

Python Assignment No. 4

Q1. Explain creating list with example.

Ans. A list is a sequence of values called items or elements. The elements can be of any type. The structure of a list is similar to the structure of a string.

CREATING LISTS

The List class defines lists. A programmer can use a list's constructor to create a list. Consider the following example.

Example: Create a list using the constructor of the list class

- Create an empty list.
`L1 = list();`
- Create a list with any three integer elements, such as 10, 20 and 30.
`L2 = list([10,20,30])`
- Create a list with three string elements, such as "Apple", "Banana" and "Grapes".
`L3 = list(["Apple","Banana","Grapes"])`
- Create a list using inbuilt `range()` function.
`L4 = list(range(0,6))` # create a list with elements from 0 to 5
- Create a list with inbuilt characters X, Y and Z.
`L5=list("xyz")`

Example: Creating a list without using the constructor of the list class

- Create a list with any three integer elements, such as 10, 20 and 30.
`L1=[10,20,30]`
- Create a list with three string elements, such as "Apple", "Banana" and "Grapes".
`L2 = ["Apple", "Banana", "Grapes"]`

Q2. Describe the term negative list indices.

Ans.

The negative index accesses the elements from the end of a list counting in backward direction. The index of the last element of any non-empty list is always -1, as shown in Figure

List1 =	10	20	30	40	50	60
	-6	-5	-4	-3	-2	-1

Figure 8.2 List with negative index

Accessing the elements of a list using a negative index.

Example

```
>>> List1=[10,20,30,40,50,60]#Create a List
>>> List1[-1]          #Access Last element of a List
60
>>>List1[-2]           #Access the second last element of
List
50
>>> List1[-3]          #Access the Third last element of
List
40
>>>List1[-6]           #Access the first Element of the
List
10
```

Q3. Explain list slicing in details with example.

Ans.

The slicing operator returns a subset of a list called slice by specifying two indices, i.e. start and end. The syntax is:

Name_of_Variable_of_a_List[Start_Index: End_Index]

Example

```
>>> L1=[10,20,30,40,50] #Create a List with 5 Different Elements
>>> L1[1:4]
20,30,40
```

The L1[1:4] returns the subset of the list starting from index the start index 1 to one index less than that of the end index, i.e. 4-1 = 3.

Example

```
>>> L1=[10,20,30,40,50]) #Create a List with 5 Different Elements
>>> L1[2:5]
[30, 40, 50]
```

The above example L1 creates a list of five elements. The index operator L1[2:5] returns all the elements stored between the index 2 and one less than the end index, i.e. 5-1 = 4.

Q4. Explain list slicing with step with examples.

Ans.

LIST SLICING WITH STEP SIZE

In slicing, the first two parameters are start index and end index. Thus, we need to add a third parameter as step size to select a list with step size. To be able to do this we use the syntax:

`List_Name[Start_Index:End_Index:Step_Size]`

Example

```
>>> MyList1=["Hello",1,"Monkey",2,"Dog",3,"Donkey"]
>>> New_List1=MyList1[0
:6:2]
print(New_List1)

['Hello', 'Monkey', 'Dog']
```

Explanation Initially we created a list named Mylist1 with five elements. The statement MyList1[0:6:2] indicates the programmer to select the portion of a list which starts at index 0 and ends at index 6 with the step size as 2. It means we first extract a section or slice of the list which starts at the index 0 and ends at the index 6 and then selects every other second element.

Example

```
>>> List1=["Python",450,"C",300,"C++",670]
>>> List1[0:6:3]    #Start from Zero and Select every
Third Element
['Python', 300]    #Output
```

Q5. Explain Python inbuilt functions for list along with the example.

Ans.

Python has various inbuilt functions that can be used with lists. Some of these are listed in Table 8.1.

Table 8.1 Inbuilt functions that can be used with lists

<i>Inbuilt Functions</i>	<i>Meaning</i>
<code>Len()</code>	Returns the number of elements in a list.
<code>Max()</code>	Returns the element with the greatest value.
<code>Min()</code>	Returns the element with the lowest value.
<code>Sum()</code>	Returns the sum of all the elements.
<code>random.shuffle()</code>	Shuffles the elements randomly.

Example

#Creates a List to store the names of Colors and return size of list.

```
>>> List1=["Red","Orange","Pink","Green"]
>>> List1
['Red', 'Orange', 'Pink', 'Green']
>>> len(List1)          #Returns the Size of List
4
```

#Create a List, find the Greatest and Minimum value from the list.

```
>>> List2=[10,20,30,50,60]
>>> List2
[10, 20, 30, 50, 60]
>>> max(List2)          #Returns the greatest element from the
list.
60
>>> min(List2)          #Returns the minimum element from the list.
10
```

#Create a List, and Shuffle the elements in random manner. #Test Case 1

```
>>>import random
>>> random.shuffle(List2)
>>> List2
```

```
[30, 10, 20, 50, 60]
>>> List2
[30, 10, 20, 50, 60]

#Test Case2
>>> random.shuffle(List2)
>>> List2
[20, 10, 30, 50, 60]
#Create a List, and find the sum of all the elements of a
List.
>>> List2=[10,20,30,50,60]
>>> List2
[10, 20, 30, 50, 60]
>>> sum(List2)
170
```

Q6. Explain list operators in details.

Ans.

