Marwadi University
Marwadi Chandarana Group

Sorting and Searching Techniques

Contents

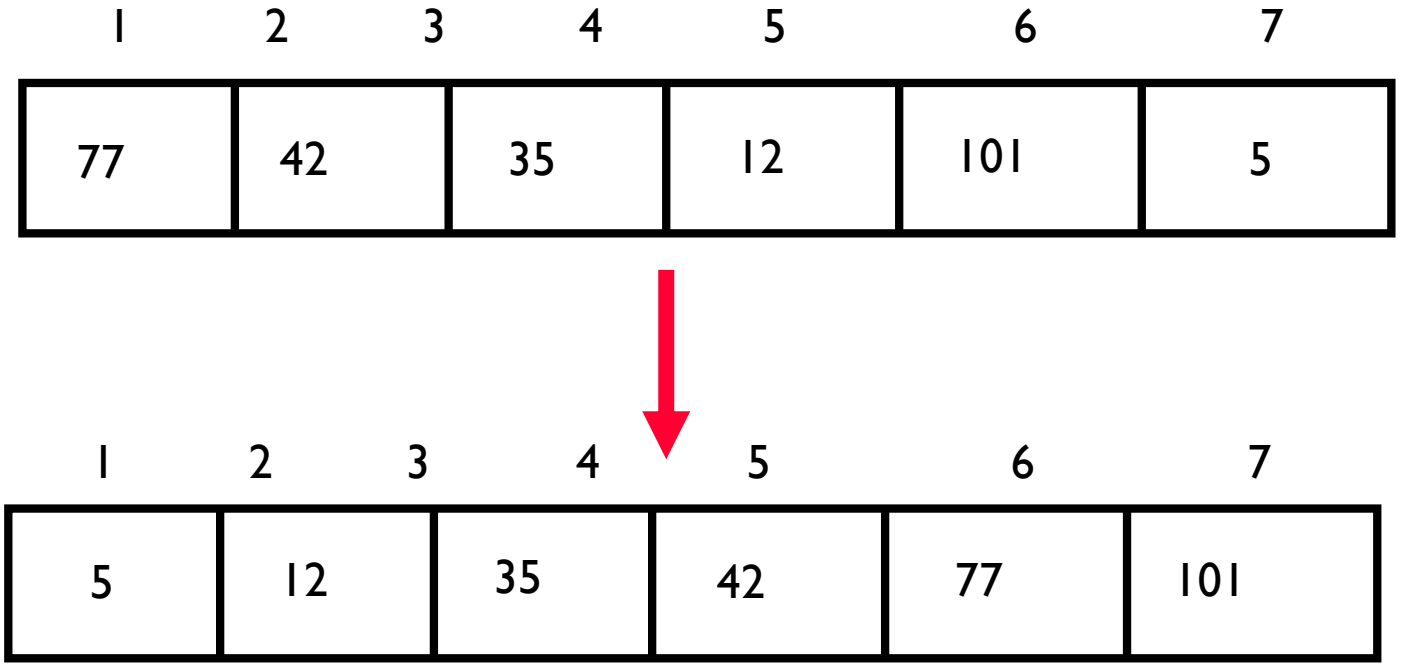
- **Sorting Concepts and methods:**
  - Bubble sort,
  - Selection sort,
  - Insertion sort,
  - Quick sort,
  - Merge sort,
- **Searching Concepts and Methods:**
  - Sequential search,
  - Binary search

Sorting

- Sorting refers to arranging data in a particular format.
- Sorting algorithm specifies the way to arrange data in a particular order.
- The importance of sorting:
  - data searching can be optimized to a very high level, if data is stored in a sorted manner.
  - used to represent data in more readable formats.
- Examples of sorting in real-life scenarios:
  - **Telephone Directory**
  - **Dictionary**

Marwadi University
Marwadi Chandarana Group

Sorting takes an unordered collection and makes it an ordered one.


1
2
3
4
5
6
7
77
42
35
12
101
5
1
2
3
4
5
6
7
5
12
35
42
77
101

SORTING TECHNIQUES

- There are various methods for sorting:
  - Bubble sort, Insertion sort, Selection sort, Quick sort, Merge sort….
  - They having different average and worst case behaviors:

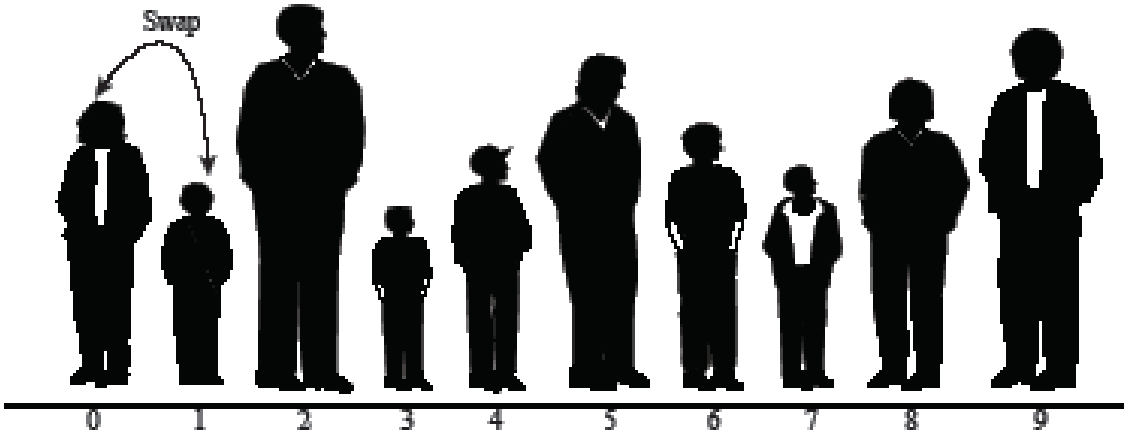
1. *BUBBLE SORT*

Marwadi University
Marwadi Chandarana Group

Introduction:

- *Bubble sorting* is a simple sorting technique in which we arrange the elements of the list by forming pairs of adjacent elements.
- That means we form the pair of the ith and (i+1)th element.
- If the order is ascending, we interchange the elements of the pair if the first element of the pair is greater than the second element.

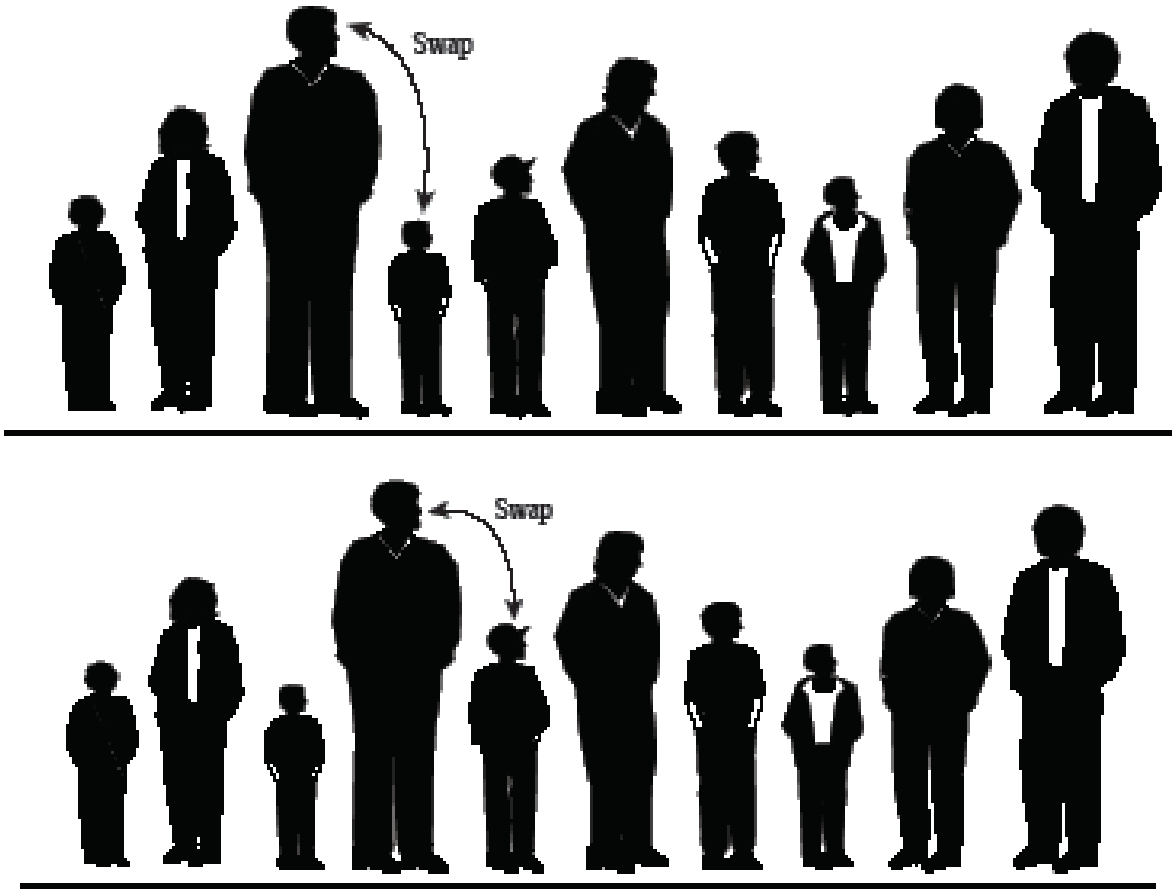
1. *BUBBLE SORT*



Bubble sort: beginning of first pass

Marwadi University
Marwadi Chandarana Group

1. *BUBBLE SORT*



1. *BUBBLE SORT*

Bubble sort: end of First pass

Pseudo code for Bubble Sort

Algorithm bubble (a, n)

Pre: Unsorted array a of length n.

Post: Sorted array in ascending order of length n

```
for i = 1 to (n – 1) do                    // n-1 passes
        for j = 1 to (n- i ) do
            if ( a[j] > a[j+1] )
                1. temp=a[j]               //swapping of numbers
                2. a[j]=a[j+1]
                3. a[j+1]=temp
```

Sort by comparing each adjacent pair of items in a list in turn, swapping the items if not in order, and repeating the pass through the list until no swaps are done.

Exercise

- Sort the following numbers using bubble sort.

25 14 62 35 69 12

Pass 1

25	14	62	35	69	12
14	25	62	35	69	12
14	25	62	35	69	12
14	25	35	62	69	12
14	25	35	62	69	12
14	25	35	62	12	69

Number of comparisons = 5

Pass 2

14	25	35	62	12	69
14	25	35	62	12	69
14	25	35	62	12	69
14	25	35	62	12	69
14	25	35	12	62	69

Number of comparisons = 4

Pass 3

14	25	35	12	62	69
14	25	35	12	62	69
14	25	35	12	62	69
14	25	12	35	62	69

Number of comparisons = 3

Pass 4

14	25	12	35	62	69
14	25	12	35	62	69
14	12	25	35	62	69

Number of comparisons = 2

Pass 5

14	12	25	35	62	69
12	14	25	35	62	69

Number of comparisons = 1

Number of elements = 6

Number of pass = 5

Number of comparison in any pass

= n – pass number

Bubble Sort

- Number of elements = n
- Number of pass = n – 1
- If k is pass number then
  - Number of comparisons in k^{th} pass = n – k

Complexity of algorithm

- No. of comparisons in 1st pass = n – 1
- No. of comparisons in 2nd pass = n – 2
- No. of comparisons in 3rd pass = n – 3

