DATA STRUCTURE

CHAPTER 9

SORTING AND SEARCHING

Sorting

- To arrange a list of elements (data) in ascending or descending order is called sorting. There are two types of sorting:
  1. Internal Sorting and
  2. External Sorting

Types of sorting

- **Internal Sorting:**

  The method (algorithm) which sorts list of data that is small enough to fit entirely in primary (internal) memory, is called internal sorting.

- **External Sorting:**

  The method (algorithm) which sorts list (file) of data that can not fit entirely in primary memory, that means to sort the entire list the method uses external memory, is called external sorting.

Classes of Internal Sorting

- **Classes of Internal Sorting**
- Exchange Sort:

  Selection sort, Insertion sort, Bubble sort
- Divide and Conquer Sort(External):

  Merge sort, Quick sort
- Tree Sort

  :Heap sort, Tournament sort
- Non-comparison based Sort

  :Radix sort, Bucket sort etc.

Internal sort

SELECTION SORT

Selection Sort [contd.]

- Using steps we can write the method as follows.

  i. Given a list of data. Find out the smallest data from the list. Remember the position/index of the smallest data. Place the smallest in first position and the data of the first position in the position of the smallest data.

  ii. Repeat the process for the list except data in first position, and so on.

Selection Sort [contd.]

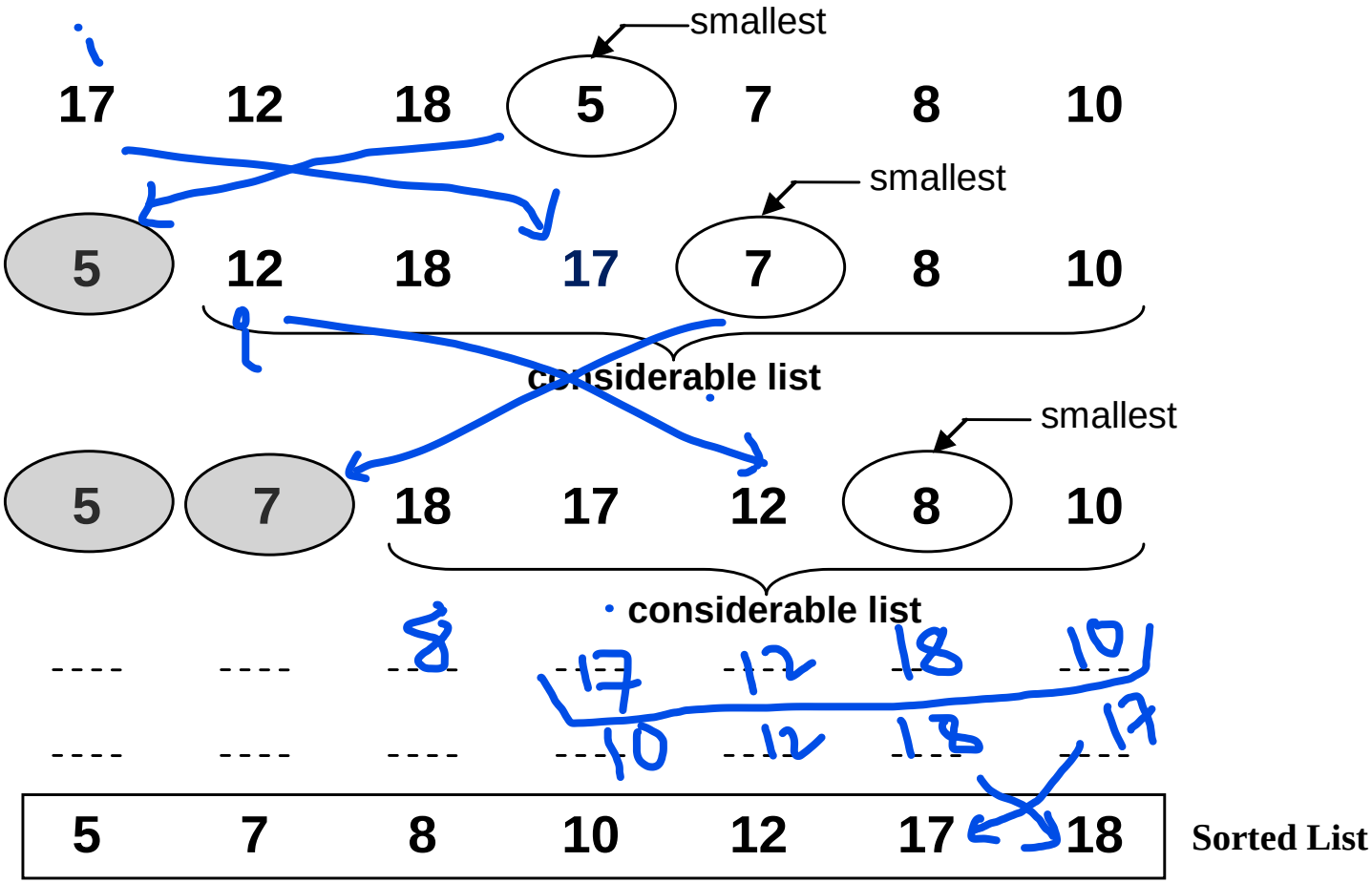


Figure 9.3: Pictorial view of Selection Sort

Algorithm for Selection Sort

1. Input Array $A[1……n]$
2. (i). for (i = 1 to n -1)
   {
   small_index = i;
   (ii). for (j = i + 1 to n)
   {
   if ($A[j] < A[small\_index]$) then $small\_index = j$;
   } // end of second for
   $temp = A[i]$;
   $A[i] \leftarrow A$[small_index];
   A[small_index] $\leftarrow temp$;
   } // end of first for
3. Output: sorted list.

Complexity of selection sort

For first phase, number of comparison is *n*-1; second phase, number of comparison is *n*-2; third phase, number of comparison is *n*-3 and so on. Then,

No. of comparisons = (n-1) + (n-2) + (n-3) + … … … … … + 1

$$= \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Therefore, complexity = $O(n^2)$

Space = O(1)

Internal sort

INSERTION SORT

Insertion Sort [contd.]

- Using steps we can describe the process as follows.
  i. Given a list of elements (data).
  ii. We have to insert a data into its correct position by moving all data (before it) to the right (that are greater than the data which is being considered at this moment).
  iii. By repeating Step-*ii* for all considerable data we can arrange the whole list in ascending order.