1. The + Operator: The concatenation operator is used to join two lists.

Example

```
>>> a=[1,2,3]                #Create a list with three elements
1,2, and 3
>>> a                        #Prints the list
[1, 2, 3]
>>> b=[4,5,6]                #Create a list with three elements
4,5, and 6
>>> b                        #print the list
[4, 5, 6]
>>> a+b
                                #Concatenat
e the list a and b[1, 2, 3, 4, 5, 6]
```

2. The * Operator: The multiplication operator is used to replicate the elements of a list.

Example

```
>>> List1=[10,20]
>>> List1
[10, 20]
>>> List2=[20,30]
>>> List2
[20, 30]
>>> List3=2*List1      #Print each element of a List1 twice.
>>> List3
[10, 20, 10, 20]
```

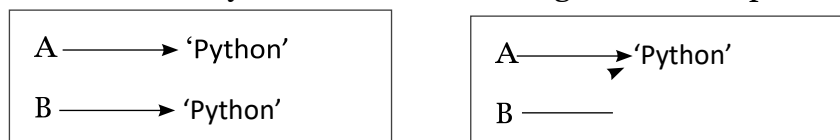
3. The in Operator: The in operator used to determine whether an element is in a list. It returns True if the element is present and False if the element is absent in the list.

Example

```
>>> List1= [10,20]
>>> List1
[10, 20]
>>> 40 in List1      #To Check if 40 is present in List1
False
>>> 10 in List1      #To Check if 10 is present in List1
True
```

4. **The isOperator:** Let us execute the following two statements:
A='Python'
B='Python'

We know that both A and B refer to a string but we don't know whether they refer to the same string or not. Two possible situations



are:

In the first case, A and B refer to two different objects that have the same values. In second case, they refer to the same object. To understand whether two variables refer to the same object, a programmer can use the 'is' operator.

Example

```
>>> A='Microsoft'
>>> B='Microsoft'
>>> A is B #Check if two variable refer to the same Object
True
```

From the above example, it is clear that Python created only one string object and both A and B refer to the same object. However, when we

create two lists with the same elements, Python creates two different objects as well.

Example

```
>>> A=['A','B','C']
>>> B=['A','B','C']
>>> A is B #Check if two lists refer to the same Object
False
```

Explanation: In the above example, the two lists A and B contain exactly the same number of elements. The is operator is used to check if both the variables A and B refer to the same object, but it returns False. It means that even if the two lists are the same, Python creates two different objects. State diagram for the above example is given in Figure 8.3.

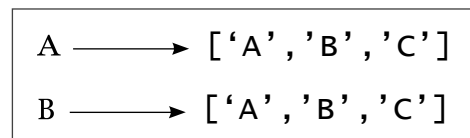


Figure 8.3 Effect of is operator on a list

It is important to note that in the above example, we can say that the two lists are equivalent because they have the same elements. We cannot say that both the lists are identical because they don't refer to the same object.

5.The del Operator: The del operator stands for Delete. The del operator is used to remove the elements from a list. To delete the element of a list, the elements of the list are accessed using their index position and the del operator is placed before them.

Example

```

Lst=[10,20,30,40,50,60,70]
>>> del Lst[2]      #Removes 3rd element from the List
>>> Lst
[10, 20, 40, 50, 60, 70]

Lst=[10,20,30,40,50,60,70]
>>> del Lst[-1]
>>> Lst              #Removes last element from the List
[10, 20, 30, 40, 50, 60]

>>> Lst=[10,20,30,40,50,60,70]
>>> del Lst[2:5]     #Removes element from index position 2 to 4
>>> Lst
[10, 20, 60, 70]

>>> Lst=[10,20,30,40,50,60,70]
>>> del Lst[:]       #Removes all the element from the List
>>> Lst
[]

```

Q7. Explain List comprehension. Write down a program to display even elements of the list using list comprehension where the range of elements is 1 to 30.

Ans.

List comprehension is used to create a new list from existing sequences. It is a tool for transforming a given list into another list.

Example: Without list comprehension

Create a list to store five different numbers such as 10, 20, 30, 40 and 50. Using the for loop, add number 5 to the existing elements of the list.

```

>>> List1= [10, 20, 30, 40, 50]
>>> List1
[10, 20, 30, 40, 50]
>>> for i in range(0,len(List1)):
    List1[i]=List1[i]+5      #Add 5 to each element of List1
>>> List1                    #print the List1 After Performing
[15, 25, 35, 45, 55]

```

The above code is workable but not the optimal code or the best way to write a code in Python. Using list comprehension, we can replace the loop with

a single expression that produces the same result.

The syntax of list comprehension is based on set builder notation in mathematics. Set builder notation is a mathematical notation for describing a set by stating the property that its members should satisfy. The syntax is

[<expression> for <element> in <sequence> if <conditional>]

The syntax is designed to read as “Compute the expression for each element in the sequence, if the conditional is true”.

Example: Using list comprehension

```
>>> List1= [10, 20, 30, 40, 50]
>>> List1
[10, 20, 30, 40, 50]

>>>for i in range(0,len(List1)):
    List1[i]=List1[i]+10
```

Without List Comprehension

```
>>> List1= [10,20,30,40,50]
>>> List1= [x+10 for x in List1]
>>> List1
[20, 30, 40, 50, 60]
```

Using List Comprehension

In the above example, the output for both without list comprehension and using list comprehension is the same. The use of list comprehension requires lesser code and also runs faster. With reference to the above example we can say that list comprehension contains:

- An input sequence
- A variable referencing the input sequence
- An optional expression
- An output expression or output variable

Example

```
List1= [20, 30, 40, 50, 60]
List1= [ x+10 for x in List1]
```

(An output variable)

(A variable referencing an input sequence)

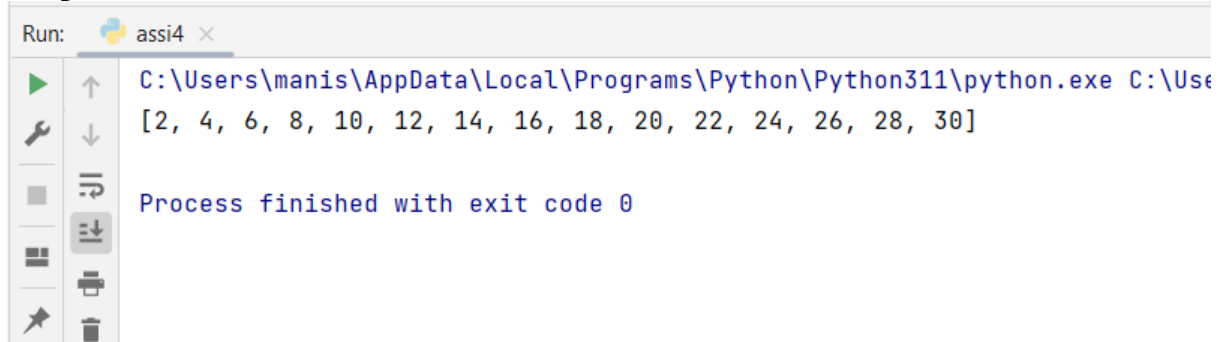
(An input sequence)