*

*

- No. of comparisons in (n – 1) pass = 1

$f(n) = (n – 1) + (n – 2) + \ldots + 1$

$= n(n-1)/2 = O(n^2)$

Therefore Worst case complexity = $O(n^2)$

If list is already sorted then no. of pass = 1 and no. of comparisons = n-1

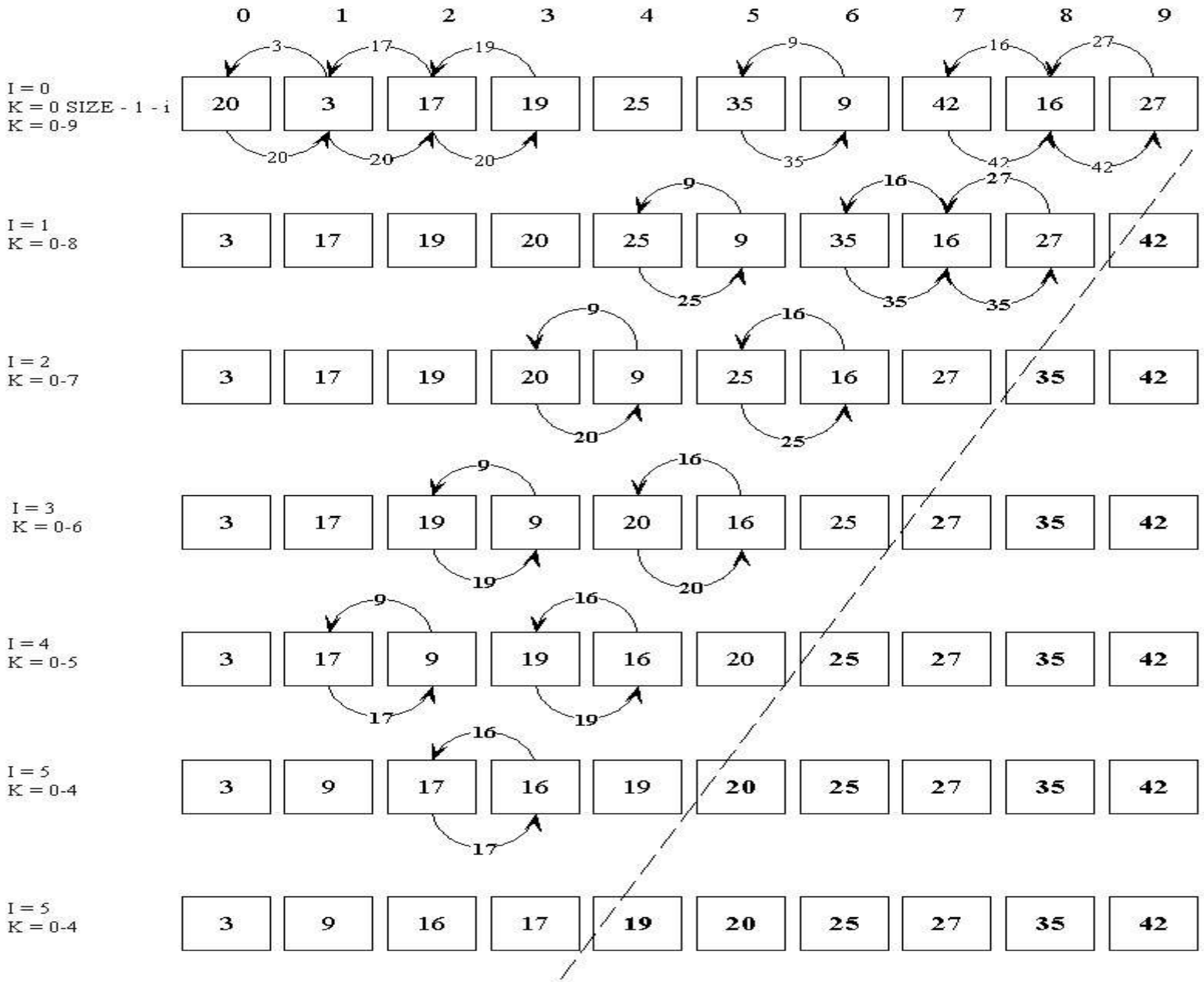
Best case complexity = O(n)

Exercise

Sort following elements using bubble sort method.

20 3 17 19 25 35 9 42 16 27

Bubble Sort or Sinking Sort

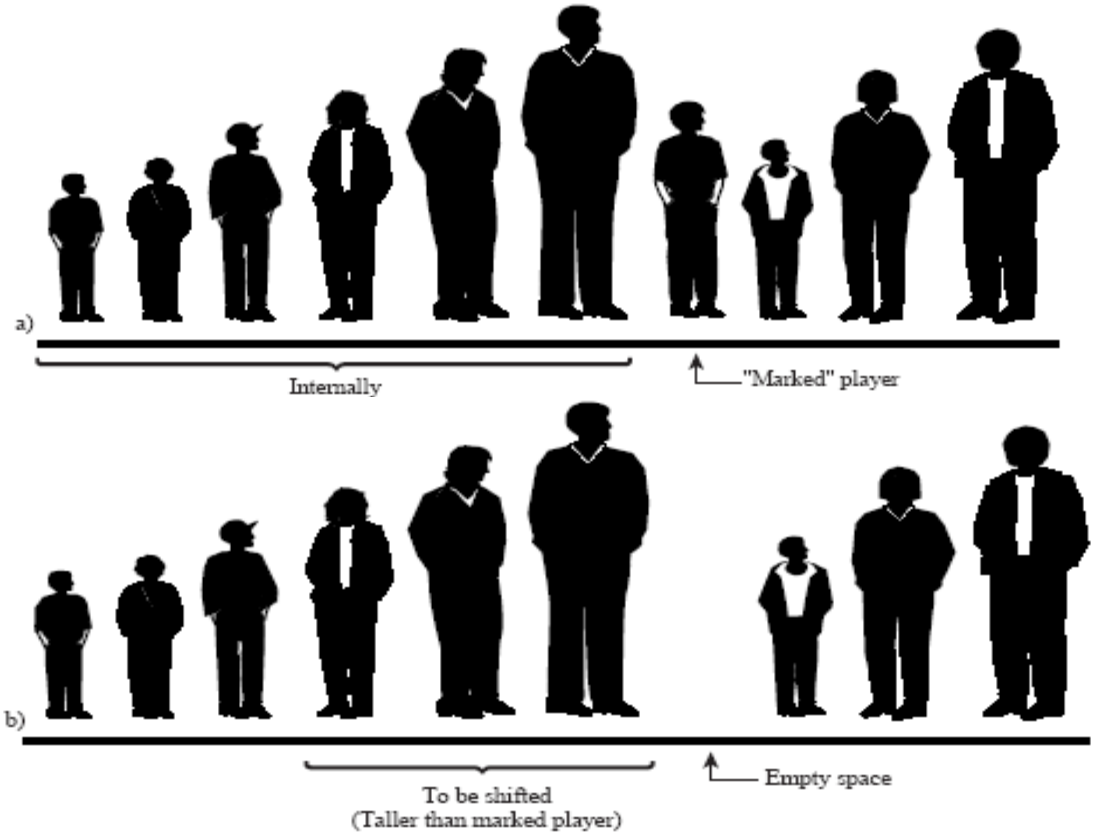


Everything below the dotted line is guaranteed to be sorted.

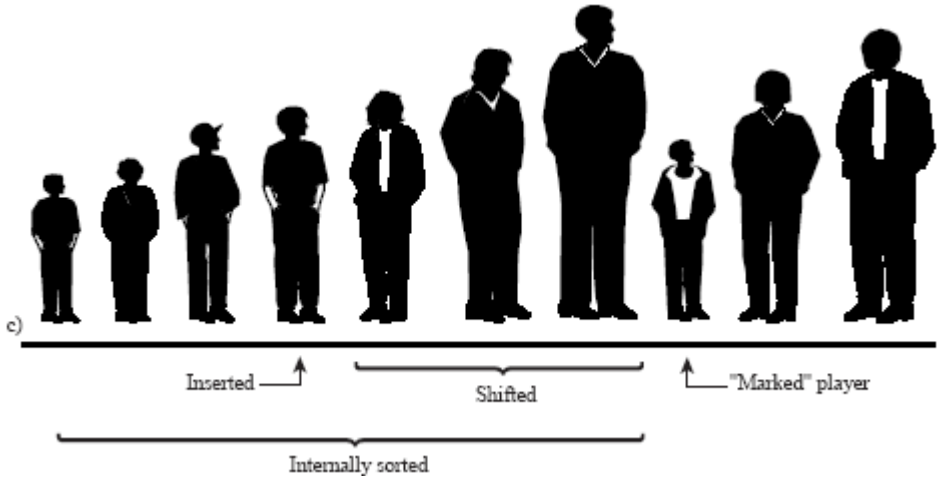
Because nothing was found out of order on the last pass we quit.

2. INSERTION SORT

Sort by repeatedly taking the next item and inserting it into the final array in its proper order with respect to items already inserted.



2. INSERTION SORT



Insertion sort

- Suppose array A with n elements A[1], A[2], A[3],….,A[n]

Pass 1: A[1] is already sorted.

Pass 2: A[2] is inserted before or after A[1] such that A[1],A[2] is sorted array.

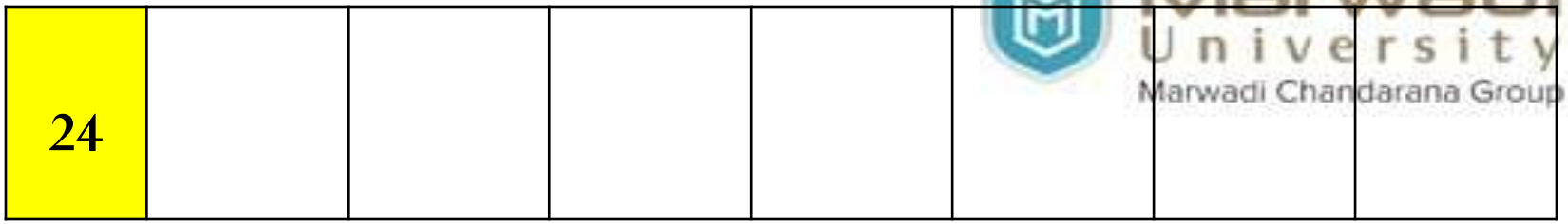
Pass 3: A[3] is inserted in A[1], A[2] in such a way that A[1],A[2],A[3] is sorted array.

Pass N: A[N] is inserted in A[1],A[2],A[3]…A[n-1] in such a way that A[1],A[2],A[3]…A[n-1], A[n] is sorted array.

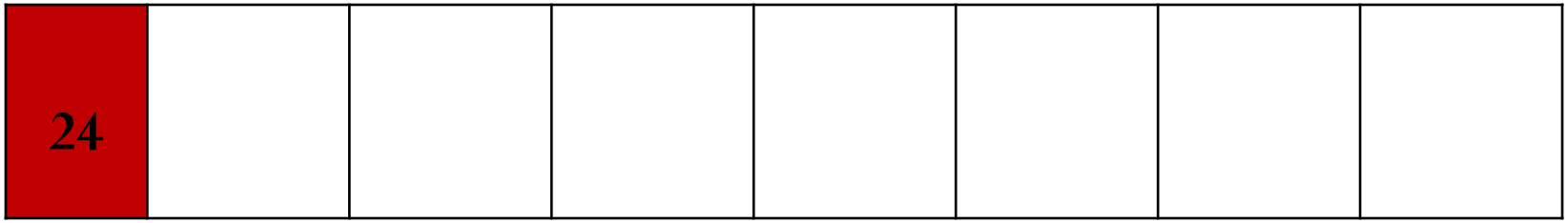
- This algorithm inserts A[k] into its proper position in the previously sorted sub array A[1], A[2],…,A[k-1]

Example:

24	**13**	**9**	**64**	**7**	**23**	**34**	**47**

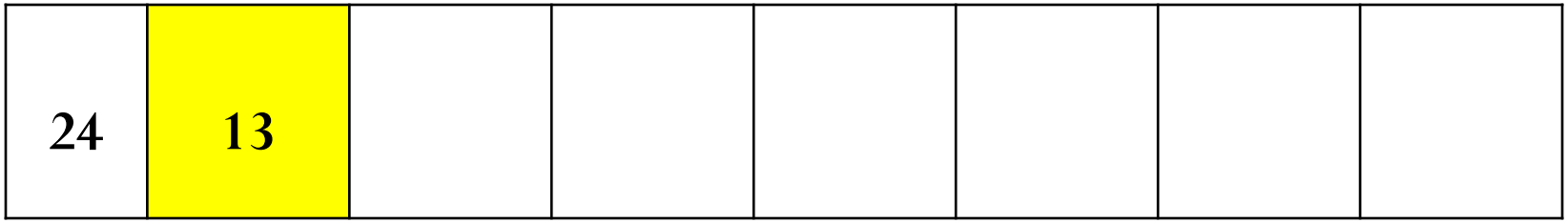


First value is considered as sorted.



