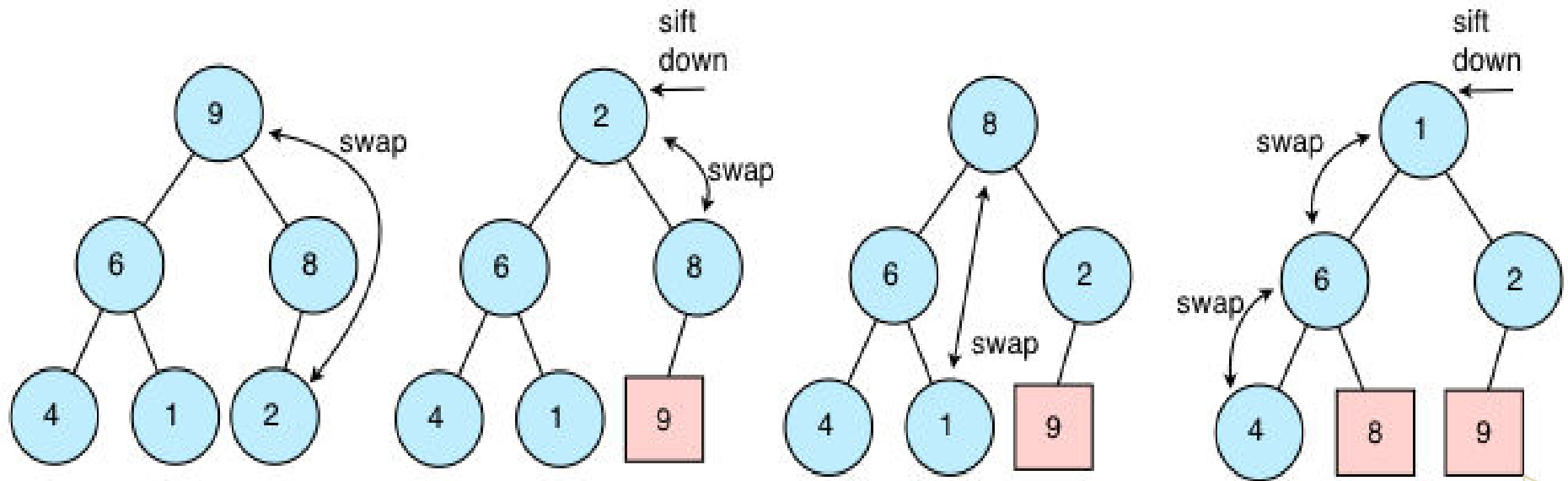
Heap Sort (Example)

6 5 3 1 8 7 2 4

Heap Sort



Heapsort



Efficiency of sorting algorithms

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Quick Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Radix Sort	$\Omega(nk)$	$\theta(nk)$	$O(nk)$

THANK YOU!