Output [20, 30, 40, 50, 60]

Program:

```
even_nums = [num for num in range(1,31) if num % 2 == 0]
print(even_nums)
```

Output:



```
Run: assi4 x
C:\Users\manis\AppData\Local\Programs\Python\Python311\python.exe C:\Use
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]

Process finished with exit code 0
```

Q8. Explain List methods in details with example.

Ans.

Once a list is created, we can use the methods of the list class to manipulate the list. Table 8.2 contains the methods of the list class along with examples.

Methods of List	Meaning
<p>None append(object x) end of</p> <p>Example: return type</p> <pre>>>> List1=['X','Y','Z'] >>> List1 ['X', 'Y', 'Z'] >>> List1.append('A') #Append element 'A' to the endof the List1 >>> List1 ['X', 'Y', 'Z', 'A']</pre> <p>Note: Append method is equivalent to doing: List1[len(List1):]=[Eleme nt_Name]</p> <p>Example:</p> <pre>>>>List1=["Red","Blue","Pink"] >>> List1 ['Red', 'Blue', 'Pink'] >>> List1[len(List1):]=['Yellow'] >>> List1 ['Red', 'Blue', 'Pink', 'Yellow']</pre>	<p>Adds an element x to the the list. None is the of method appended.</p>

<p>None <u>clear()</u> Example: >>> List1=["Red", "Blue", "Pink"] >>> List1 ['Red', 'Blue', 'Pink'] >>> List1.clear() # Removes all the element of <u>List</u> >>> List1 # Returns Empty List after removing all <u>elements</u> []</p>	<p>Removes all the items from the list.</p>
<p>int <u>count(object x)</u> Example: >>> List1=['A','B','C','A','B','Z'] >>> List1 ['A', 'B', 'C', 'A', 'B', 'Z'] #Count the number of times the element 'A' has appeared in the list >>> List1.count('A') 2 # Thus, 'A' has appeared 2 times in List1</p>	<p>Returns the number of times the element x appears in the list.</p>
<p>List <u>copy()</u> Example: >>> List1=["Red", "Blue", "Pink"] >>> List1 ['Red', 'Blue', 'Pink'] >>> List2=List1.copy() # Copy the contents of List1 to List2 >>> List2 ['Red', 'Blue', 'Pink'] Note: <u>Copy()</u> Method is equivalent to doing List2=List1[:] # Copies the content of List1 to List2 Example: >>> List1=["Red", "Blue", "Pink"] >>> List2=List1[:] >>> List2 ['Red', 'Blue', 'Pink']</p>	<p>This method returns a shallow copy of the list.</p>
<p>None <u>extend(list L2)</u> Example: >>> List1= [1,2,3] >>> List2= [4,5,6] >>> List1 [1, 2, 3]</p>	<p>Appends all the elements of list L2 to the list.</p>

```
>>> List2
[4, 5, 6]
>>> List1.extend(List2) #Appends all the elements
of List2 to List1
>>> List1
[1, 2, 3, 4, 5, 6]
```

int index(object x)

Example:

```
>>> List1=['A','B','C','B','D','A']
```

```
>>> List1
```

```
['A', 'B', 'C', 'B', 'D', 'A']
```

#Returns the index of first occurrence of element
'B' from the list1

```
>>> List1.index('B')
```

```
1 #Returns the index of element B
```

None insert(int index, Object X)

Example:

```
>>> Lis1=[10,20,30,40,60]
```

```
>>> Lis1
```

```
[10, 20, 30, 40, 60]
```

```
>>> Lis1.insert(4,50) #Insert Element 50 at index 4
```

```
>>> Lis1
```

```
[10, 20, 30, 40, 50, 60]
```

Returns the index of the first occurrence of the element *x* from the list.

Insert the element at a given index.

Note: The index of the first element of a list is always zero.

Object pop(i)

Example:

```
>>> Lis1=[10,20,30,40,60]
```

```
>>> Lis1
```

```
[10, 20, 30, 40, 60]
```

```
>>> Lis1.pop(1) # Remove the element which is at
index 1.
```

```
20
```

```
>>> Lis1 # Display List after removing the
element from index 1.
```

```
[10, 30, 40, 60]
```

```
>>> Lis1.pop() # Remove the last element from the
list
```

```
60
```

```
>>> Lis1
```

```
[10, 30, 40] #Display the list after removing last
element
```

Removes the element from the given position. Also, it returns the removed element.

Note: The parameter *i* is optional. If it is not specified then it removes the last element from the list.

None <u>remove</u> (object x) Example: <pre>>>> List1=['A','B','C','B','D','E'] >>> List1 ['A', 'B', 'C', 'B', 'D', 'E'] >>> List1.remove('B')#Removes the first occurrence of element B >>> List1 ['A', 'C', 'B', 'D', 'E']</pre>	Removes the first occurrence of element x from the list.
None <u>reverse</u> () Example: <pre>>>> List1=['A','B','C','B','D','E'] >>> List1 ['A', 'B', 'C', 'B', 'D', 'E'] >>> List1.reverse() # Reverse all the elements of the list. >>> List1 ['E', 'D', 'B', 'C', 'B', 'A']</pre>	Reverses the element of the list.
None <u>sort</u> () Example: <pre>>>> List1=['G','F','A','C','B'] >>> List1 ['G', 'F', 'A', 'C', 'B'] #Unsorted List >>> List1.sort() >>> List1 ['A', 'B', 'C', 'F', 'G'] #Sorted List</pre>	Sort the elements of list.