Insertion Sort [contd.]

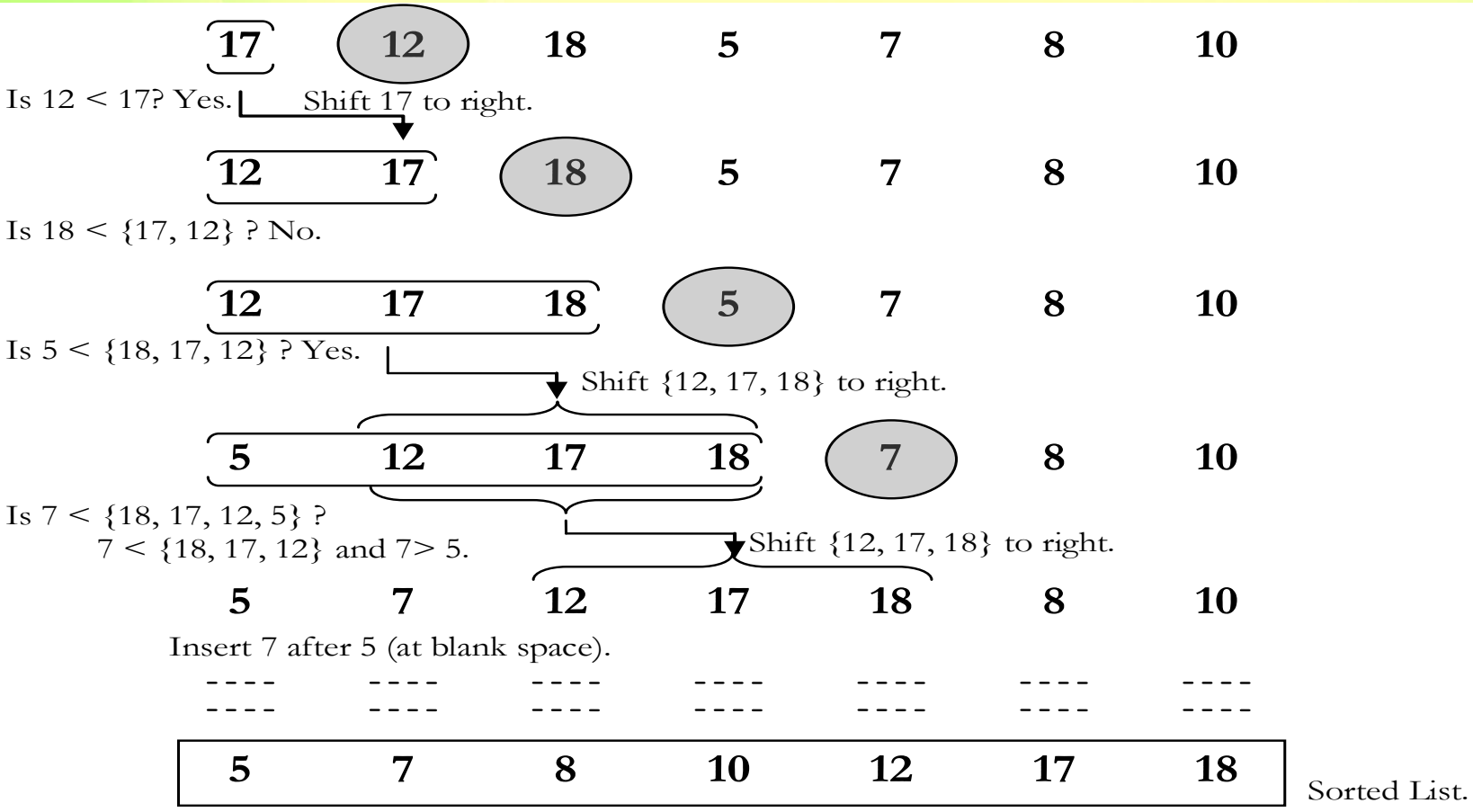


Figure 9.4: Pictorial view of Insertion Sort

Insertion Sort [contd.]

Algorithm 9.4: Algorithm for insertion sorting

1. Input Array A[1……*n*]
2. (i). for (*j* = 2 to *n*)
   {
   key-value = A [*j*];
   i = *j*-1;
   (ii). while (*i* > 0 and A[*i*] > *key-value*)
   {
   A[*i* + 1] ← A[*i*];
   i = *i* -1;
   } // end of while
   A[*i* + 1] ← *key-value*;
   } //end of for
3. Output: sorted list.

Complexity of Insertion Sort

- No. of comparisons = 1 + 2 + 3 + ... + (n-1)

$$= \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Therefore, complexity = $O(n^2)$ // means less than n^2

Internal sort

MERGE SORT

Merge Sort

- The merge sort algorithm is a divide and conquer method. It operates as follows:
  - **Divide**: Divide the n-element sequence into two n/2 element sequences. It divide the sequence of data recursively. Division continued until it gets single element. After getting two single element it merge them. Next it divide another part and make it two parts of single elements and merge them.
  - Conquer: it combine or merge two sorted parts (single elements are sorted) and make a single part by arranging (sorting) data.