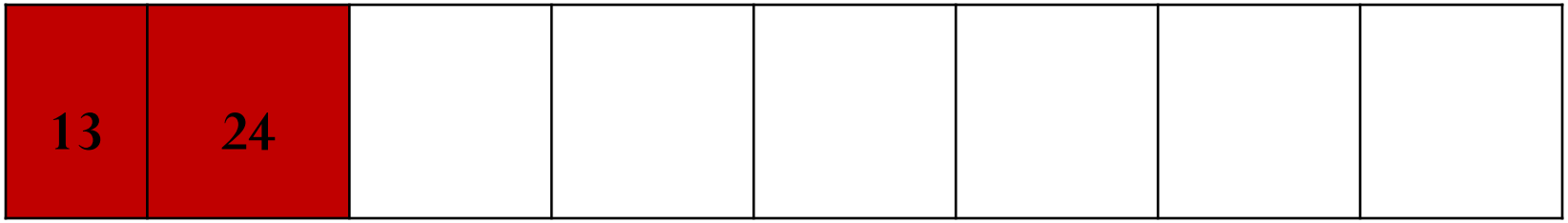
Pass 1:

Insert next value 13



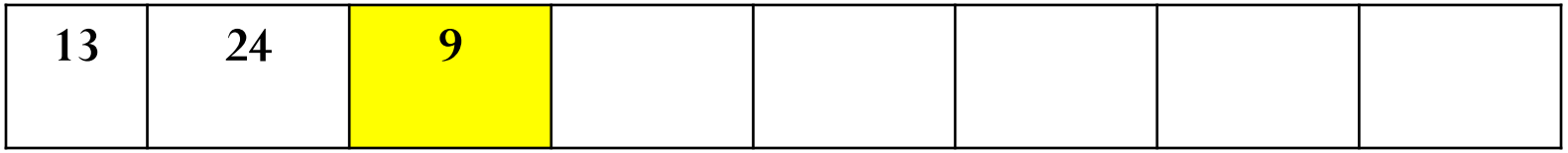
13 is less than 24

Swap 24 and 13



Pass 2:

Insert next value 9



9 is less than 24

Swap 24 and 9

9 is less than 13

Swap 13 and 9

Pass 3:

Insert next value 64

9	13	24	64				

Pass 4 :

Insert next value 7

9	13	24	64	7			

7 is less than 64

Swap 7 with 64 .

9	13	24	7	64			

7 is less than 24

Swap 7 with 24 .

9	13	24	7	64			

9	13	7	24	64			

7 is less than 13

Swap 7 with 13

9	7	13	24	64			

7 is less than 9

Swap 7 with 9

9	7	13	24	64			

7	9	13	24	64			

Pass 5 :

Insert next value 23

7	9	13	24	64	23		

23 is less than 64

Swap 23 with 64

7	9	13	24	64	23		

7	9	13	24	23	64		

23 is less than 24

Swap 23 with 24

7	9	13	23	24	64		

- **Pass 6:**
- Insert next value 34

7	9	13	23	24	64	34	

34 is less than 64

Swap 34 with 64

7	9	13	23	24	34	64	

- **Pass 7:**
- Insert next value 47

7	9	13	23	24	34	64	47

47 is less than 64

Swap 47 with 64

7	9	13	23	24	34	47	64

Insertion sort

- Number of pass n-1
- In each pass proper position is created for element to be inserted by sliding elements which are greater than that element.

Insertion sort - Algorithm

Algorithm insertion (a, n)

Pre: Unsorted list a of length n.

Post: Sorted list a in ascending order of length n

1. for i = 1 to (n -1) do // n-1 passes
    1. temp = a[i] //value to be inserted
    2. ptr = i – 1 //pointer to move downward
    3. while (temp < a[ptr] and ptr >= 0)
        1. a[ptr + 1] = a[ptr]
        2. ptr = ptr - 1
    4. a[ptr +1] = temp

Insertion sort - Complexity

- Best Case: - O (n)

  List is already sorted. In each iteration, first element of unsorted list compared with last element of sorted list, thus (n-1) comparisons.

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

  List sorted in reverse order. First element of unsorted list compared with one element of sorted list, second compared with 2 elements. Last element to be inserted compared with all the n-1 elements.

  $1 + 2 + 3 + \ldots\ldots\ldots\ldots\ldots\ldots (n-2) + (n-1)$

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

  $= O (n^2)$

- Average Case: - $O(n^2)$

Marwadi University
Marwadi Chandarana Group

Exercise

- Sort the following array using insertion sort.

25	2	10	5	8	7

Insertion Sort Example

Marwadi University
Marwadi Chandarana Group

25 2 10 5 8 7

25 First value is considered as sorted.

Pass 1:

2 25

Insert next value 2

2 is less than 25

25 slides over

Insertion Sort Example

Pass 2:

2 10 25

Insert next value 10

10 is less than 25

25 slides over

Pass 3:

2 5 10 25

Insert next value 5

5 is less than 10, 25

10, 25 slide over

Insertion Sort Example

Pass 4:

2 5 8 10 25

Insert next value 8

8 is less than 10, 25

10, 25 slide over

Pass 5:

2 5 7 8 10 25

Insert next value 7

7 is less than 8,10, 25

8,10, 25 slide over

Exercise

- Sort following elements using insertion sort algorithm.

1. 78 12 34 98 22 65 11
2. 43 12 56 69 21 105 63 72 36 23
3. 58 63 78 10 19 81 51 25 37 49
4. 38 10 47 65 19 210 50 70 36

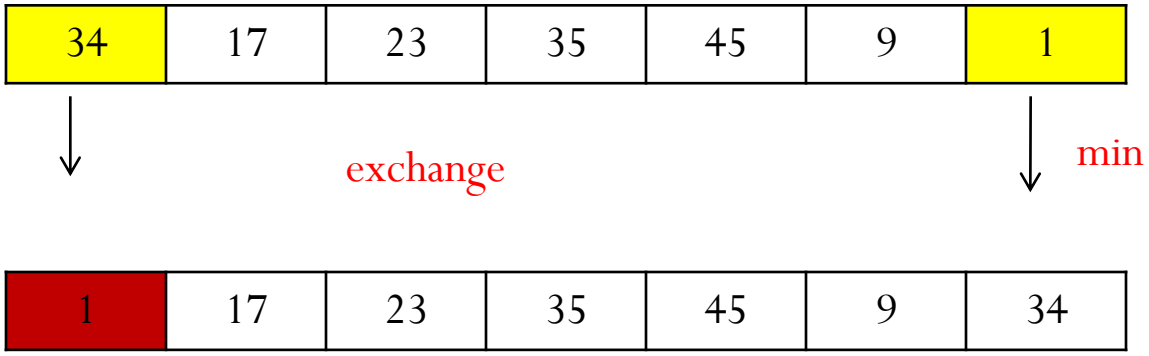
Marwadi University
Marwadi Chandarana Group

Selection Sort

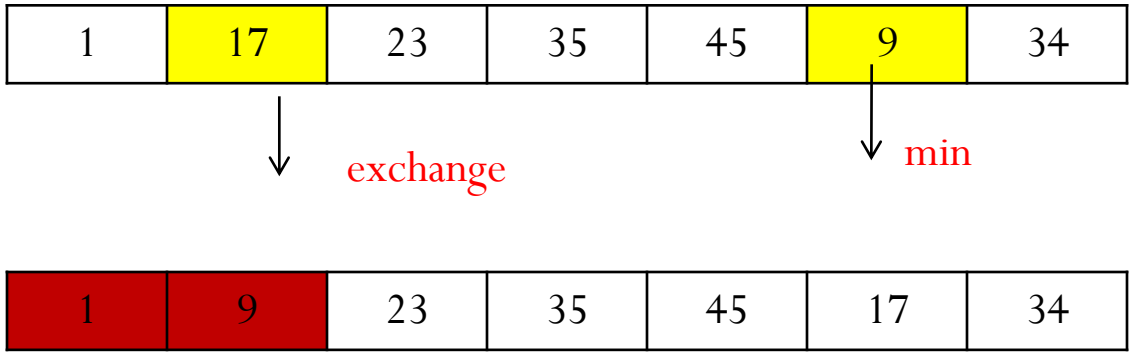
Selection sort

- Find the first smallest element in the list and place it at the first position.
- Find next smallest number and place it at the second position.
- And so on…

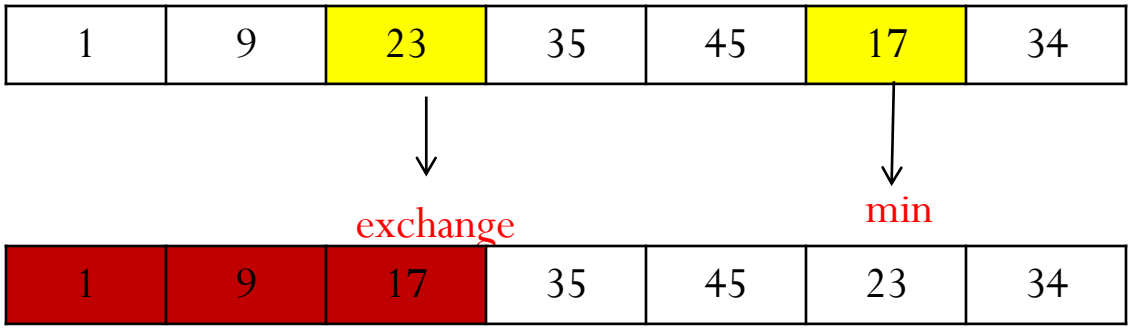
Pass 1:


34
17
23
35
45
9
1
exchange
min
1
17
23
35
45
9
34

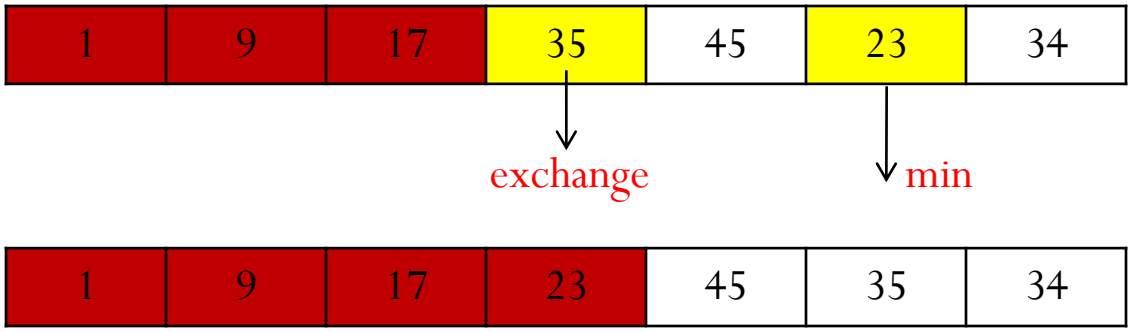
Pass 2:


1
17
23
35
45
9
34
exchange
min
1
9
23
35
45
17
34

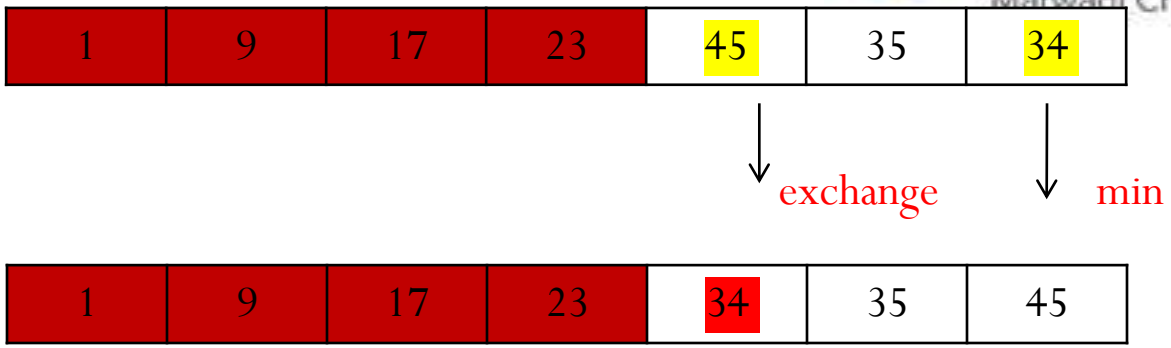
Pass 3:


