Basic Data Structures

Unit 6 : Searching and Sorting

Prepared by :

Prof. Khushbu Chauhan

Computer Engg. Dept.

MPSTME, NMIMS

Outlines

- Basic search techniques
- Sequential search
- Binary search
- Sorting definitions
- Bubble Sort
- Selection Sort
- Insertion Sort
- Radix Sort

Introduction to Searching

- Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
- There are two popular methods for searching the array elements:
  - Linear/Sequential search
  - Binary search.

Applications

- Location of data for retrieval
- Modification or verification of data For example
- Find a name in a phone book
- A number in a sorted array
- A path in a maze or a keyword in a document
- Telephone directory for searching a right name
- Library book search
- Page numbers search

Linear Search

- Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).
- For example, if an array A[] is declared and initialized as,

```
int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
```

- The value to be searched is VAL = 7, then searching means to find whether the value ‘7’ is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).
- Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made. Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made. However, the performance of the linear search algorithm can be improved by using a sorted array

Algorithm for linear search

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:             IF A[I] = VAL
                            SET POS = I
                            PRINT POS
                            Go to Step 6
                    [END OF IF]
                     SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Binary Search

- Binary search is a searching algorithm that works efficiently with a sorted list. The mechanism of binary search can be better understood by an analogy of a telephone directory. When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again, we open some page in the middle and the whole process is repeated until we finally find the right name.

- Consider an array A[] that is declared and initialized as

  int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

- and the value to be searched is VAL = 9. The algorithm will proceed in the following manner.

  BEG = 0, END = 10, MID = (0 + 10)/2 = 5

- Now, VAL = 9 and A[MID] = A[5] = 5
- A[5] is less than VAL, therefore, we now search for the value in the second half of the array. So, we change the values of BEG and MID.

- Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8

  VAL = 9 and A[MID] = A[8] = 8
- A[8] is less than VAL, therefore, we now search for the value in the second half of the segment. So, again we change the values of BEG and MID.
- Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9
- Now, VAL = 9 and A[MID] = 9.

- However, if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

(a) If VAL < A[MID], then VAL will be present in the left segment of the array. So, the value of END will be changed as END = MID – 1.

(b) If VAL > A[MID], then VAL will be present in the right segment of the array. So, the value of BEG will be changed as BEG = MID + 1.

- Finally, if VAL is not present in the array, then eventually, END will be less than BEG. When this happens, the algorithm will terminate and the search will be unsuccessful.

Algorithm for binary search

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:          SET MID = (BEG + END)/2
Step 4:          IF A[MID] = VAL
                       SET POS = MID
                       PRINT POS
                       Go to Step 6
                 ELSE IF A[MID] > VAL
                       SET END = MID - 1
                 ELSE
                       SET BEG = MID + 1
                 [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

- The complexity of the binary search algorithm can be expressed as f(n), where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made. In the binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half. Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as

$$2f(n) > n \text{ or } f(n) = \log_2 n$$

Introduction to Sorting

- Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ..... < A[N]. For example, if we have an array that is declared and initialized as

  int A[] = {21, 34, 11, 9, 1, 0, 22};

- Then the sorted array (ascending order) can be given as:

  A[] = {0, 1, 9, 11, 21, 22, 34;

- A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order. Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly. There are two types of sorting:
- **Internal sorting** which deals with sorting the data stored in the computer's memory
- **External sorting** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory

Bubble Sort

- Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.
- This procedure of sorting is called bubble sorting because elements ‘bubble’ to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Working of bubble sort

a) In Pass 1, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–2] is compared with A[N–1]. Pass 1 involves n–1 comparisons and places the biggest element at the highest index of the array.

b) In Pass 2, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–3] is compared with A[N–2]. Pass 2 involves n–2 comparisons and places the second biggest element at the second highest index of the array.

c) In Pass 3, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–4] is compared with A[N–3]. Pass 3 involves n–3 comparisons and places the third biggest element at the third highest index of the array.

d) In Pass n–1, A[0] and A[1] are compared so that A[0] < A[1]. After this step, all the elements of the array are arranged in ascending order.