Q9. Explain following in short:

a. Splitting a string in a list:

Ans.

The `list()` function breaks a string into individual letters. In this section, we will explore how to split a string into words.

The `str` class contains the `split` method and is used to split a string into words.

Example

```
>>> A="Wow!!! I Love Python Programming" #A Complete String
>>> B=A.split() # Split a String into Words
>>> B #Print the contents of B
['Wow!!!', 'I', 'Love', 'Python', 'Programming']
```

Explanation In the above example, we have initialised string to A as "Wow!!! I Love Python Programming". In the next line, the statement, `B = A.split()` is used to split "Wow!!! I Love Python Programming" into the list `['Wow!!!', 'I', 'Love', 'Python', 'Programming']`.

It is fine to split a string without a delimiter. But what if the string contains

the delimiter? A string containing a delimiter can be split into words by removing the delimiter. It is also possible to remove the delimiter from the string and convert the entire string into a list of words. In order to remove the delimiter, the `split()` method has a parameter called `split(delimiter)`. The parameter `delimiter` specifies the character to be removed from the string. The following example illustrates the use of a delimiter inside the `split()` method.

Example

```
>>> P="My-Data-of-Birth-03-June-1991" # String with Delimiter
'\ '

>>> P                                     # Print the Entire String
'My-Data-of-Birth-03-June-1991'

>>> P.split('-')                         #Remove the delimiter '-' using
split method.

['My', 'Data', 'of', 'Birth', '03', 'June', '1991']
```

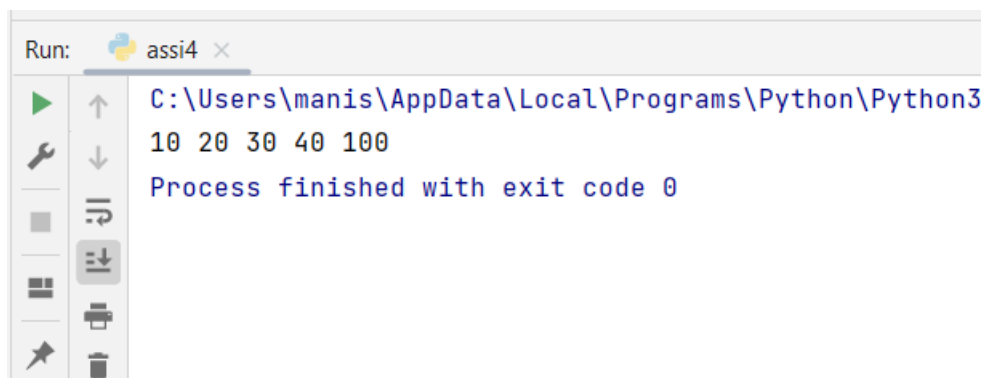
b. Passing list to a function:

Ans.

As list is a mutable object. A programmer can pass a list to a function and can perform various operations on it. We can change the contents of a list after passing it to a function. Since a list is an object, passing a list to a function is just like passing an object to a function.

Consider the following example to print the contents of a list after the list is passed to a function.

```
def print_list(lst):
    for num in lst:
        print(num, end=" ")
lst = [10,20,30,40,100]
print_list(lst)
```



c. Returning list from a function.**Ans.**

We can pass a list while invoking a function. Similarly, a function can return a list. When a function returns a list, the list's reference value is returned.

Consider the following example to pass a list to a function. After passing, reverse the elements of the list and return the list.

```
def Reverse_List(Lst):
    print('List Before Reversing = ',Lst)
    Lst.reverse()    # The reverse() to reverse the contents of list
    return Lst       # Return List
Lst=[10,20,30,40,3]
print('List after Reversing = ',Reverse_List(Lst))
```

Output

```
List Before Reversing = [10, 20, 30, 40, 3]
List after Reversing = [3, 40, 30, 20, 10]
```

Q10. Define Searching techniques.**Ans.**

Searching is a technique of quickly finding a specific item from a given list of items, such as a number in a telephone directory. Each record has one or more fields such as name, address and number. Among the existing fields, the field used to distinguish records is called the key. Imagine a situation where you are trying to find the number of a friend. If you try to locate the record corresponding to the name of your friend, the 'name' will act as the key. Sometimes there can be various mobile numbers allotted to the same name. Therefore, the selection of the key plays a major role in defining the search algorithm. If the key is unique, the records are also identified uniquely. For example, mobile number can be used as the key to search for the mobile number of a specific person. If the search results in locating the desired record then the search is said to be successful, else the search is said to be unsuccessful. Depending on way the information is stored for searching a particular record, search techniques are categorised as:

- (a) Linear or sequential search
- (b) Binary search

Q11. Explain Linear Search in details.**Ans.**

In linear search, elements are examined sequentially starting from the first element. Element to be searched, i.e. the (key element), is compared

sequentially with each element in a list. The search process terminates when the element to be searched, i.e. the key element, matches with the element present in the list or the list is exhausted without a match being found in it. If the element to be searched is found within the list, the linear search returns the index of the matching element. If the element is not found, the search returns -1.

```
def Linear_Search(My_List,key):
    for i in range(len(My_List)):
        if(My_List[i]==key):
            #print(key,"is found at index",i)
            return i
            break
    return -1
My_List=[12,23,45,67,89]
```

```
print("Contents of List are as follows:")
print(My_List)
key=(int(input("Enter the number to be searched:")))
L1=Linear_Search(My_List,key)
if(L1!=-1):
    print(key ," is found at position",L1+1)
else:
    print(key," is not present in the List")
```

Output

```
#Test Case 1
Contents of List are as follows:
[12, 23, 45, 67, 89]
Enter the number to be searched:23
23 is found at position 2
#Test Case 2
Contents of List are as follows:
[12, 23, 45, 67, 89]
Enter the number to be searched:65
65 is not present in the List
```

Explanation In the above program, we have defined the function called `Linear _ Search()`. The list and element to be searched, i.e. the key, is passed to the function. The comparison starts from the first element of the list. The comparison goes on sequentially till the key element matches the element present within the list or the list is exhausted without a match being found.