1
9
23
35
45
17
34
exchange
min
1
9
17
35
45
23
34

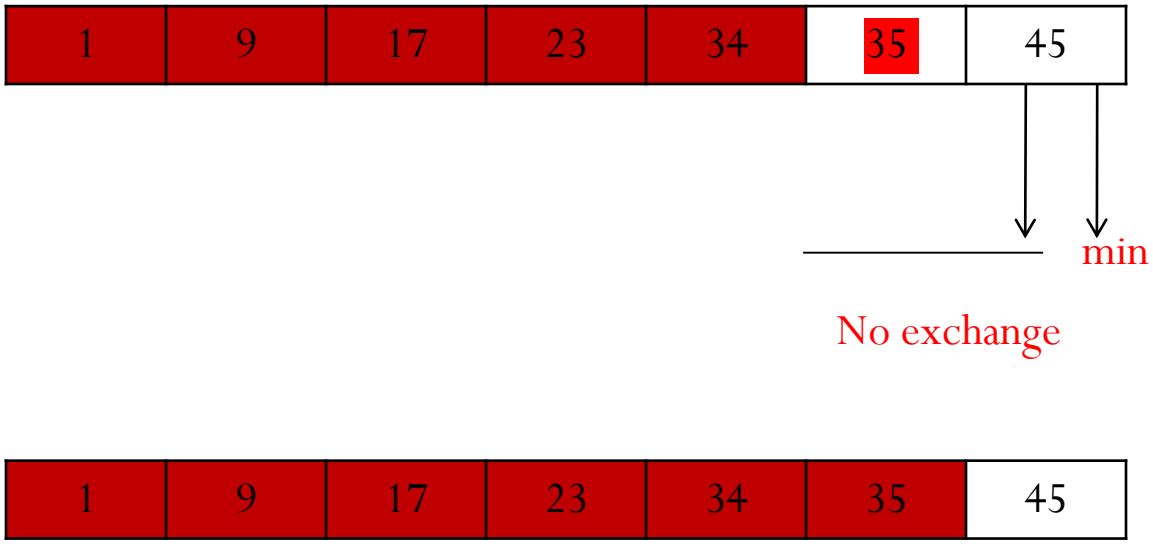
Pass 4:


1
9
17
35
45
23
34
exchange
min
1
9
17
23
45
35
34

Pass 5:



Pass 6:



Number of elements = n

Number of pass = n - 1

Selection Sort

Algorithm selection (a, n)

Pre: Unsorted array a of length n.

Post: Sorted list in ascending order of length n

1. for i = 0 to (n -2) do // n-1 passes
    1. min_index=i
    2. for j = (i+1) to (n -1) do
        1. if (a[min_index] > a[j])
            1. min_index = j
    3. if (min_index != i) //place smallest element at i^{th} place
        1. temp= a[i]
        2. a[i]=a[min_index]
        3. a[min_index]=temp

Complexity of algorithm

Worst case and best case complexity:

- No. of comparisons in 1st pass = N – 1
- No. of comparisons in 2nd pass = N – 2
- No. of comparisons in 3rd pass = N – 3

*

- No. of comparisons in (N – 1) pass = 1

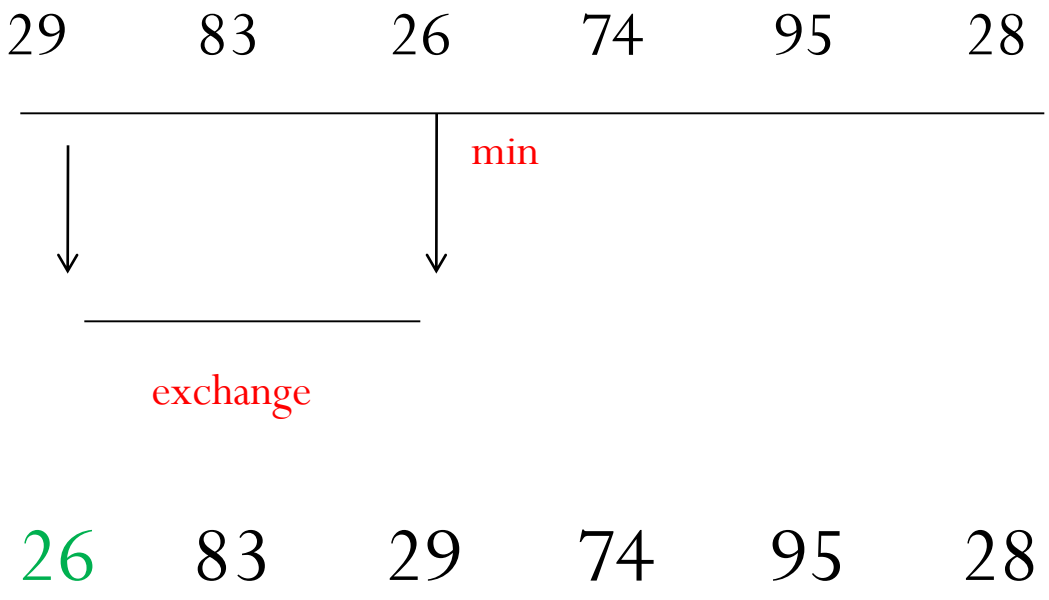
$f(n) = (n – 1) + (n – 2) + \ldots + 1$

$= n(n-1)/2 = O(n^2)$

- Sort following elements using selection sort method of sorting.

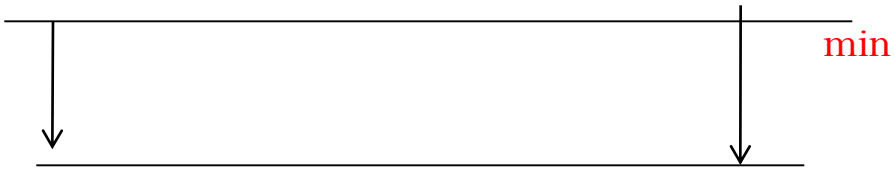
29 83 26 74 95 28

Pass 1



Pass 2

26 83 29 74 95 28

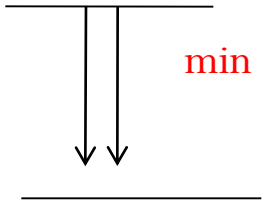


exchange

26 28 29 74 95 83

Pass 3

26 28 29 74 95 83

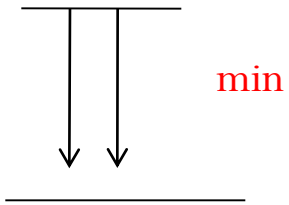


No exchange

26 28 29 74 95 83

Pass 4

26 28 29 74 95 83

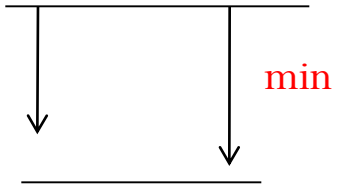


No exchange

26 28 29 74 95 83

Pass 5

26 28 29 74 95 83



Exchange

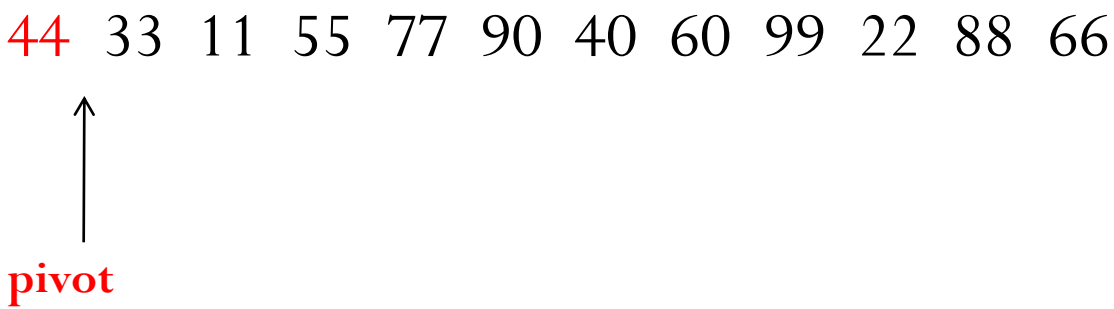
26 28 29 74 83 95

- Sort the following numbers using Selection Sort.

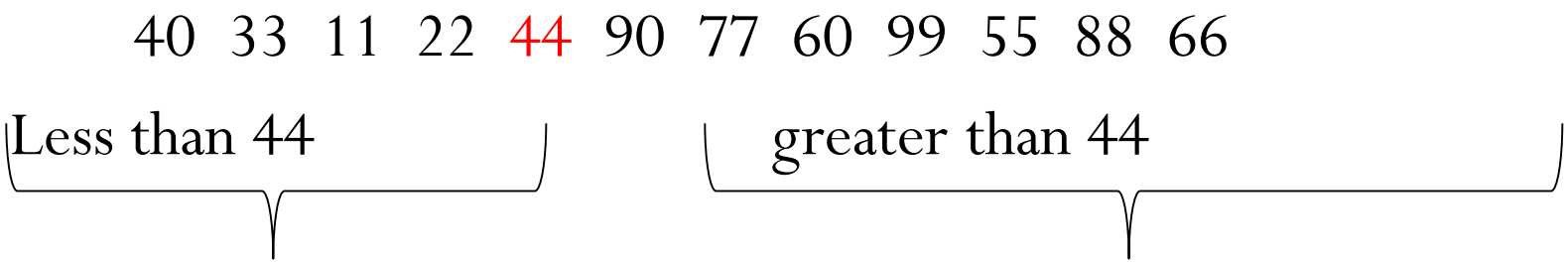
1. 59 31 40 90 76 100 21 5 85 14
2. 14 6 4 8 11 12 10 13
3. 25 57 48 37 12 92 86 33

Quick sort

Quick sort



After partitioning



Quick sort

- Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.
- The steps are:
  - Pick an element, called a pivot, from the list.
  - Reorder the list so that all elements which are less than the pivot come before the pivot and all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
  - Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Quick sort - partitioning

- Sort following elements using quick sort.

44 33 11 55 77 90 40 60 99 22 88 66

Quick sort – partition algorithm

Step 1: Repeatedly increase the pointer down by one position until a[down] > pivot

Step 2: Repeatedly decrease the pointer up by one position until a[up] <= pivot.

Step 3: if down < up, interchange a[down] and a[up]

Steps 1,2 ,3 are repeated until step 3 fails.

i.e. if up <= down, interchange pivot and a[up]

```
int partition (a, beg, end)
// Places pivot element piv at its proper position; elements
   before it are less than it & after it are greater than it
```

1. piv = a[beg]
2. up = end
3. down = beg
4. while (down < up)
    1. while((a[down] <= piv) && (down<end))
        1. down=down + 1
    2. while(a[up]>piv)
        1. up=up-1
    3. if (down<up)
        1. swap (a[down], a[up])
5. swap(a[beg], a[up])
6. return up

Algorithm sort (a, beg, end)

// a - array to be sorted, beg - starting index of array to be sorted, end - ending index of array to be sorted

Pre: Unsorted list a of length n.

Post: Sorted list in ascending order of length n

1. if (beg < end)
    1. j = partition(a, beg, end)
    2. sort(a, beg, j-1)
    3. sort (a, j+1, end)
2. else
    1. return

Complexity of quick sort algorithm

- Assume that array size n is power of 2
- Let $n = 2^m$, so that $m = \log_2 n$
- Assume that proper position for the pivot always turn out to be middle of array.
- During first pass there will be n comparisons.
- Array is split into two subarrays.
- For each of the subarrays n/2 comparisons are required.
- In next pass total 4 files are created each of size n/4

Complexity of quick sort algorithm

- Each file require n/4 comparisons yielding n/8 files.
- After m pass, there will be n files each of size 1.
- Total number of comparisons =
- = n + 2(n/2) + 4 (n/4) + 8 (n/8)+……
- = n + n + n + n+….+n (m times)
- = n.m
- = $n \log_2 n$
- Complexity of quicksort algorithm = $O(n \log_2 n)$

- If array is already sorted
- X[lb] is in its correct position.
- The original file is split into subfiles of sizes 0 and n-1.
- If this process continues ,total n-1 subfiles are sorted,the first of size n,the second of size n-1,the third of size n-3 and so on.
- So,
- Total no. of comaparisions are:

$f(n) = n+(n-1)+(n-3)+\ldots\ldots+2+1$

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

- So, The time complexity is $O(n^2)$.