Example

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

- Pass 1:

a) Compare 30 and 52. Since 30 < 52, no swapping is done.

b) Compare 52 and 29. Since 52 > 29, swapping is done.

30, 29, 52, 87, 63, 27, 19, 54

a) Compare 52 and 87. Since 52 < 87, no swapping is done.

b) Compare 87 and 63. Since 87 > 63, swapping is done.

30, 29, 52, 63, 87, 27, 19, 54

a) Compare 87 and 27. Since 87 > 27, swapping is done.

30, 29, 52, 63, 27, 87, 19, 54

a) Compare 87 and 19. Since 87 > 19, swapping is done.

30, 29, 52, 63, 27, 19, 87, 54

a) Compare 87 and 54. Since 87 > 54, swapping is done.

30, 29, 52, 63, 27, 19, 54, 87

- Pass 2:

a) Compare 30 and 29. Since 30 > 29, swapping is done.
   29, 30, 52, 63, 27, 19, 54, 87

a) Compare 30 and 52. Since 30 < 52, no swapping is done.

b) Compare 52 and 63. Since 52 < 63, no swapping is done.

c) Compare 63 and 27. Since 63 > 27, swapping is done.
   29, 30, 52, 27, 63, 19, 54, 87

a) Compare 63 and 19. Since 63 > 19, swapping is done
   29, 30, 52, 27, 19, 63, 54, 87

a) Compare 63 and 54. Since 63 > 54, swapping is done.
   29, 30, 52, 27, 19, 54, 63, 87

- Pass 3:

a) Compare 29 and 30. Since 29 < 30, no swapping is done.

b) Compare 30 and 52. Since 30 < 52, no swapping is done.

c) Compare 52 and 27. Since 52 > 27, swapping is done.

29, 30, 27, 52, 19, 54, 63, 87

a) Compare 52 and 19. Since 52 > 19, swapping is done.

29, 30, 27, 19, 52, 54, 63, 87

a) Compare 52 and 54. Since 52 < 54, no swapping is done.

- Pass 4:

a) Compare 29 and 30. Since 29 < 30, no swapping is done.

b) Compare 30 and 27. Since 30 > 27, swapping is done.

29, 27, 30, 19, 52, 54, 63, 87

a) Compare 30 and 19. Since 30 > 19, swapping is done.

29, 27, 19, 30, 52, 54, 63, 87

a) Compare 30 and 52. Since 30 < 52, no swapping is done.

- Pass 5:

a) Compare 29 and 27. Since 29 > 27, swapping is done.

27, 29, 19, 30, 52, 54, 63, 87

a) Compare 29 and 19. Since 29 > 19, swapping is done.

27, 19, 29, 30, 52, 54, 63, 87

a) Compare 29 and 30. Since 29 < 30, no swapping is done.

- Pass 6:

a) Compare 27 and 19. Since 27 > 19, swapping is done.

19, 27, 29, 30, 52, 54, 63, 87

a) Compare 27 and 29. Since 27 < 29, no swapping is done.

- Pass 7:

a) Compare 19 and 27. Since 19 < 27, no swapping is done

19, 27, 29, 30, 52, 54, 63, 87

Algorithm for bubble sort

```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For 1 = 0 to N-1
Step 2:     Repeat For J = 0 to N - I
Step 3:                 IF A[J] > A[J + 1]
                        SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 4: EXIT
```