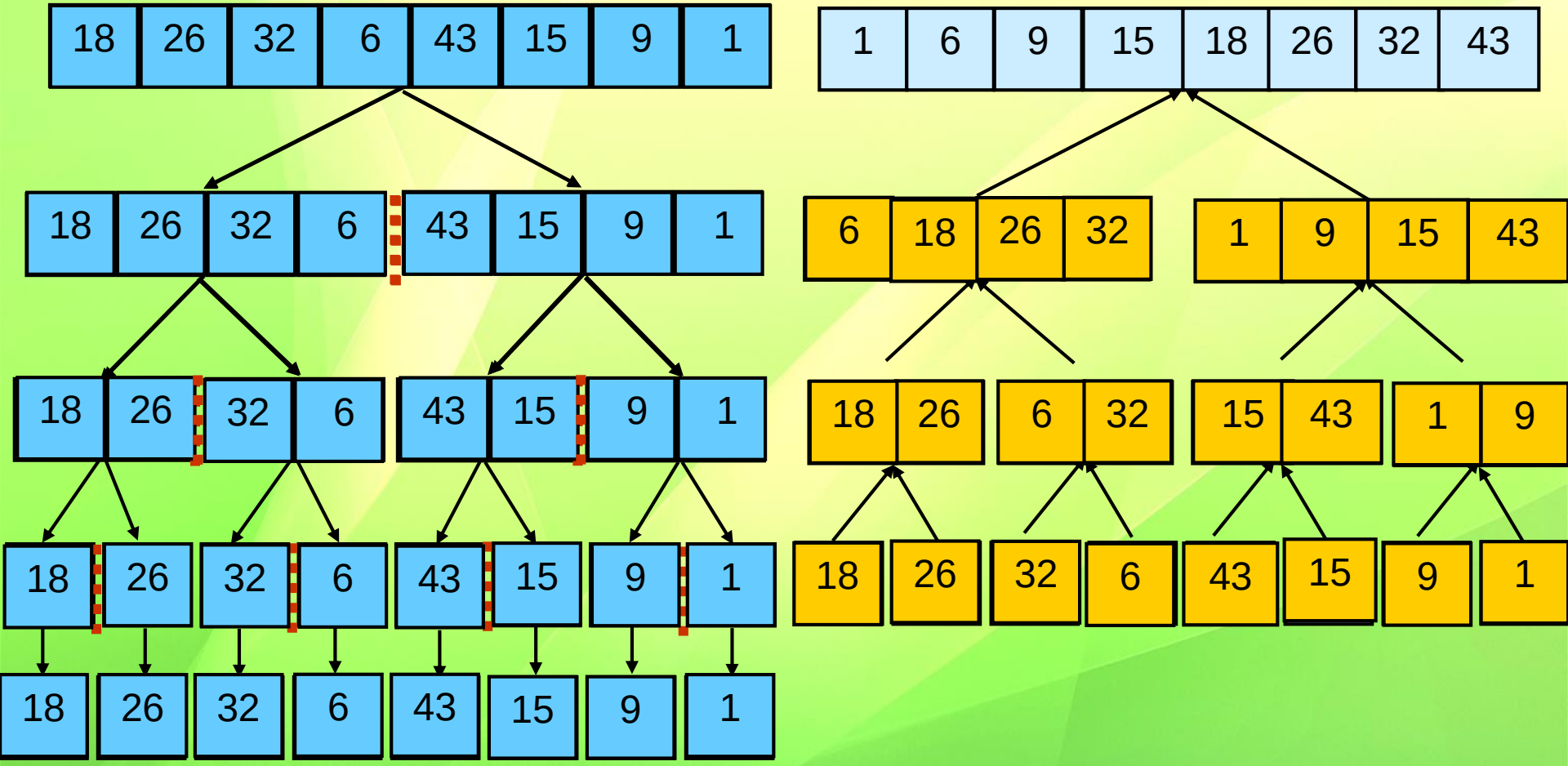
Merge Sort

- Merge sort is a divide and conquer method.
- It works by dividing the list into two parts, sorts the parts and merges them together.

Merge sort is a recursive procedure (function).

- If a function calls itself repeatedly, then the function is called recursive function.
- A disadvantage of merge sort is that it requires extra spaces (array) for merging.