- Sort array using quick sort.

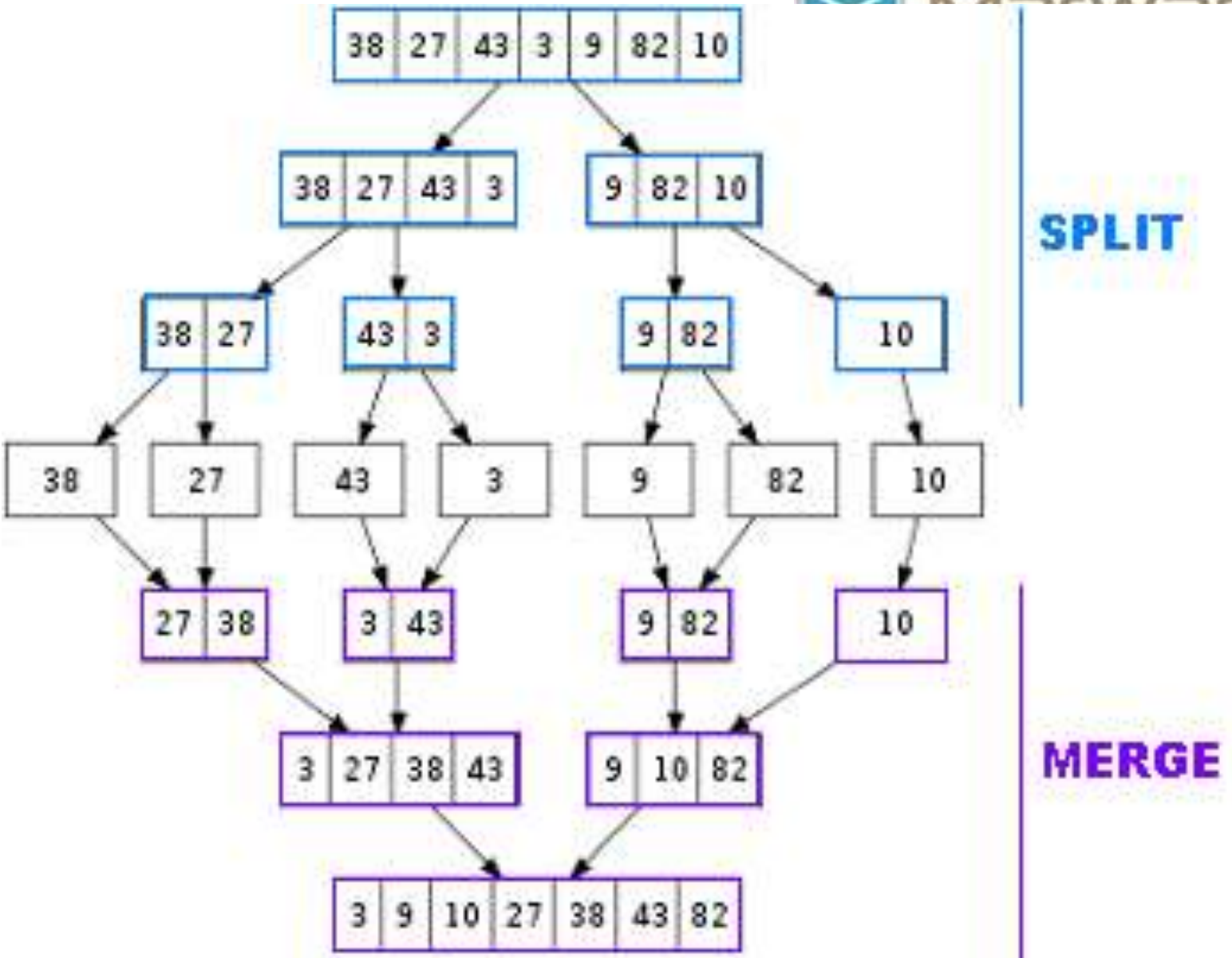
65, 21, 14, 97, 87, 78, 74, 76, 45, 84, 22

Merge Sort

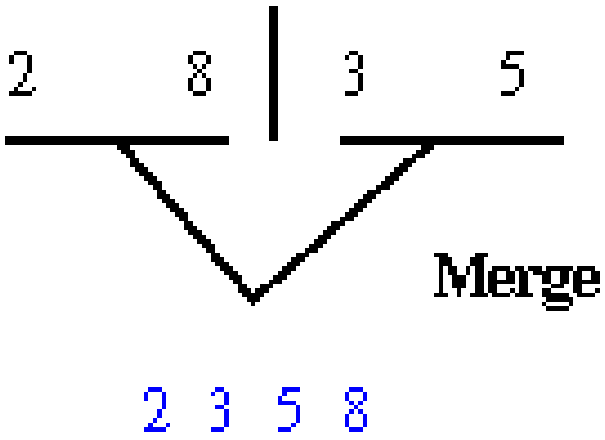
Merge Sort

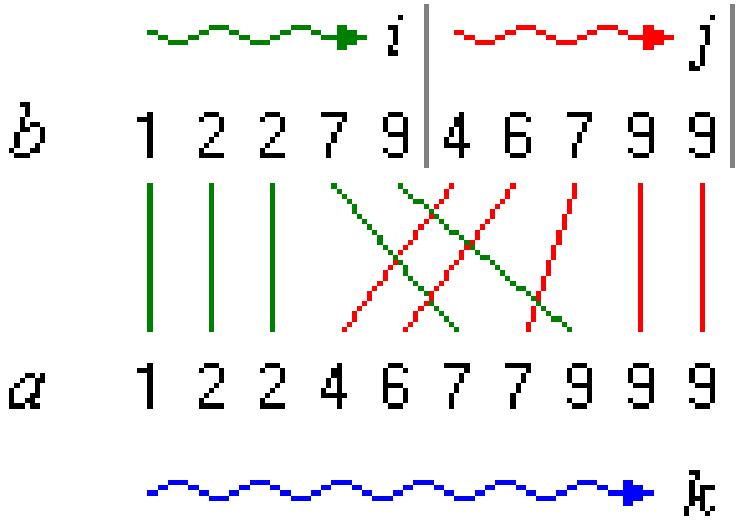
- Divide and conquer strategy

- The steps are –
  - Divide the unsorted list into two sub lists of about half the size.
  - Sort each sub list recursively by re-applying merge sort, till you reach a single element array
  - Merge the sub lists back into one sorted list.


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82
SPLIT
MERGE

Merging 2 sorted arrays





Merge Sort

Algorithm mergesort (a, low, high)

// a is array to be sorted, low is starting index of array to be sorted, high is ending index of array to be sorted

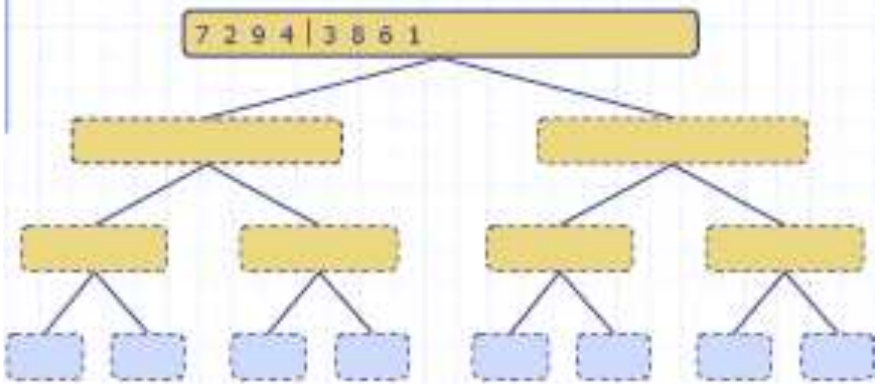
Pre: Unsorted list of length n.

Post: Sorted list in ascending order of length n

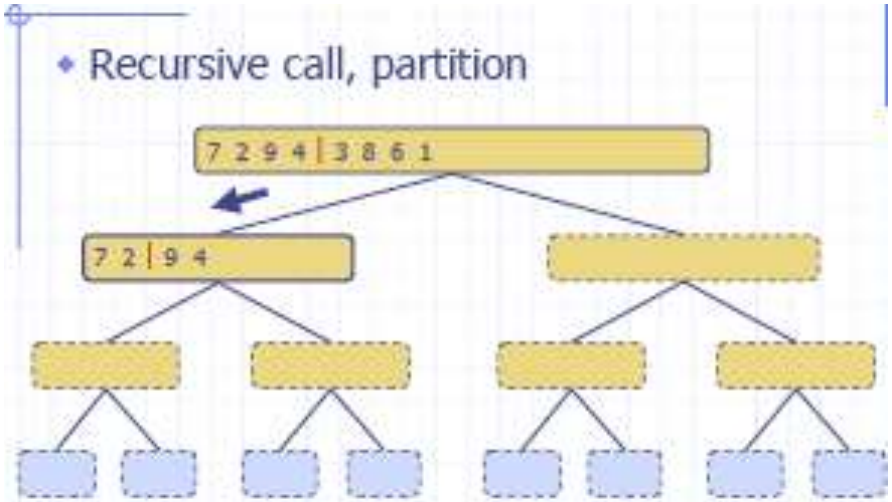
1. if (low < high)
    1. mid = (low + high)/2
    2. mergesort(x, low, mid)
    3. mergesort(x, (mid+1), high)
    4. merge(x, low, mid, high)


```
merge (a, low1, high1, high2)
1. i = low1;  j = high1 + 1;  k = 0
2. while (i<= high1) and (j<=high2)                    //Merge arrays
      1. if (x[i] <=x[j])
            1. aux[k] = x[i]
            2. k=k+1; i=i+1
      2. else
            1. aux[k] = x[j]
            2. k=k+1;  j=j+1
3. while (i<= high1)                                   // If jth list over, copy  ith as it is
            1. aux[k] = x[i]
            2. k=k+1; i=i+1
4. while (j<= high2)                                   // If ith list over, copy  jth as it is
            1. aux[k] = x[j]
            2. k=k+1;  j=j+1
5. k=0
6. for j = low1 to high2
            1. a[j] = aux[k]
            2. k = k+1
```