Unordered List—Analysis of Sequential Search

Table 9.1 shows that the analysis has been made with respect to the unordered list, i.e. if the content of the list is not in any order, either ascending or descending.

Table 9.1 Sequential search analysis in an unordered list

Case	Best Case	Worst Case	Average Case
Element is present in the list	1	N	$N/2$
Element is not present in the list	N	N	$N/2$

Sorted List—Analysis of Sequential Search

Expected number of comparisons required for an unsuccessful search can be reduced if the list is sorted.

Example

```
List1[] = 10 15 20 25 50 60 70 80
           ↑
Element to be searched = 30
           |
           Search should terminate here.
```

Assuming the elements are stored in an ascending order, the search should terminate as soon as the value of the element in the list is greater than value of the element (key) to be searched or the key (element to be searched) is found (Table 9.2).

Table 9.2 Sequential search analysis on a sorted list

	Best Case	Worst Case	Average Case
Element is present in the list	1	N	$N/2$
Element is not present in the list	N	N	$N/2$

Q12. Explain Binary search in details.

Ans.

Linear search is convenient for a small list. What happens if the size of a list is large? Let us consider the size of the list is 1 Million (2^{20}). So, if we want to search using a sequential search algorithm then in the worst case we require 2^{20} comparisons. This means that a sequential search algorithm is not suitable for a large list. Thus, we require more efficient algorithms. In this section, we will explore how binary search is a simple and efficient algorithm.

For binary search, the elements in a list must be in a sorted order. Let us consider the list is in ascending order. The binary search compares the element to be searched, i.e. the key element with the element in the middle of the list. The binary search algorithm is based on the following three conditions:

1. If the key is less than the list's middle element then a programmer has to search only in the first half of the list.

2. If the key is greater than the list's middle element then a programmer has to search only in the second half of the list.
3. If the element to be found, i.e. the key element is equal to the middle element in the list then the search ends.
4. If the element to be found is not present within the list then it returns None or -1 which indicates the element to be searched is not present in the list.

Example of Binary Search

Consider the sorted list of 10 integers given below.

10 18 19 20 25 28 48 55 62 70

Element to be searched = 48

Iteration 1

Index	0	1	2	3	4	5	6	7	8	9
Element of List	10	18	19	20	25	28	48	55	62	70
	↑				↑					↑
	Low=0				Mid=4					High=9

$$\begin{aligned} \text{Mid} &= (\text{Low} + \text{High})/2 \\ &= (0 + 9)/2 \\ &= 4 \end{aligned}$$

Now we will compare the middle element which is 25 with the element that we want to search, i.e. 48.

Since $48 > 25$,

we will eliminate the first half of the list and we will search again in the second half of the list. Now,

Low = Mid + 1 = 4 + 1 = 5 #Change the Position of Low

High = 9 # High will remain as earlier.

Iteration 2

Index	0	1	2	3	4	5	6	7	8	9
Element of List	10	18	19	20	25	28	48	55	62	70
						↑		↑		↑
						Low=5		Mid=7		High=9

$$\begin{aligned}
 \text{Mid} &= (\text{Low} + \text{High}) / 2 \\
 &= (5 + 9) / 2 \\
 &= 7
 \end{aligned}$$

Now we will compare the middle element which is 55 with element we want to search, i.e. 48.

Since $48 < 55$,

we will search the element in the left half of the list.

Now,

Low = 5 #It will remain as it is.

High = Mid-1 = 7-1 #Change the Position of High
= 6

Iteration 3

Index	0	1	2	3	4	5	6	7	8	9
Element of List	10	18	19	20	25	28	48	55	62	70
						↑	↑			
						Low=5	High=6			
						Mid=5				

$$\begin{aligned}
 \text{Mid} &= (\text{Low} + \text{High}) / 2 \\
 &= (5 + 6) / 2 \\
 &= 5
 \end{aligned}$$

Now we will compare the middle element which is 28 with the element we want to search, i.e. 48.

Since $28 < 48$,

we will search the element in the right portion of the mid of the list.

Now,

Low = mid+1 = 6 #Change the Position of Low

High = 6 #High will remain as it is.

Iteration 4

Index	0	1	2	3	4	5	6	7	8	9
Element of List	10	18	19	20	25	28	48	55	62	70



Low=5 High= Mid=6

$$\text{Mid} = (\text{Low} + \text{High})/2$$

$$= 6$$

Now we will compare the middle element which is 48 with the element we want to search, i.e. 48.

Since 48=48, the number is found at index position 6.

Q13. Describe Sorting and types of sorting.

Ans.

Consider a situation where a user wants to pick up a book from a library but finds that the books are stacked in no particular order. In this situation, it will be very difficult to find any book quickly or easily. However, if the books were placed in some order, say alphabetically, then a desired title could be found with little effort. As in this case, sorting is used in various applications in general to retrieve information efficiently.

Sorting means rearranging the elements of a list, so that they are organised in some relevant order. The order can be either ascending or descending. Consider a list L1, in which the elements are arranged in an ascending order in a way that $L1[0] < L1[1] < \dots < L1[N]$.

Example

If a list is declared and initialised as:

$L1 = [9, 3, 4, 2, 1]$

The sorted list in an ascending

order can be: $L1 = [1, 2, 3, 4, 9]$

From the above example, it is clear that sorting is a process of converting an unordered set of elements into an ordered set.