Merge Sort - Example



Merging procedure

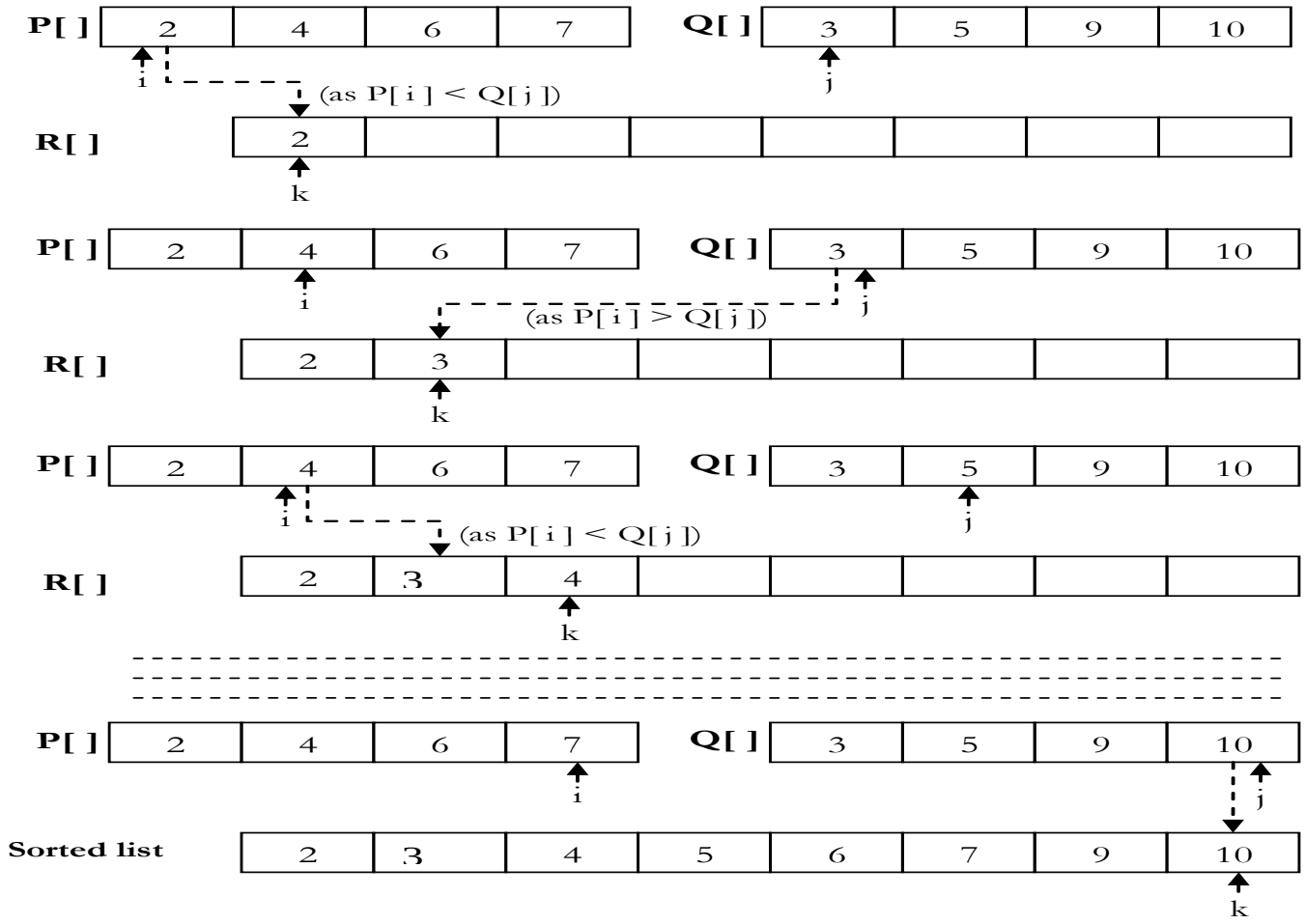


Figure 9.7: Pictorial view of merging process

Algorithm of Merge Sort

Algorithm for merge sort

```
merge_sort (A,  f, l)
{
if (f < l)
   {
   m = (f + l) / 2;
   merge_sort (A, f, m);
   merge_sort (A, m + 1, l);
   Merge (A, f, m, l);
   } // end of if
}
```

Merging function

- Here *A* is an array, *f* is the first index and *l* is the last index.

```
Merge (A, f, m, l)
    {
    Take an array T[1 . . . l];
    i = f; j = m + 1; k = f;
    while (i ≤ m and j ≤ l)
      {
      if A[i] ≤ A[j], then T[k] ← A[i]; i = i +1;
      else T[k] ← A[j]; j = j + 1;
      //end of if
      k = k + 1;
      } //end of while
    //copy rest of the data
```

CONTINUED >>

Merging function[contd.]

```
if (i > m) then
        for (b = j to l) do
                        {
                        T[k] ← A[b]; k = k + 1;
                        }
     else for (b = i to m) do
                        {
                        T[k] ← A[b]; k = k + 1;
                        }
     for(i=f to l)    //to copy each merging result
      {
    A[i]=T[i];
     }
} //end of function
```

Analysis of Merge Sort

If the time for the merging operation is proportional to *n* then the computing time for *merge sort* is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1 \\ 2\,T(n/2) + n & n > 1 \end{cases}$$

When *n* is a power of 2, $n = 2^k$; we can solve this equation by successive substitutions.

Analysis of merge sort

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$= 2\left\{2T\left(\frac{n}{4}\right) + \frac{n}{2}\right\} + n$$

$$= 4T\left(\frac{n}{4}\right) + 2n$$

$$= 8T\left(\frac{n}{8}\right) + 3n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

$$\cdots\cdots$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

Complexity of merge sort

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

$$= 2^k T(1) + kn \qquad [\, n = 2^k \,]$$

$$= an + kn \qquad [\, T(1) = a \,]$$

$$= an + n\log n \qquad [\, k = \log n \,]$$

$$= O(n\log n)$$

Internal sort

QUICK SORT

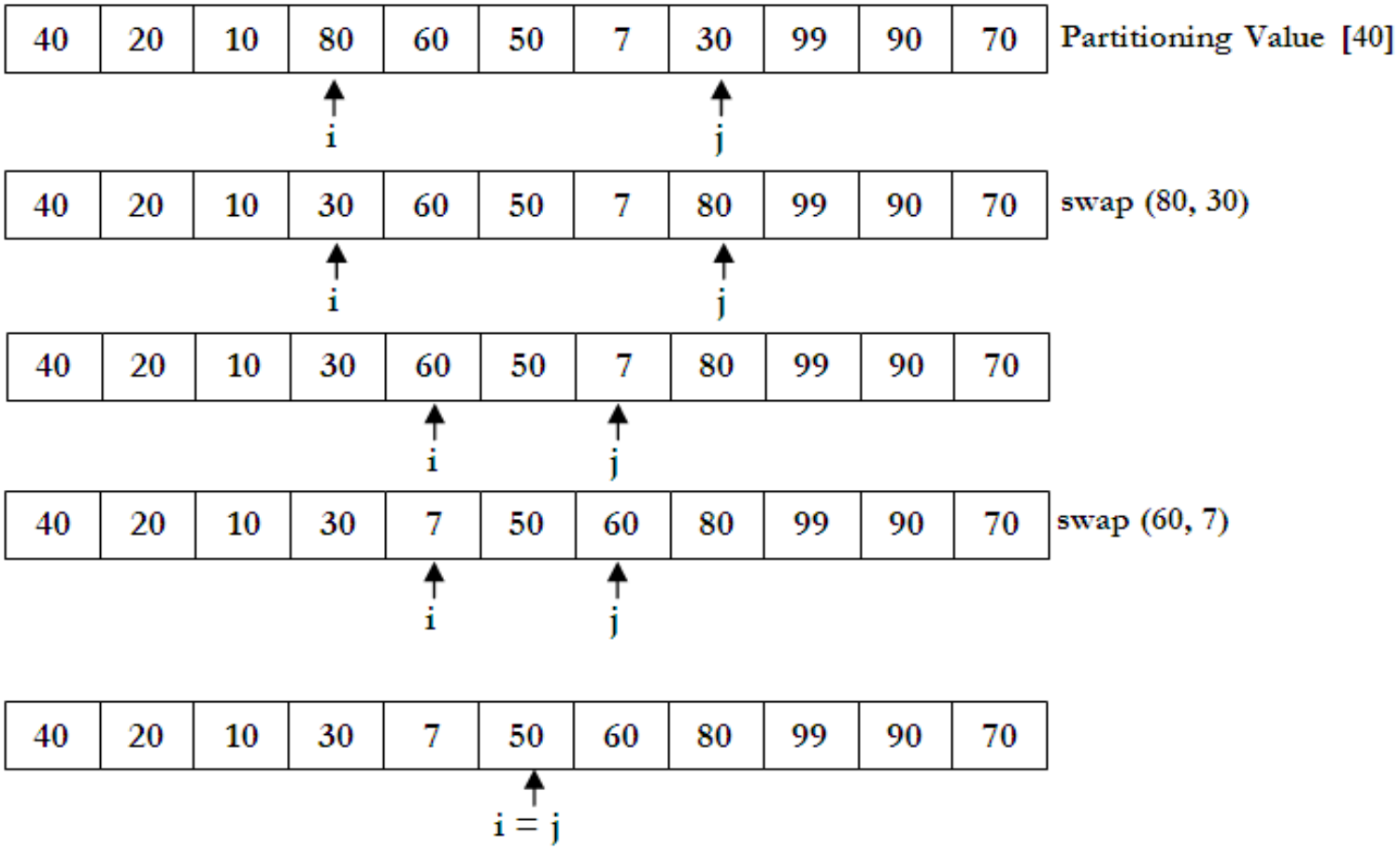
Quick Sort

- Quick sort is a divide and conquer method.
- In this method one element is to be chosen as partitioning element.
- Divide the whole list of data into two parts with respect to the partitioning element.
- The data which are less than or equal to the partitioning element, will remain in the first sub-list or first part and the data which are greater than the partitioning element will remain in the second sub-list (second part).