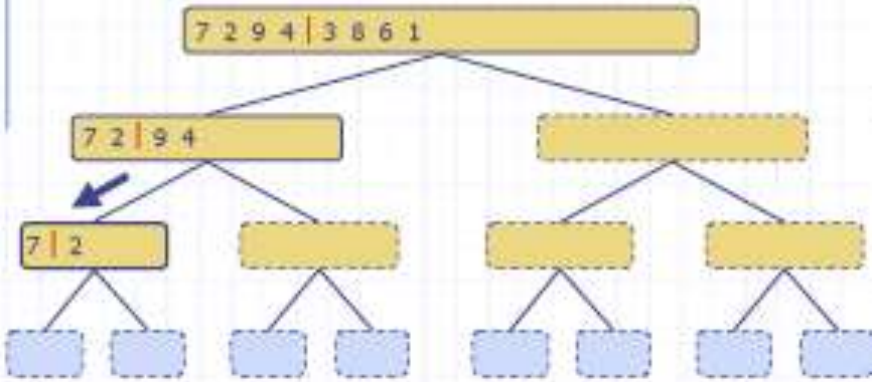
- Partition



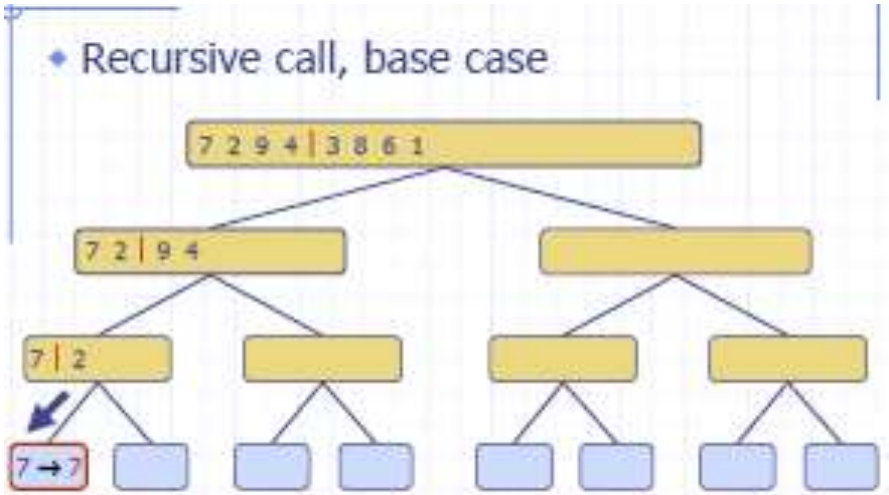
- Recursive call, partition



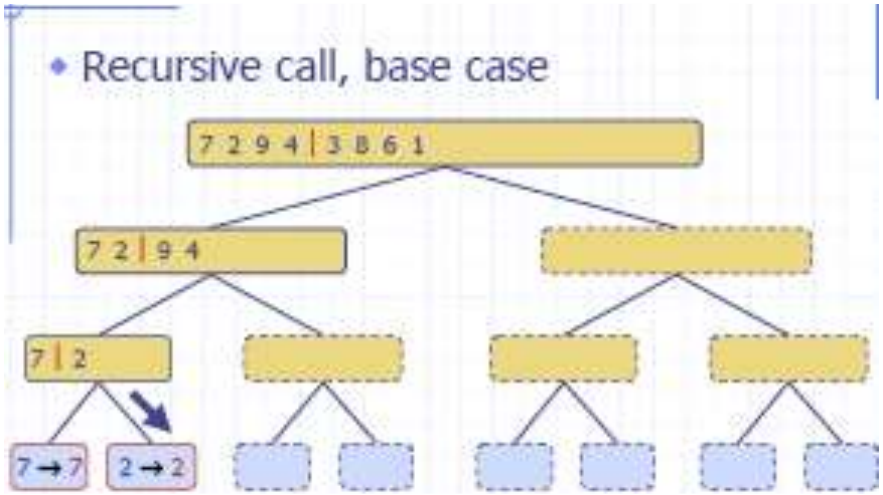
- Recursive call, partition



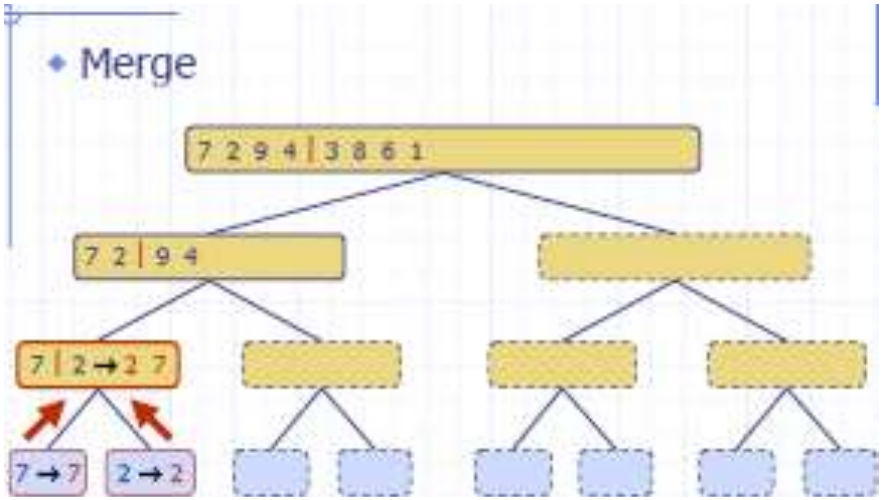
- Recursive call, base case



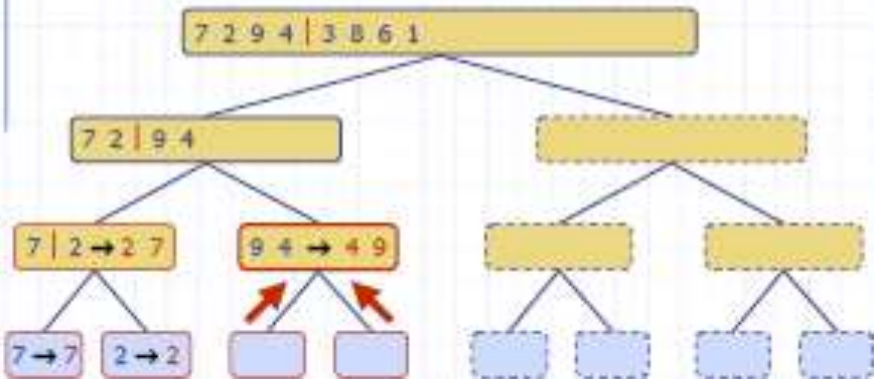
- Recursive call, base case



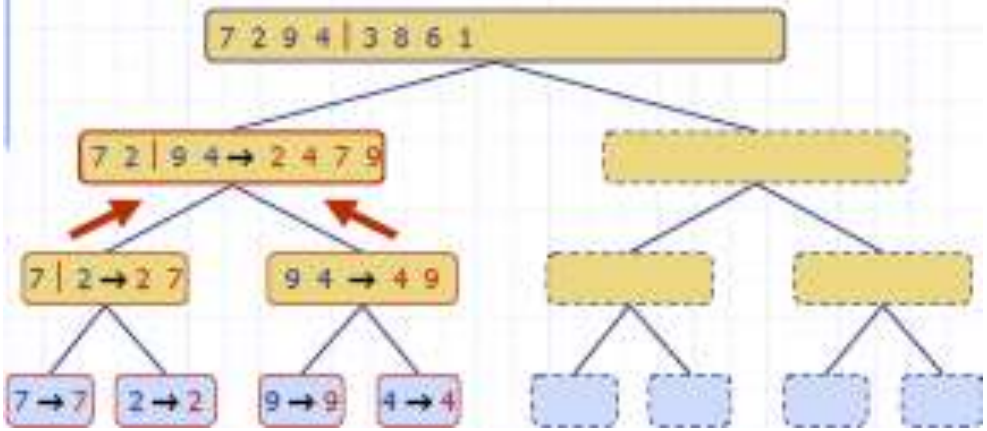
- Merge



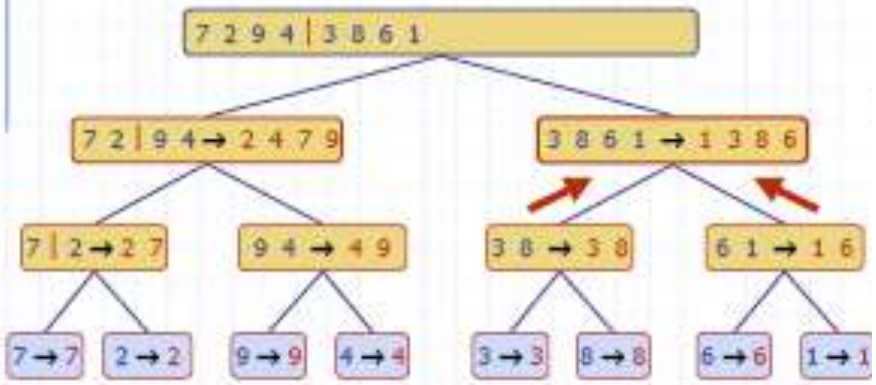
- Recursive call, ..., base case, merge



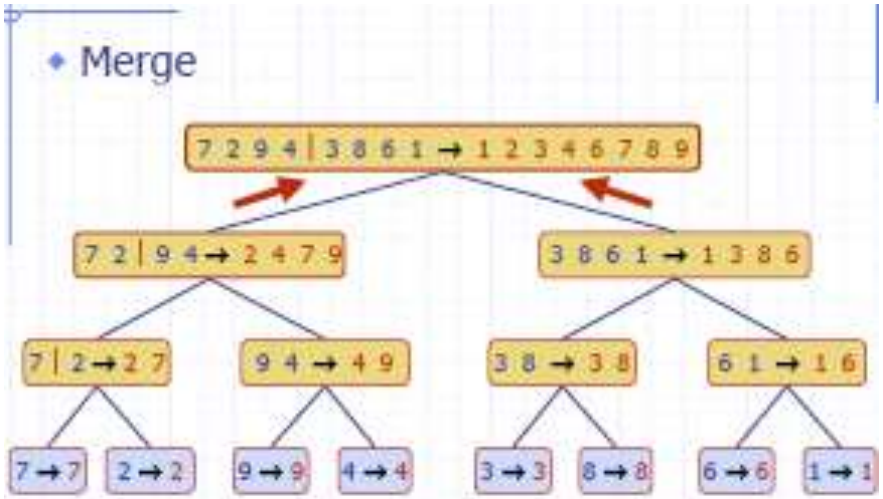
- Merge



- Recursive call, ..., merge, merge



- Merge



Complexity

- Best Case, Average Case, Worst case: O (n log n)

  Assume $n = 2^m$, $m = \log_2 n$.

  In first pass array split in two parts each of size n/2, 2nd pass

  4 parts of size n/4, then 8 parts of size n/8.....thus, file split m

  (log n) times before getting single element array. So there are

  (log n) passes.

  At every level there are n elements that have to be merged so

  there can not be more than n comparisons in every pass.

  Thus, complexity is O (n log n)

Searching

It is process of checking and finding an element from list of elements.