Types of Sorting

Sorting algorithms are divided into two main categories, viz.

1. Internal sorting
2. External sorting

If all the records to be sorted are kept internally in the main memory then they can be sorted using internal sort. However, if a large number of records are to be sorted and kept in secondary storage then they have to be sorted

using external sort.

1. **Internal sorting algorithms:** Any sorting algorithm which uses the main memory exclusively during sorting is called an internal sort algorithm. It takes advantage of the random access nature of the main memory. Internal sorting is faster than external sorting.
2. **External sorting algorithms:** External sorting is carried on secondary storage. Therefore, any sorting algorithm which uses external memory, such as tape or disk during sorting is called external sort algorithm. It is carried out if the number of elements to be stored is too large to fit in the main memory. Transfer of data between secondary and main memory is best done by moving blocks of contiguous elements.

The various sorting algorithms are Bubble sort, Selection sort, Insertion sort, Quick sort and Merge sort.

Q14. Explain Bubble sort in details.

Ans.

Bubble sort is the simplest and oldest sorting algorithm. Bubble sort sorts a list of elements by repeatedly moving the largest element to the highest index position of the list. The consecutive adjacent pair of elements in both the lists is compared with each other. If the element at lower index is greater than the element at higher index then the two elements are interchanged so that the element with the smaller value is placed before the one with a higher value. The algorithm repeats this process till the list of unsorted elements is exhausted. This entire procedure of sorting is called bubble sort. The algorithm derives its name as bubble sort because the smaller elements bubble to the top of the list.

Example of Bubble Sort

Consider the elements
within a list as:

$L1 = [30, 50, 45, 20, 90, 78]$

Sort the list using bubble sort.

Solution**Iteration 1**

30	40	45	20	90	78	No Exchange
30	40	45	20	90	78	No Exchange
30	40	45	20	90	78	Exchange
30	40	20	45	90	78	No Exchange
30	40	20	45	90	78	Exchange
30	40	20	45	78	90	Output of bubble sort after the first iteration. But still the output is not in a sorted order. Repeat the above steps for iteration 2.

Iteration 2

Apply bubble sort to the output of the first iteration.

30	40	20	45	78	90	No Exchange
30	40	20	45	78	90	Exchange
30	20	40	45	78	90	No Exchange
30	20	40	45	78	90	No Exchange
30	20	40	45	78	90	Output of bubble sort after the second iteration. But still the output is not in a sorted order. Repeat the above steps for iteration 3.

Iteration 3

Apply bubble sort to the output of the second iteration.

30	20	40	45	78	90	Exchange
20	30	40	45	78	90	No Exchange
20	30	40	45	78	90	No Exchange
20	30	40	45	78	90	No Exchange
20	30	40	45	78	90	No Exchange

Thus, in third iteration itself we have obtained a sorted list of elements in an ascending order.

Working of Bubble Sort

In the above example, the working of bubble sort can be generalised as:

1. In each iteration, the first element of the list, i.e. $L[1]$ is compared with the second element of the list, i.e. $L[2]$ then $L[2]$ is compared with $L[3]$, $L[3]$ is compared with $L[4]$ and so on. Finally, $L[N-1]$ is compared with $L[N]$. This process is continued till we obtain the list in a sorted order.

2. In the second iteration, $L[1]$ is compared with $L[2]$, $L[2]$ is compared with $L[3]$ and so on. Finally, $L[N-2]$ is compared with $L[N-1]$. The iteration 2 just requires $N-2$ comparisons. Therefore, at the end of the second iteration, the second biggest element is placed at the second highest index position of the list.
3. Similarly, the above process is continued for the subsequent iterations. Therefore, in the last iteration we obtain all the elements within the list in a sorted order.

Q15. Explain Insertion Sort in details.

Ans.

Insertion sort is based on the principle of inserting an element in its correct place in a previously sorted list. It always maintains a sorted sublist in the lower portion of the list. Each new element is inserted back into the previous sub list. Thus, insertion sort sorts a list of elements repeatedly by inserting a new element into a sorted sublist until the whole list is sorted. An example of insertion sort is given below.

Example

Consider the unsorted list as

MyList = [15,0,11,19,12,16,14]

Initially, the sorted sublist contains the first element in the list, i.e. 15

15	0	11	19	12	16	14
----	---	----	----	----	----	----

- ⊙ **STEP 1:** Initially, the sorted sublist contains the first element in the list, i.e. 15. Now insert the next element from the list, i.e. 0 into the sublist.

0	15	11	19	12	16	14
---	----	----	----	----	----	----

- ⊙ **STEP 2:** The sorted sublist is [0, 15]. Insert 11 into the sublist.

0	11	15	19	12	16	14
---	----	----	----	----	----	----

- ⊙ **STEP 3:** The sorted sublist is [0,11,15]. Insert 19 into the sublist.

0	11	15	19	12	16	14
---	----	----	----	----	----	----

- ⊙ **STEP 4:** The sorted sublist is [0,11,15,19]. Insert 12 into the sublist.

0	11	12	15	19	16	14
---	----	----	----	----	----	----

- ⊙ **STEP 5:** The sorted sublist is [0,11,12,15,19]. Insert 16 into the sublist.

0	11	12	15	16	19	14
---	----	----	----	----	----	----

- ⊙ **STEP 6:** The sorted sublist is [0,11,12,15,16,19]. Insert 14, into the sublist.

0	11	12	15	16	19	14
---	----	----	----	----	----	----

- ⊙ **STEP 7:** The sorted sublist is [0,11,12,14,16,19].

0	11	12	14	15	16	19
---	----	----	----	----	----	----

Finally, we obtain the sorted list of elements in Step 7.