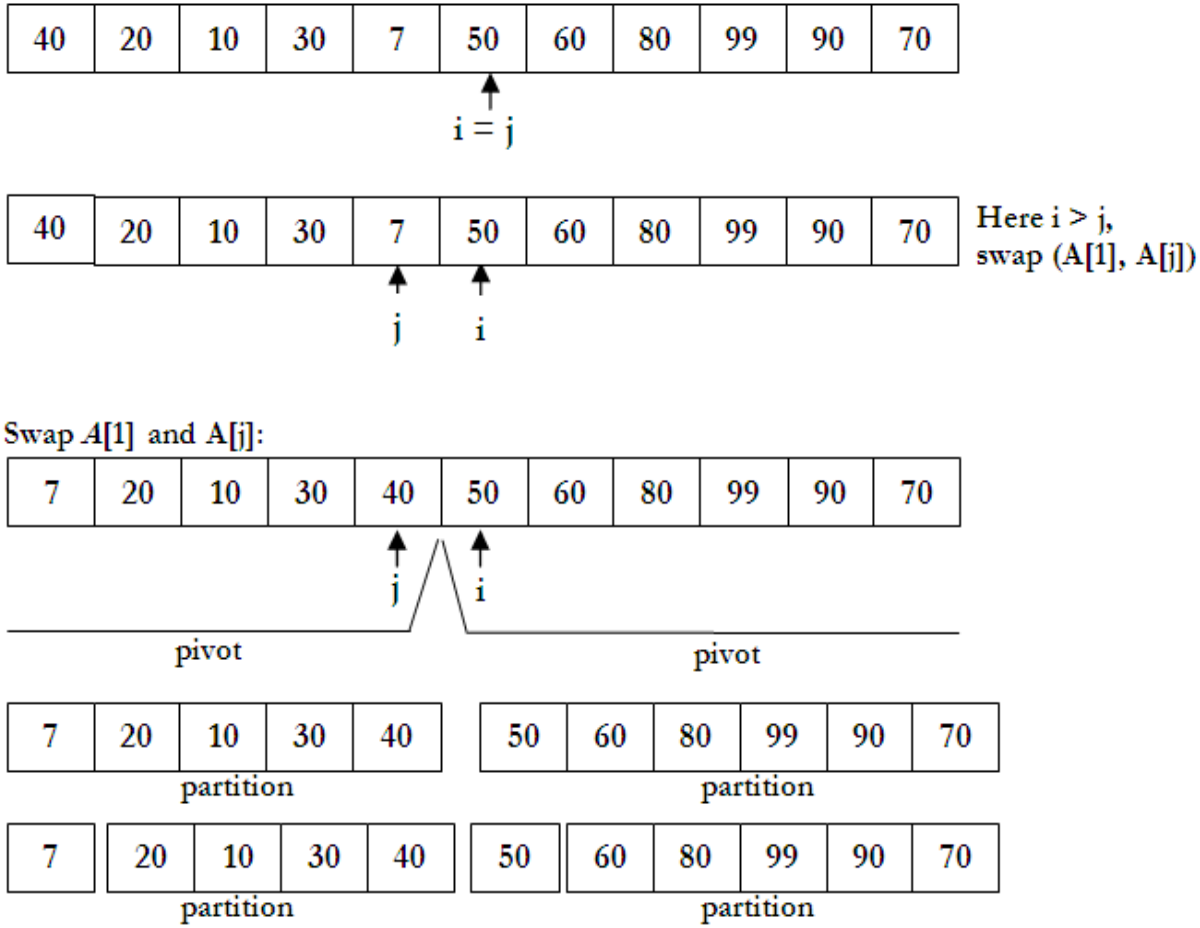
Quick sort

- If we find any data (in the first part) which is greater than the partitioning value that will be transferred to the second part.
- In the second part, if we find any data which is less than the partitioning element, that will be transferred to the first part.
- Transferring of data have been done by exchanging the positions of the data found in first part and second part.
  - By repeating the process, we can sort the whole list of data.

Example of Quick Sort



Example of Quick Sort [contd.]



Example of Quick Sort [contd.]