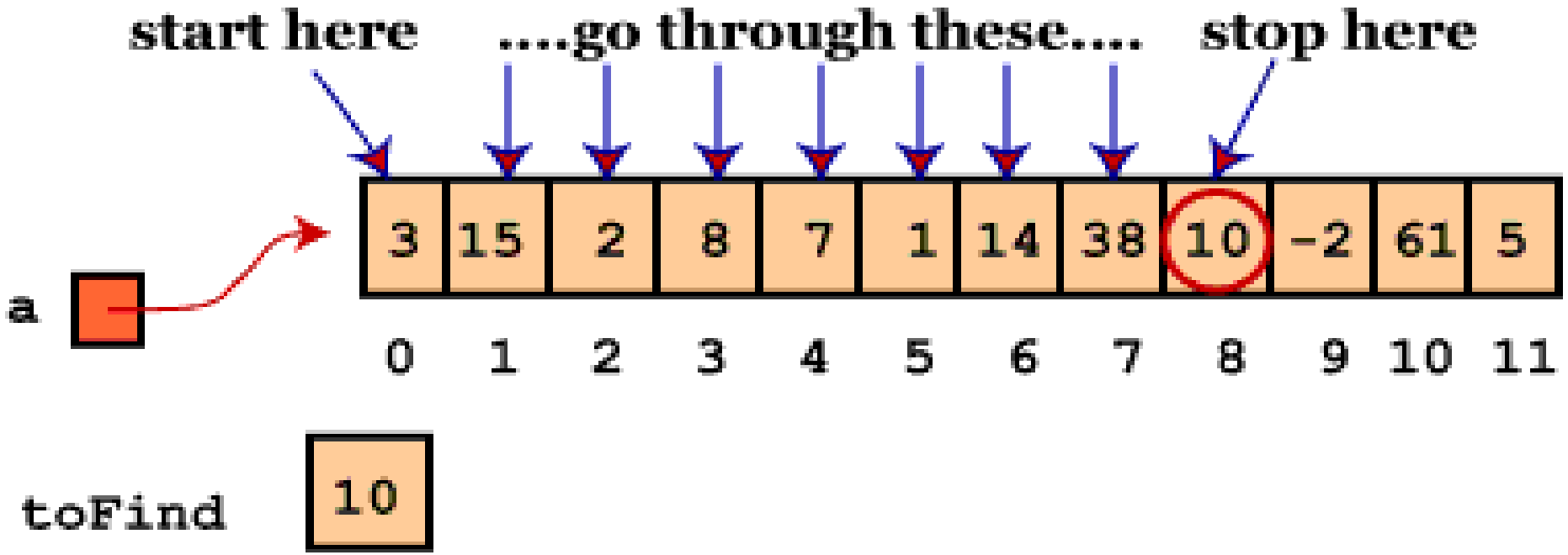
- Linear search
- Binary search

Sequential Search

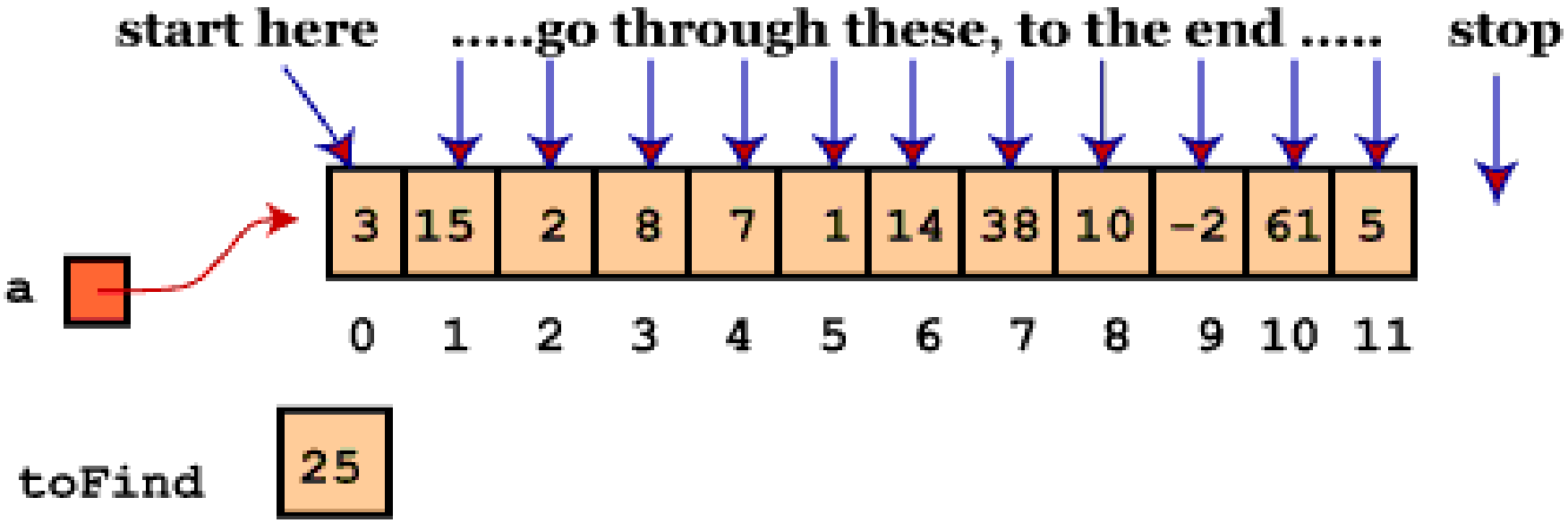
- Also called linear search.
- The algorithm searches key by comparing it with each element in turn.

Marwadi University
Marwadi Chandarana Group

Linear Search



Linear Search



Linear Search

Algorithm linear (a, n, key)

// key is data to be searched in array a of size length

Pre: Unsorted list of length n.

Post: If found, return position of key in array a. If key not present in list, return negative value

```
1.  for i = 0 to (n - 1)  do
        if  (key == a[i])
            return i
2.  return -1
```

Complexity

- Best Case: - O (1)
  - Item found at first position.
- Worst Case: - O(n)
  - Item found at last position.
- Average Case: - O(n)
  - On an average (n+1)/2 comparisons required

Practical :

Given a target value, perform Sequential Search on an array of numbers .

Marwadi
University
Marwadi Chandarana Group

Binary Search

Binary Search

- When array is not sorted, sequential search is only the option for sorting.
- But if array is sorted binary search is more efficient algorithm.
- It starts with the testing of data at the middle of an array.
- Target may be in first half or second half of the array.
- To find middle of the list, beginning of the list and end of the list are used.

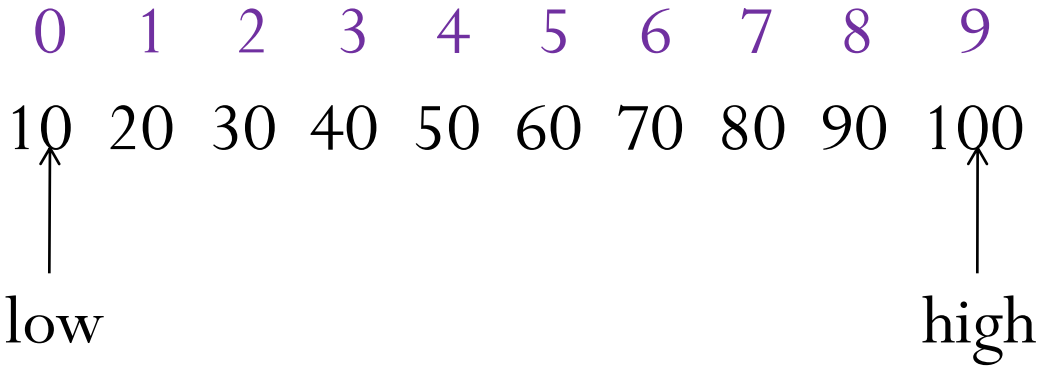
- Search key = 80 in given array.

10 20 30 40 50 60 70 80 90 100

Binary Search

Key = 80

n = 10



low = 0

high = 9

mid = (low + high) / 2 = 4

a[mid] = 50

80>50, low = mid + 1 = 4+1 = 5

Binary Search

Key = 80

n = 10

0	1	2	3	4	5	6	7	8	9
10	20	30	40	50	60	70	80	90	100

low = 5

high = 9

mid = (low + high) / 2 = 7

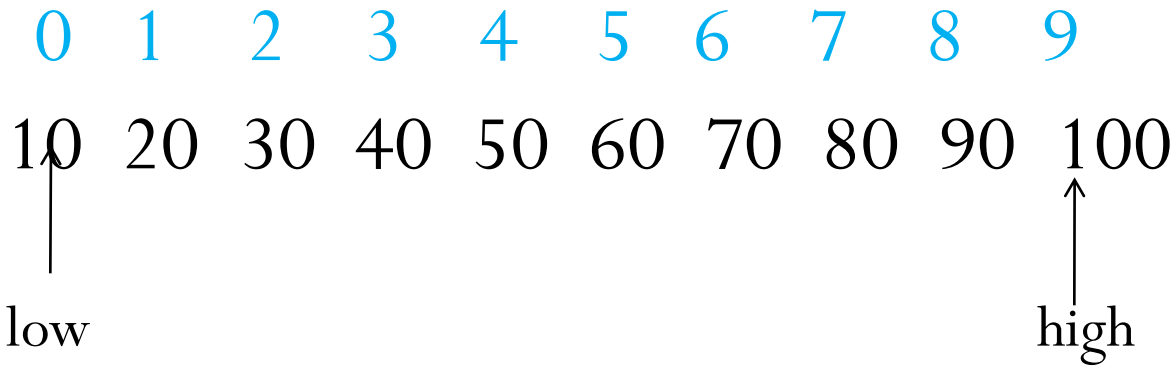
a[mid] = 80 //found

- Search key = 30 in given array.

10 20 30 40 50 60 70 80 90 100

Key = 30
n = 10

Binary Search



low = 0

high = 9

mid = (low + high) / 2

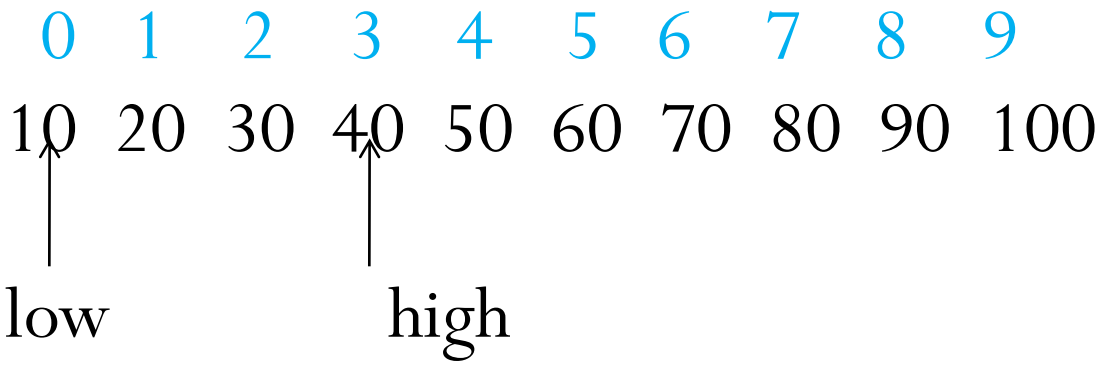
a[mid] = 50

30 < 50, high = mid – 1= 4 – 1 = 3

Binary Search

Key = 30

n = 10



low = 0

high = 3

mid = (low + high) / 2

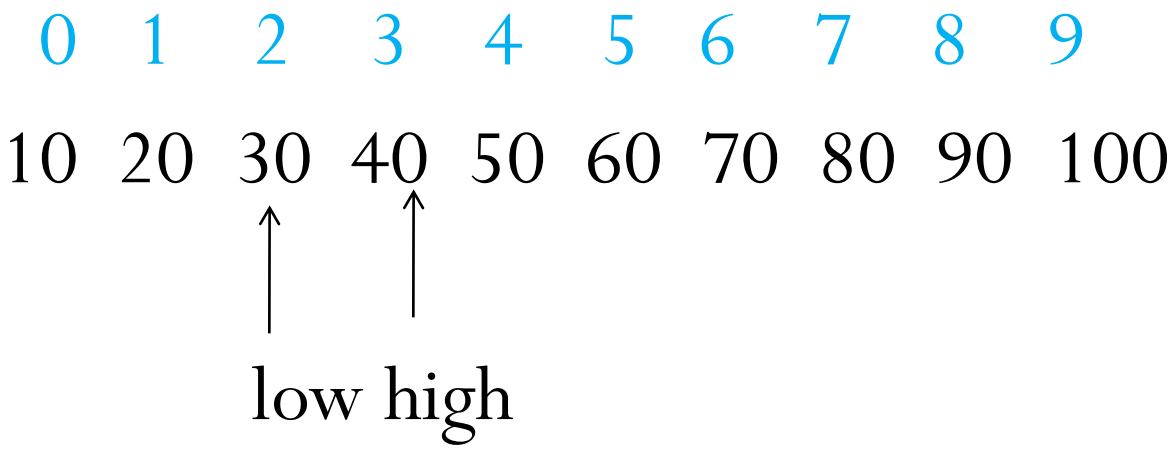
a[mid] = 20

30 > 20, low = mid +1 = 1+1 = 2

Binary Search

Key = 30

n = 10



low = 2

high = 3

mid = (low + high) / 2 = 2

a[mid] = 30 //found

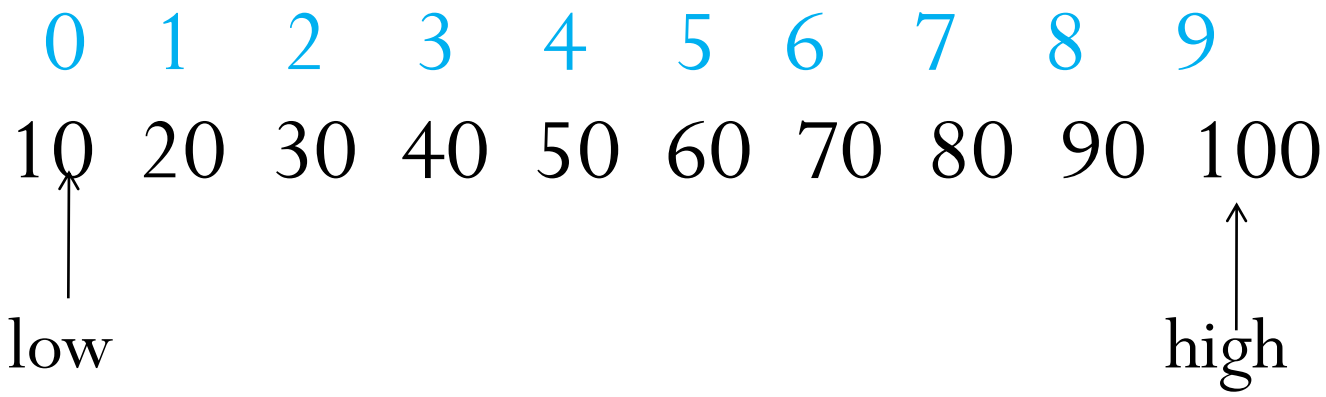
- Search key = 35 in given array.