Q16. Explain Quick Sort in details.**Ans.**

Quick sort is one of the fastest internal sorting algorithms. It is based on the following three main strategies:

1. Split or Partition: Select a random element called pivot from the sequence of elements to be sorted. Suppose the selected element is X, where X is any number. Now split (divide) the list into the two small lists, viz. Y and Z such that:
 - All the elements of the first part Y are less than the selected element pivot.
 - All the elements of the second part Z are greater than the selected element pivot.
2. Sort the sub-arrays.
3. Merge (join/concatenate) the sorted sub-array.

The split divides the lists into two smaller sublists. When these sublists are ultimately sorted recursively using quick sort these sublists are called conquered. Therefore, the quick sort algorithm is also so called the divide and conquer algorithm.

Suppose there are N elements as $a[0]$, $a[1]$, $a[2]$,..... $a[N-1]$. The steps for using the quick sort algorithm are given below.

- ⊙ **STEP 1:** Select any element as the pivot. For example, select the element stored at the first position in a list as the pivot element. Although there are many ways to choose the pivot element, we will use the first item from the list. It helps to split a list into twoparts.

Pivot = $a[\text{First}]$ //Select Pivot Element

where the value of First is 0.

- ⊙ **STEP 2:** Initialize the two pointers i and j.

$i = \text{First} + 1$ (The first (low) index of a list)

$j = \text{Last}$ (The last (upper) index of a list)

- ⊙ **STEP 3:** Now increase the value of i until we locate an element that is greater than the pivot element.

while $i \leq j$ and $a[i] \leq \text{Pivot}$

$i++$

- ⊙ **STEP 4:** Decrease the value of j until we find a value less than the pivot element.

while $i \leq j$ and $a[j] \geq \text{Pivot}$

$j--$

- ⊙ **STEP 5:** If $i < j$ interchange $a[i]$ and $a[j]$.
- ⊙ **STEP 6:** Repeat Steps 2 to 4 until $i > j$.
- ⊙ **STEP 7:** Interchange the selected data element pivot and $a[j]$.

Q17. Explain Merger sort in details.**Ans.**

All sorting algorithms are mainly used for internal sorting where the data to be sorted fits in the main memory. When the data to be sorted resides in a file or on a disk and does not fit in the available memory, the merge sort method is used. Merge sort is a well-known and efficient method for external sorting.

Like quick sort, merge sort is also based on three main strategies:

- Split the list into two sub lists (Split or Divide): Split implies partitioning the n elements of a list into two sublists, where each sublist contains $n/2$ elements.
- Sort sublists (Conquer): Sorting two sub-arrays recursively using merge sort.
- Merge the sorted sublists (Combine): Combine implies merging two sorted sublists, each of size $n/2$, to produce a sorted list of n elements.

Example of Merge Sort

Consider the following elements within a list.

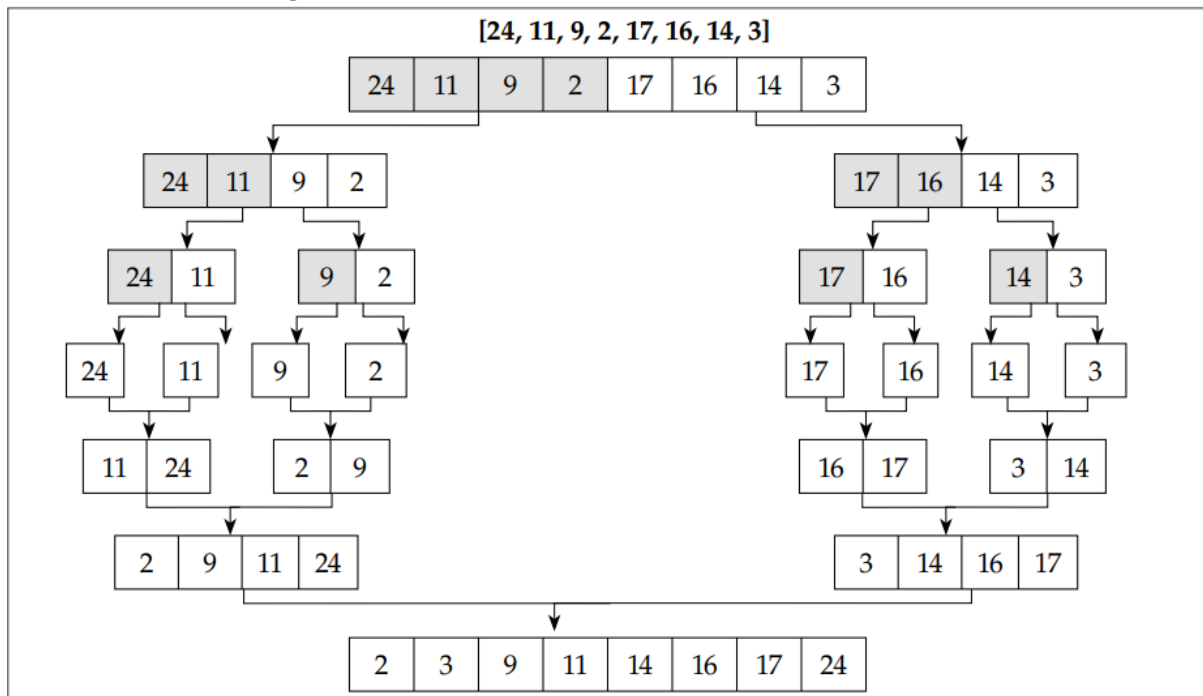


Figure 9.1 Example of merge sort

The list in Figure 9.1 has 8 elements. The index of the first element is $i=0$ and the index of the last element is $j=7$. In order to divide the above list around the middle element, the index of the middle element is found, i.e. $\text{mid}=(i+j)/2$.