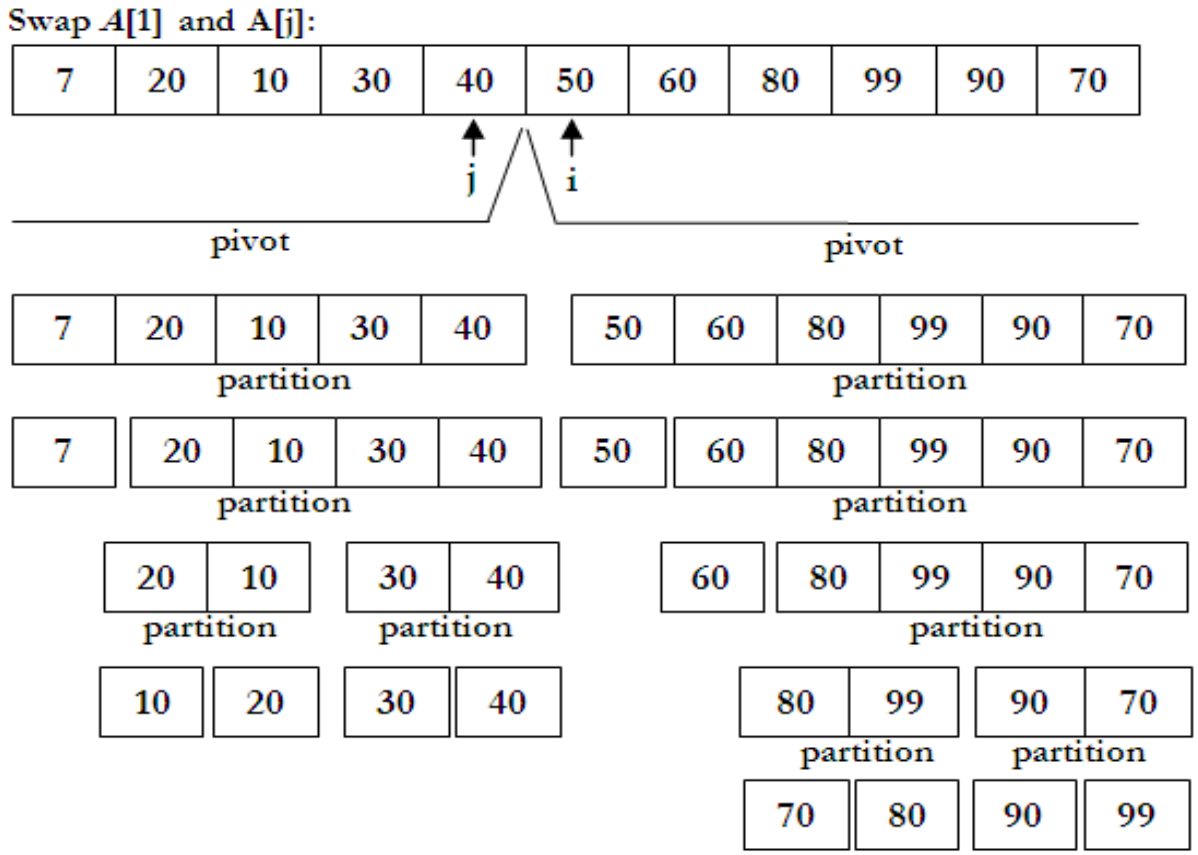


Figure 9.8: Pictorial view of Quicksort Partitioning

Algorithm of Quick Sort

- Algorithm for quick sort

```
Quick_sort (f, l)
  {
  if (f < l)
    {
    j = makepart (a, f, l + 1) ;
    Quick_sort (f, j - 1) ;
    Quick_sort (j + 1, l);
    }
  }
```

Partitioning function

```
makepart (a,  first, last)
{
    part_value = a[first]; i = first; j = last;
    do
    {
          do
             {
              i = i +1;
              } while (a[i] ≤ part_value);
          do
             {
              j = j  - 1;
             } while (a[i] > part_value);
       if (i < j) then interchange (a[i], a[j]);
      } while (i < j);
    a[first] = a[j]; a[j] = part_value;
    return j;
}
```

- Here, *first* is the first index and *last* is the last index; and *a* is the array. And *part_value* is the partitioning element (first element for first iteration).

Analysis of Quick Sort

Let us consider *T(n)* be time complexity with respect to the number of comparisons in average case.

$$T(n) = n + 1 + \frac{1}{n}\sum_{i=1}^{n}\{T(i-1) + T(n-i)\} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

Here $(n+1)$ comparisons is required for first round, $\frac{1}{n}$ is the probability to choose partitioning element. After partitioning, if $i-1$ elements are in one part, then $n-i$ elements are in other part.

Note that, $T(0) = T(1) = 0$. By putting the value of *i = 1, 2, 3,……, n,* the equation (1) can be rewritten as follows:

$$T(n) = n + 1 + \frac{2}{n}\{T(0) + T(1) + \ldots + T(n-1)\}$$

Analysis of Quick Sort [contd.]

Multiplying both sides by n we obtain

$$nT(n) = n(n+1) + 2\{T(0) + T(1) + \ldots + T(n-1)\} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

Replacing n by $n-1$ in (2) we get

$$(n-1)T(n-1) = n(n-1) + 2\{T(0) + T(1) + \ldots + T(n-2)\} \ldots\ldots\ldots\ldots\ldots (3)$$

Subtracting (3) from (2) we can write

$$nT(n) - (n-1)T(n-1) = 2n + 2T(n-1)$$

- $nT(n) = (n-1)T(n-1) + 2n + 2T(n-1)$
- $nT(n) = T(n-1)\{n-1+2\} + 2n$
- $nT(n) = (n+1)T(n-1) + 2n$
- $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$ [dividing by n $(n + 1)$]

Analysis of Quick Sort [contd.]

Repeatedly using this to substitute for *T(n - 2)*, *T(n - 3)*, . . . we get

$$\frac{T(n)}{n+1} = \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{T(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{T(n-4)}{n-3} + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

.

.

.

$$= \frac{T\{n-(n-1)\}}{\{n-(n-2)\}} + \frac{2}{\{n-(n-3)\}} + \frac{2}{\{n-(n-4)\}} + \ldots + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{T(1)}{2} + \frac{2}{3} + \frac{2}{4} + \ldots + \frac{2}{n} + \frac{2}{n+1}$$

Analysis of Quick Sort [contd.]

$$= \frac{T(1)}{2} + 2\left(\frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n+1}\right)$$

$$= \frac{T(1)}{2} + 2\sum_{x=3}^{n+1} \frac{1}{x}$$

$$= 2\sum_{x=3}^{n+1} \frac{1}{x}$$

Here $\sum_{x=3}^{n+1} \frac{1}{x} \le \int_{2}^{n+1} \frac{1}{x} dx = \log_e(n+1) - \log_e 2$

Therefore, $\frac{T(n)}{n+1} \le 2\left[\log_e(n+1) - \log_e 2\right]$

$$T(n) \le 2(n+1)\left[\log_e(n+1) - \log_e 2\right]$$

$$T(n) \le 2n \log_e n + \ldots$$

$$T(n) = O(n \log_e n)$$

Complexity of Quick sort

- That means, the quicksort algorithm takes $O\ (n \log_e n)$ time in average case. On the other hand the worst case time is $O\ (n^2)$.

- Remember that merge sort takes $O\ (n \log_2 n)$ time in average and worst cases, whereas quick sort takes $O\ (n \log_e n)$ time in average case and $O\ (n^2)$ time in worst case.