10 20 30 40 50 60 70 80 90 100

Binary Search

Key = 35

n = 10



low = 0

high = 9

mid = (low + high) / 2 = 4

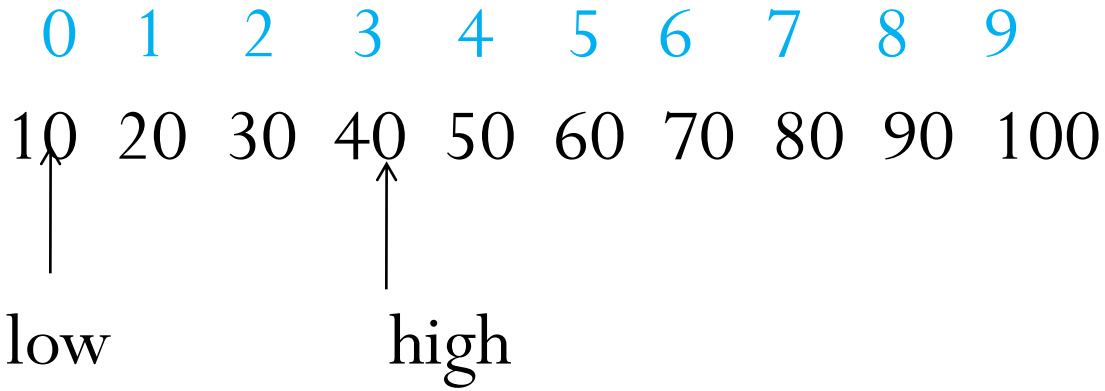
a[mid] = 50

35 < 50, high = mid – 1 = 4 – 1 = 3

Key = 35

n = 10

Binary Search



low = 0

high = 3

mid = (low + high) / 2 = 1

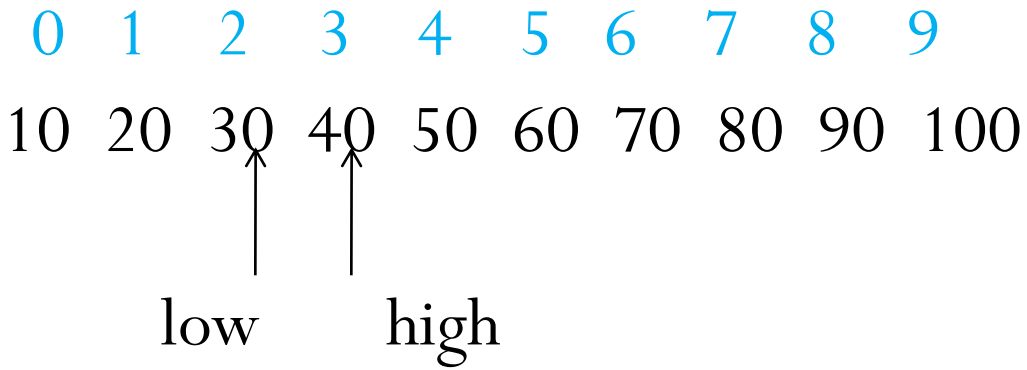
a[mid] = 20

35 > 20, low = mid + 1 = 1 + 1 = 2

Key = 35

n = 10

Binary Search



low = 2

high = 3

mid = (low + high) / 2 = 2

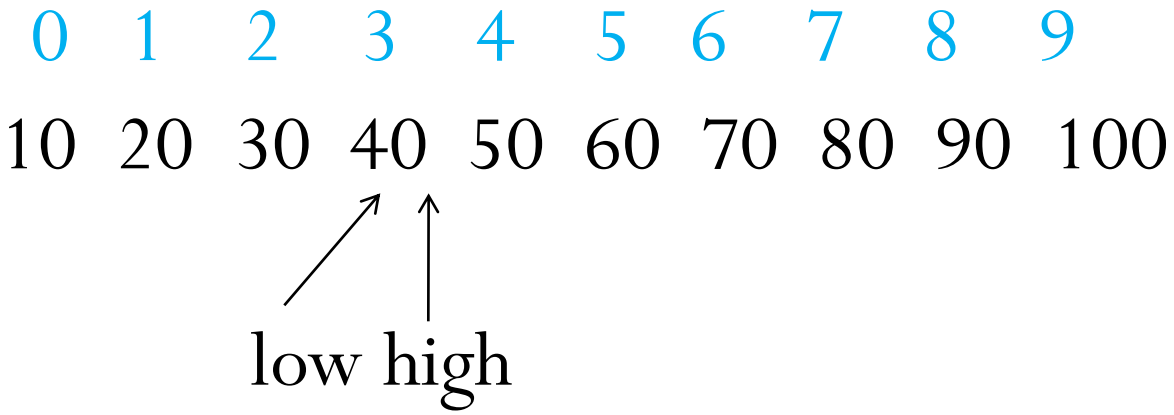
a[mid] = 30

35 > 30, low = mid +1 = 2 + 1 = 3

Key = 35

n = 10

Binary Search



low = 3

high = 3

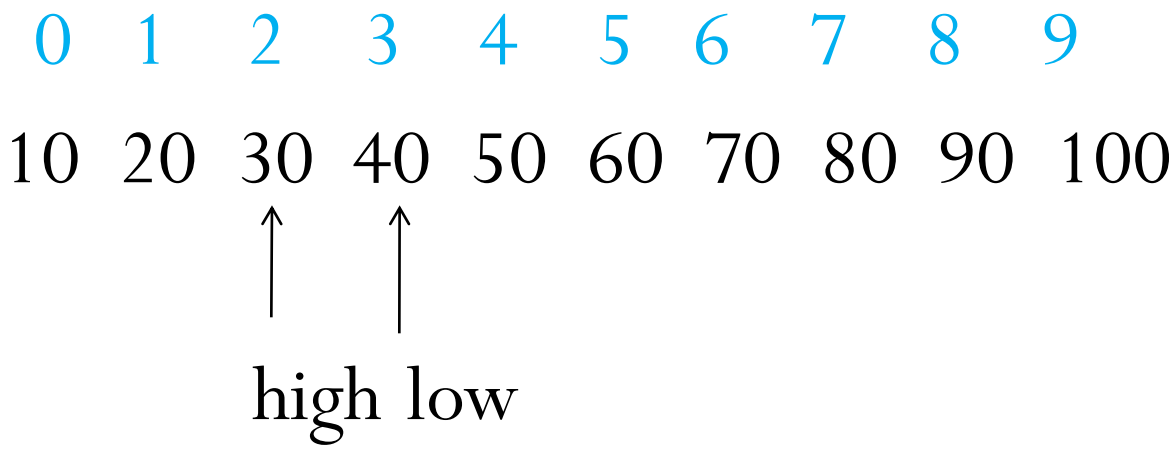
mid = (low + high) / 2 = 3

a[mid] = 40

35 < 40, high = mid – 1 = 3 – 1 = 2

Key = 35
n = 10

Binary Search



low = 3

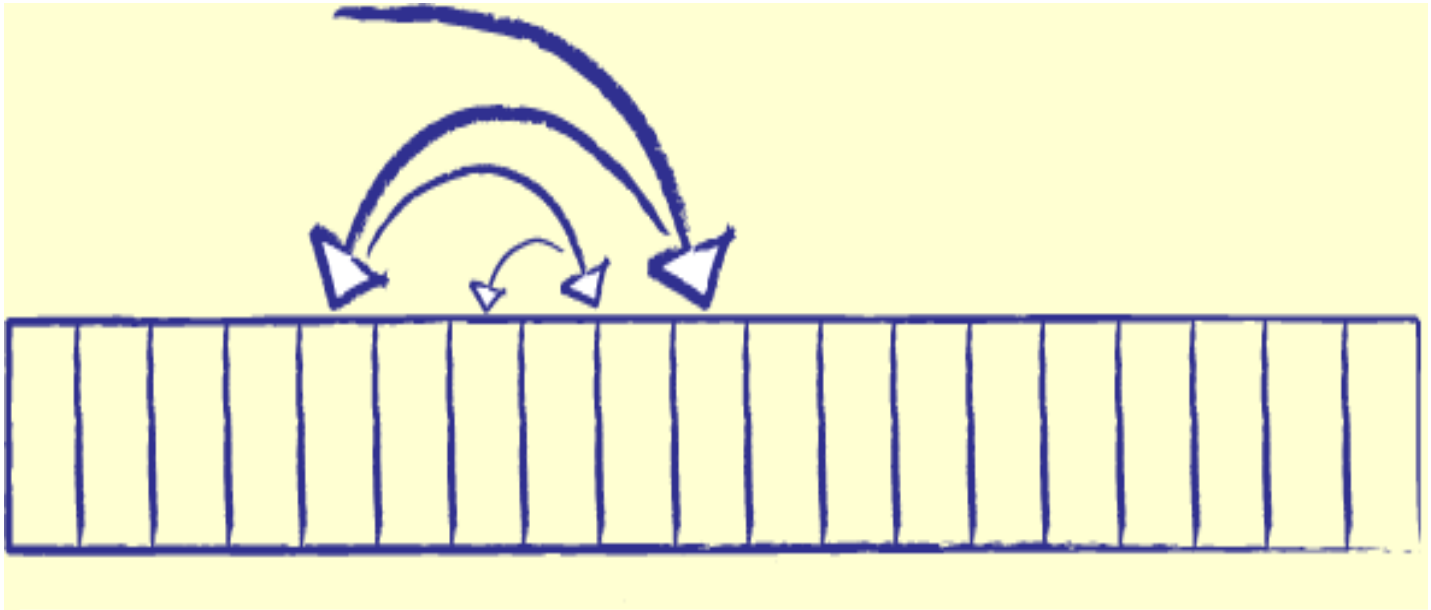
high = 2

low > high //No further execution

//element not found

Marwadi University
Marwadi Chandarana Group

Binary Search



Marwadi University
Marwadi Chandarana Group

Binary Search

Binary Search

Algorithm binary_search (a, n, key)

// key - data to be searched in array a of size n

Pre: Sorted list of length n.

Post: If present, return position of key in array a; Else return -1

1. low =0
2. high = n-1
3. while (low <= high)
    1. mid = (low + high)/2
    2. if (key == a[mid])
        return mid
    3. if (key < a[mid])
        high = mid -1
    4. else
        low = mid + 1
4. return -1

Example

The given array is as follows.

11 22 30 33 40 44 55 60 66 77 80 88 99

A) Search key = 40

B) Search key = 85

Complexity

- Complexity: O ($\log_2 n$)
- Each comparison in binary search reduces the number of possible elements by factor of 2.
- Say array size n $=2^m$, then number of passes = m
- When n $=2^m$, thus m = $\log_2 n$. (e.g. if n=64, m = 6)
- During each pass, maximum only 2 operations will be required (one to check equality & if that fails another to check in which part of list is the key to be further searched)
- Thus 2m comparisons i.e. 2 $\log_2 n$. Thus O($\log_2 n$)

Recursive Binary Search

Marwadi University
Marwadi Chandarana Group

BinarySearch(a, key, low, high)

```
1.   if (low > high)
        return -1                          // not found
2.  mid = (low + high) / 2
3.  if (key < a[mid] )
        return BinarySearch ( a, key, low,  mid-1)
4.  else if (key > a[mid])
        return BinarySearch ( a, key, mid+1,  high)
5.  else
        return mid                         // found
```

Question:

- Write an algorithm for binary search.For the following array of 10 elements search 167 using binary search.Also trace the steps.

25 62 71 86 92 106 110 134 167 178

Difference between linear and Binary search

	Linear search	Binary Search
1	Data may be any order	Data must be in sorted order
2	Time complexity O(n)	O(log n)
3	Access is slower	Access is faster
4	Search works by looking each element in list until either it finds the target or reaches the end.	As input data is sorted we can leverage this information to decrease the number of item we need to look at to find our target. We know that if we look random item in the data and that item is greater than our target , then all items to the right of that item will also be greater than our target . This means that only need to look at left part of the data. Each time we search for the target and eliminate half of the remaining item

Marwadi University
Marwadi Chandarana Group

Thank You...