Therefore, $i = 0$ and $j = 7$

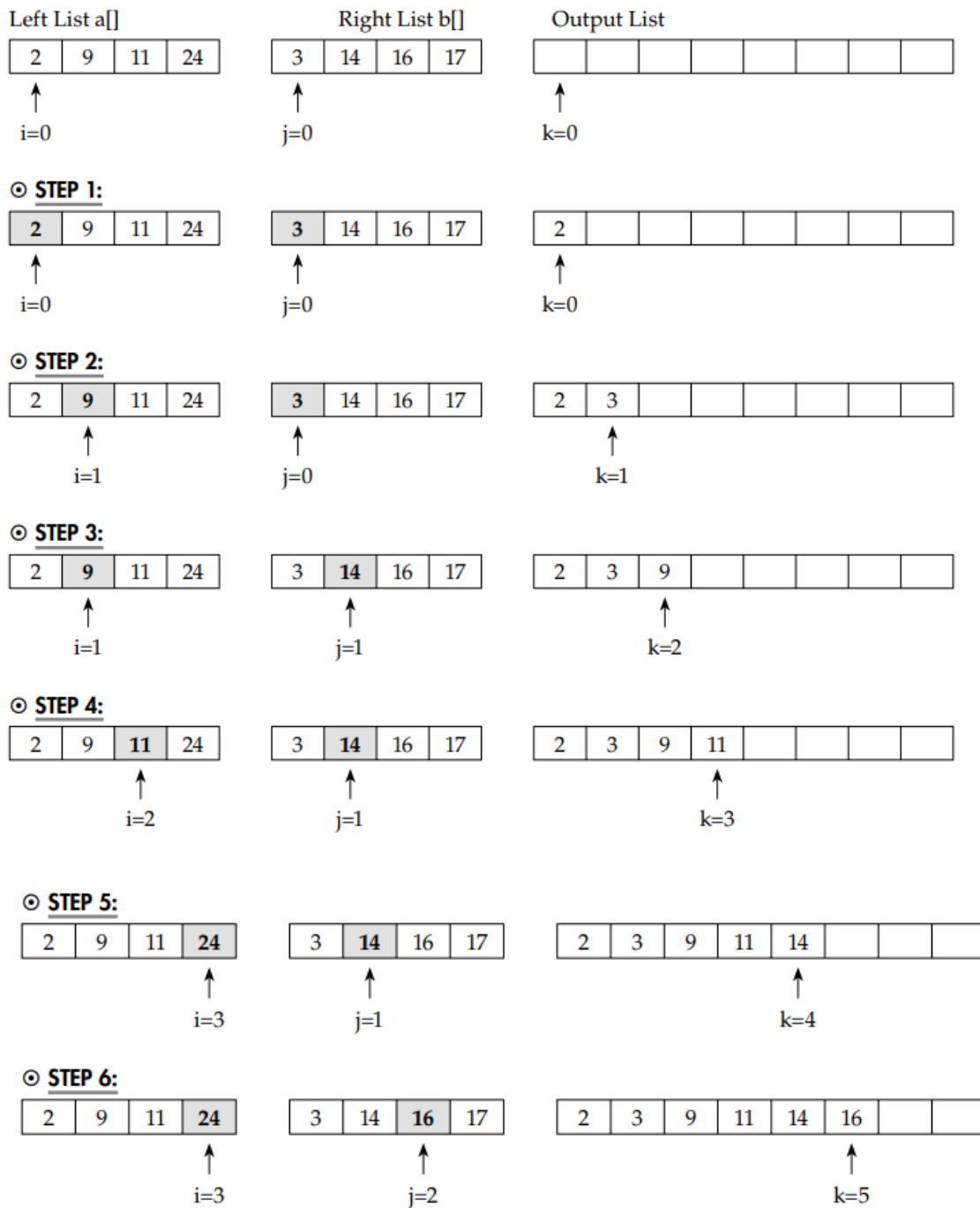
$$\text{Mid} = (i + j) / 2 = (0 + 7) / 2 = 3.$$

Merge sort is applied recursively to the left part of the list from $i = 0$ to $j = 3$. After sorting of the left half of the list, the right half of the list is sorted from $i = 4$ to $j = 7$ recursively using merge sort. After sorting the left half of the list from $i = 0$ to $j = 3$ and right half of the list from $i = 4$ to $j = 7$, the two lists are merged to produce a single sorted list.

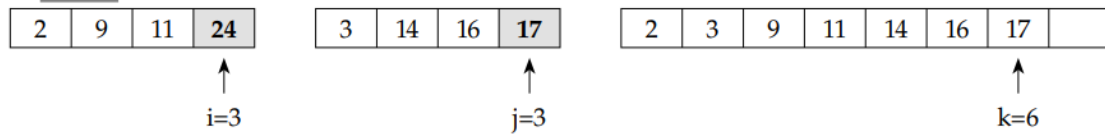
Merging Operation in Merge Sort

A fundamental operation in the merge sort algorithm is merging of two sorted lists. The merging algorithm takes two sorted lists $a[]$ and $b[]$, i.e. (left and right list) as the input and the third list $c[]$ as the output list. Each element of a list, i.e. (LeftList) $a[i]$ is compared with the elements of the (RightList) $b[j]$. The smaller element among $a[i]$ and $b[j]$ is copied to the output list $c[k]$. When either of the input list is exhausted, the remainder of the other list is copied to the output list c .

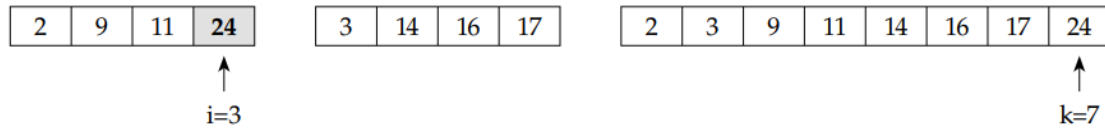
In the above example, we obtained two sorted sublists, viz. $a[]$ as the left list and $b[]$ as the rightlist, as shown.



⊙ **STEP 7:**



⊙ **STEP 8:** In Step 7, the list `b[]` is exhausted. Therefore, the remaining elements of the list `a[]` are added to the output list.



Finally, in Step 8, all the elements of the list are sorted.