Insertion Sort

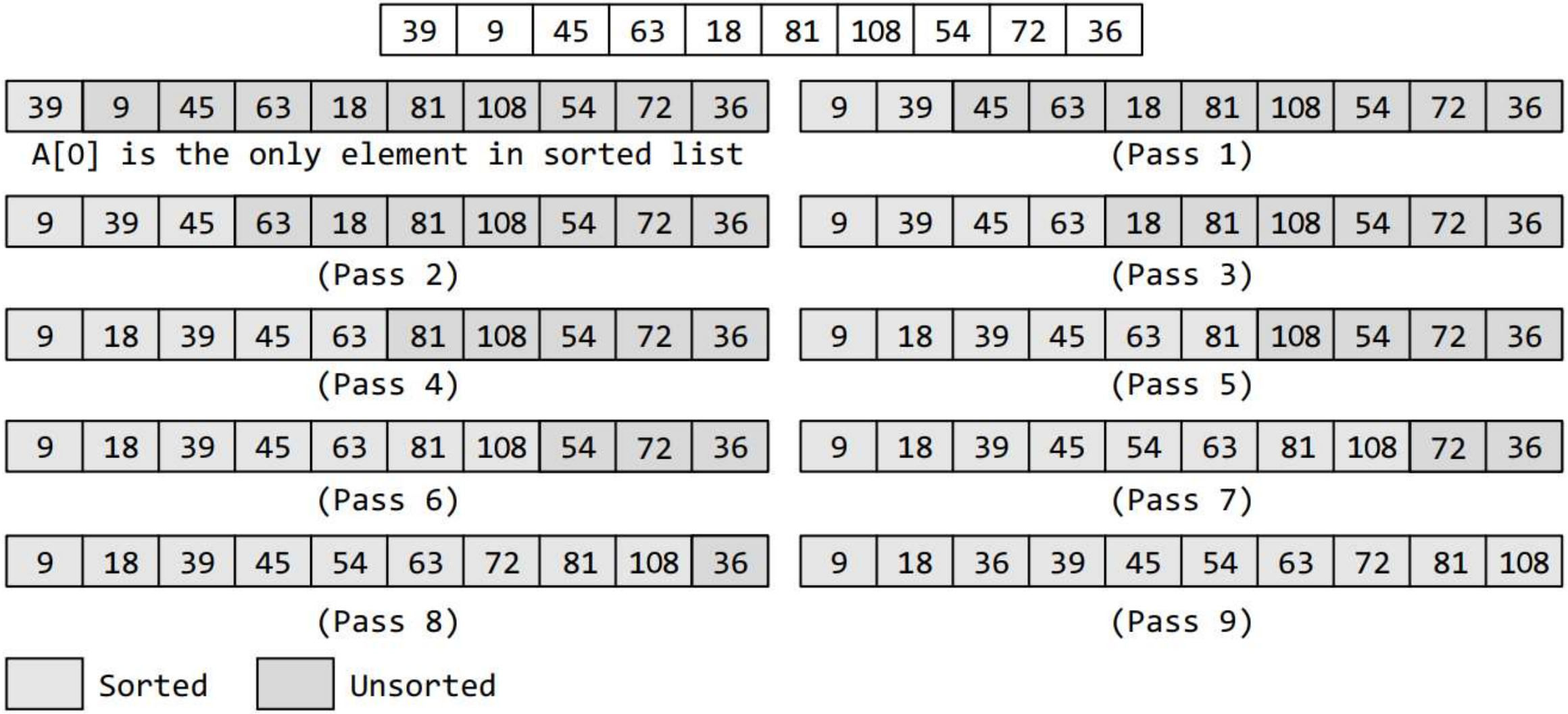
- Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.
- The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.
- Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort

Working of Insertion sort

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if LB = 0).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set

- Start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the first two elements and put at its correct position.
- Repeat until the entire array is sorted.