Summary of Complexities

The complexity of sorting algortihm:

i) The complexity selection sort is $O(n^2)$.

ii) The complexity of insertion sort is $O(n^2)$.

iii) The complexity of merge sort $O(n\log_2 n)$ both in average and worst cases.

iv) The complexity of quick sort $O(n\log_e n)$ time in average case and $O(n^2)$ in worst case .

Sorting

EXTERNAL SORT

External Sorting

- External sorting is required when the number of records (data) to be stored is larger than the computer can hold in its internal (main) memory.
- Now-a-days, to sort extremely large data is becoming more and more important for large corporations, banks and government institutions.
- External sorting is quite different from internal sorting, even though the problem in both cases is to sort a given list of data into increasing or decreasing order.
- The most common external sorting algorithm used is still the Merge sort.

External Sorting [contd.]

- In external sorting method, at first the sorted runs (sorted sub-files) are produced, and then the sorted runs are merged to produce single run which gives us a sorted file (list of data).
- The runs can be produced using any internal sorting algorithm like quick sort.

External Sorting [contd.]

- Let us consider that we have to sort three thousand records R_1, R_2,……,R_{3000} and each record is 20 words long.
- It is assumed that only one thousand of the records will fit in the internal memory of our computer at a time.
- Now this data can be sorted in the following ways. Suppose, the runs are written in different files on a disk.
- According to our example there will be three runs and let there are three files as follows

- file-1: R_1, R_2, R_3,……,R_{1000}
- file-2: R_{1001}, R_{1002},……,R_{2000}
- file-3: R_{2001}, R_{2002},……,R_{3000}

External Sorting [contd.]

- Now, we shall produce a run by merging the file-1 and file-2 and these will write in file-4.
- Again we merge the file-3 and file-4 and produce a single run which may be written to file-1.

- file-4: $R_1, R_2, \ldots\ldots\ldots, R_{2000}$
- file-3: $R_{2001}, \ldots\ldots\ldots, R_{3000}$
- file-1: $R_1, R_2, \ldots\ldots\ldots, R_{3000}$

External Sorting [contd.]

- During merging phase we shall read some records suppose 500 records from file-1 and 500 records from the file-2 and merge them.
- Merging file will be written to file-4.
- Again we shall read the last 500 records from the file-1 and 500 records from file-2 and merge them.
- The merging records will be written to file-4. Similarly we merge the file-4 and the file-3.

Searching

- Searching means to find out or locate any element from a given list of elements.
- To identify or locate an element or position of the element from a list of elements is called searching.
- Here we shall study two types of searching
  - >Linear searching
  - >Binary Searching

Linear Searching [contd.]

Suppose given a list of numbers as follows:

17 12 18 5 7 8 10

We have to find out whether a number, 7 is in the list or not.

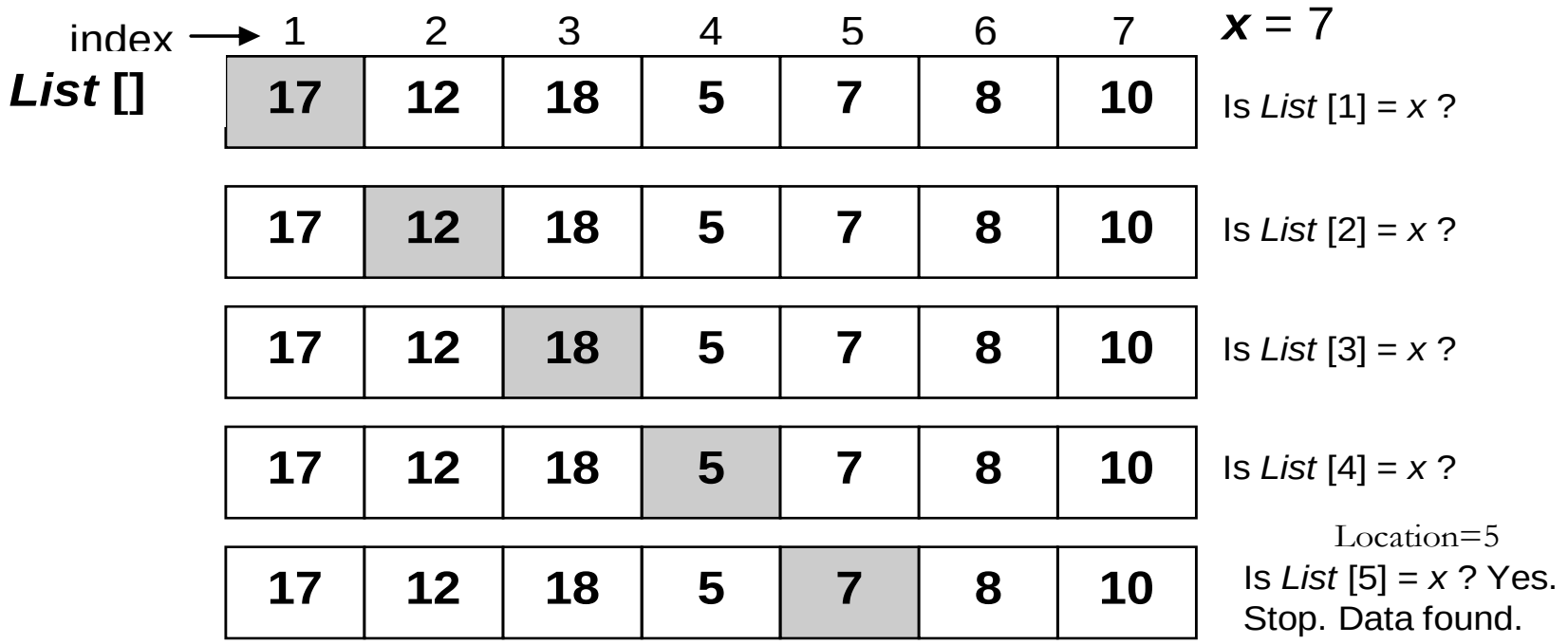


Figure 9.1: Pictorial view of Linear Searching (shaded items showed the searching sequence)

Algorithm for linear searching

1. Input a list and an item (element to be found)
   A [1…*n*]; *item* = *x*; Location = 0
2. Search the list to find the target element
   for (*i* = 1; *i* ≤ *n*; *i* = *i* + 1)
   {
   if (*A*[*i*] = =item)
   {
   print “Found ”;
   location = i
   Stop searching;
   }
   }
3. if (*i* > *n*) print “Not Found ”;
4. Output : “Found” or “Not Found”.

Complexity for linear searching

- In average case, complexity $= \frac{1+2+3+.....+n}{n}$

$$= \frac{n(n+1)}{2n}$$

$$= \frac{1}{2}n + \frac{1}{2}$$

$$= O(n)$$

Binary Searching

- prerequisite for binary searching is that the elements must be arranged either in ascending or in descending order.

- **Problem**
  - Given list of data elements arranged in ascending or descending order, locate (find the position of) a particular (target) element from the list.

  Remember: It **is not** Binary Search Tree (BST). BST is a data structure, whereas binary search is a method or technique (not a data structure).

Binary Searching [contd.]

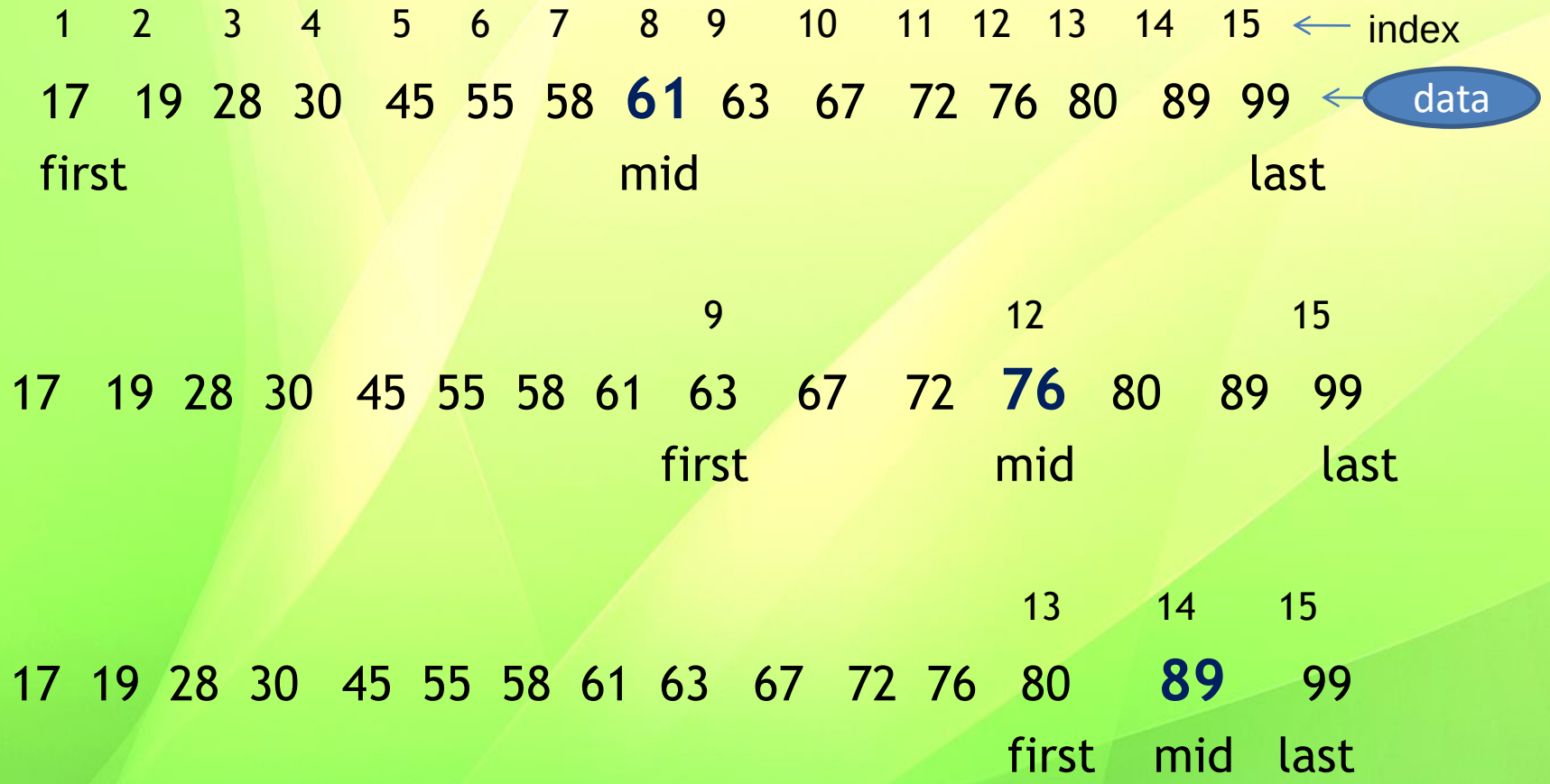
- **Example:**

- Suppose we have a list of numbers as follows

17 19 28 30 45 55 58 61 63 67 72 76 80 89 99

- And we have to find out 89 from the list

mid = (first + last)/2;


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
index
17 19 28 30 45 55 58 61 63 67 72 76 80 89 99
data
first
mid
last
9
12
15
17 19 28 30 45 55 58 61 63 67 72 76 80 89 99
first
mid
last
13
14
15
17 19 28 30 45 55 58 61 63 67 72 76 80 89 99
first
mid
last

X = 89 Found

Algorithm for Binary Searching (pseudocode)

Input *A*[1 . . . *m*], *x*; //*A* is an array with size *m* and *x* is the target
//element

1. *first* = 1, *last* = *m*;
2. while (*first* ≤ *last*)

{

mid = (*first* + *last*) / 2;

(i) if (*x* = *A*[*mid*]), then print *mid*; // target element = A[*mid*]

(ii) else if (*x* < *A*[*mid*]) then *last* = *mid* - 1;

(iii) else first = *mid* +1;

}

3. if(first > last) , print “not found”;
4. Output: mid or “not found”

Complexity of binary search

- Suppose, there are n elements in the list.
- Let $n = 2^k$; then $k = log_2 n$.
- Therefore, complexity = $O\ (log_2\ n)$

Summary of Complexities

The complexity of searching algorithm:

i) The complexity of linear search is ***O(n)***.

ii) The complexity of binary search is $O(log_2 n)$.

THANK YOU.