Example


39 9 45 63 18 81 108 54 72 36
39 9 45 63 18 81 108 54 72 36
A[0] is the only element in sorted list
9 39 45 63 18 81 108 54 72 36
(Pass 1)
9 39 45 63 18 81 108 54 72 36
(Pass 2)
9 39 45 63 18 81 108 54 72 36
(Pass 3)
9 18 39 45 63 81 108 54 72 36
(Pass 4)
9 18 39 45 63 81 108 54 72 36
(Pass 5)
9 18 39 45 63 81 108 54 72 36
(Pass 6)
9 18 39 45 54 63 81 108 72 36
(Pass 7)
9 18 39 45 54 63 72 81 108 36
(Pass 8)
9 18 36 39 45 54 63 72 81 108
(Pass 9)
Sorted
Unsorted

Algorithm for insertion sort

```
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 <= ARR[J]
                    SET ARR[J + 1] = ARR[J]
                    SET J = J - 1
             [END OF INNER LOOP]
Step 5:      SET ARR[J + 1] = TEMP
         [END OF LOOP]
Step 6: EXIT
```

Advantages of insertion sort

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- it requires less memory space (only O(1) of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements

Selection Sort

- Selection sort is a sorting algorithm that has a quadratic running time complexity of O(n2), thereby making it inefficient to be used on large lists. Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and small keys.

Working of selection sort

- First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,
- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
- In Pass 2, find the position POS of the smallest value in sub-array of N–1 elements. Swap ARR[POS] with ARR[1]. Now, ARR[0] and ARR[1] is sorted.
- In Pass N–1, find the position POS of the smaller of the elements ARR[N–2] and ARR[N–1]. Swap ARR[POS] and ARR[N–2] so that ARR[0], ARR[1], ..., ARR[N–1] is sorted

Example

39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

Algorithm for selection sort

```
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-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
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 A[K] with ARR[POS]
      [END OF LOOP]
Step 4: EXIT
```

Complexity of Selection Sort

- Selection sort is a sorting algorithm that is independent of the original order of elements in the array. In Pass 1, selecting the element with the smallest value calls for scanning all n elements; thus, n–1 comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position. In Pass 2, selecting the second smallest value requires scanning the remaining n – 1 elements and so on. Therefore,

  $(n – 1) + (n – 2) + ... + 2 + 1$

  $= n(n – 1) / 2 = O(n2)$ comparisons

Advantages of Selection Sort

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort

Radix Sort

- Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the radix is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.
- During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the nth pass, where n is the length of the name with maximum number of letters.

- After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on.
- When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the length of the number having maximum number of digts.

Algorithm for radix sort

```
Algorithm for 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 <= NOP-1
Step 5:            SET I = 0 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:                  INCEREMENT bucket count for bucket numbered DIGIT
                   [END OF LOOP]
Step 10:           Collect the numbers in the bucket
        [END OF LOOP]
Step 11: END
```

Example

Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

Pass-1: The numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

Pass-2: The numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

Pass-3: The numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

- The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as

  123, 345, 472, 555, 567, 654, 808, 911, 924

- To calculate the complexity of radix sort algorithm, assume that there are n numbers that have to be sorted and k is the number of digits in the largest number. In this case, the radix sort algorithm is called a total of k times. The inner loop is executed n times. Hence, the entire radix sort algorithm takes O(kn) time to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in O(n) asymptotic time

Pros and Cons of Radix Sort

- Radix sort is a very simple algorithm. When programmed properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters.
- But there are certain trade-offs for radix sort that can make it less preferable as compared to other sorting algorithms. Radix sort takes more space than other sorting algorithms. Besides the array of numbers, we need 10 buckets to sort numbers, 26 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters

- Another drawback of radix sort is that the algorithm is dependent on digits or letters. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten. Even if the sorting order changes, the algorithm has to be rewritten. Thus, radix sort takes more time to write and writing a general purpose radix sort algorithm that can handle all kinds of data is not a trivial task.
- Radix sort is a good choice for many programs that need a fast sort, but there are faster sorting algorithms available. This is the main reason why radix sort is not as widely used as other sorting algorithms.

Time Complexities of all Sorting Algorithms

Algorithm	Time Complexity		
	Best case	**Average case**	**Worst case**
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Radix Sort	O(nk)	O(nk)	O(nk)

DISCUSSION…

SVKM'S NMIMS
Deemed to be UNIVERSITY

THANK YOU