PART-1

Iterators and Recursion : Recursive Fibonacci, Tower of Hanoi.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.1. What do you mean by iterator ?

Answer

1. An iterator is an object that contains a countable number of values.
2. An iterator is an object that can be iterated upon, meaning that we can traverse through all the values.
3. Python iterator, implicitly implemented in constructs like for-loops, comprehensions, and python generators.
4. Python lists, tuples, dictionary and sets are all examples of in-built iterators.
5. These types are iterators because they implement following methods :
   a. **__iter__ :** This method is called on initialization of an iterator. This should return an object that has a next() method.
   b. **next() (or __next__) :** The iterator next method should return the next value for the iterable. When an iterator is used with a 'for in' loop, the for loop implicitly calls next() on the iterator object. This method should raise a StopIteration to signal the end of the iteration.

For example :

```
# An iterable user defined type
class Test:
    # Constructor
    def __init__(self, limit):
        self.limit = limit
    # Called when iteration is initialized
    def __iter__(self):
        self.x = 10
        return self
    # To move to next element.
    def next(self):
        # Store current value of x
        x = self.x
```

```
        # Stop iteration if limit is reached
        if x > self.limit:
            raise StopIteration
        # Else increment and return old value
        self.x = x + 1;
        return x
# Prints numbers from 10 to 15
for i in Test(15):
    print(i)
# Prints nothing
for i in Test(5):
    print(i)
```

Output :

10
11
12
13
14
15

Que 5.2. **Define recursion. Also, give example.**

Answer

1. In Python, recursion occurs when a function is defined by itself.
2. When a function calls itself, directly or indirectly, then it is called a recursive function and this phenomenon is known as recursion.
3. Recursion is the property how we write a function. A function which performs the same task can be written either in a recursive form or in an iterative form.
4. Recursion is the process of repeating something self-similar way.

For example :

```
def fact (n) :
    if n == 0:
        return 1
    else :
        return n * fact(n – 1)
    print(fact(0))
    print(fact(5))
```

Output :

1
120

Que 5.3. **Explain Fibonacci series using Python.**

Answer

1. Fibonacci series is a series of numbers formed by the addition of the preceding two numbers in the series.
2. It is simply the series of numbers which starts from 0 and 1 and then continued by the addition of the preceding two numbers.
3. Example of Fibonacci series: 0, 1, 1, 2, 3, 5.
4. **Python code for recursive Fibonacci :**

```
def FibRecursion(n) :
    if n <= 1 :
        return n
    else :
        return(FibRecursion(n – 1) + FibRecursion(n – 2))
nterms = int(input("Enter the term : ")) # take input from the user
if nterms < = 0: # check if the number is valid
    print ("Please enter a positive integer")
else :
    print ("Fibonacci sequence :")
    for i in range (nterms) :
        print(FibRecursion(i))
```

Output : Enter the term : 5
Fibonacci sequence :
0 1 1 2 3

1. In the given Python program, we use recursion to generate the Fibonacci sequence.
2. The function FibRecursion is called recursively until we get the output.
3. In the function, we first check if the number n is zero or one. If yes, it returns the value of n. If not, we recursively call FibRecursion with the values $n - 1$ and $n - 2$.

Que 5.4. Explain Tower of Hanoi problem in detail.

OR

Explain iterator. Write a program to demonstrate the Tower of Hanoi using function. **AKTU 2019-20, Marks 10**

Answer

Iterator : Refer Q. 5.1, Page 5–2T, Unit-5.

Tower of Hanoi problem :

1. Tower of Hanoi is a mathematical puzzle which consists of three rods and a number of disks of different sizes, which can slide onto any rod.
2. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

3. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules :
   a. Only one disk can be moved at a time.
   b. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
   c. A disk can only be moved if it is the uppermost disk on a stack.
   d. No disk may be placed on top of a smaller disk.

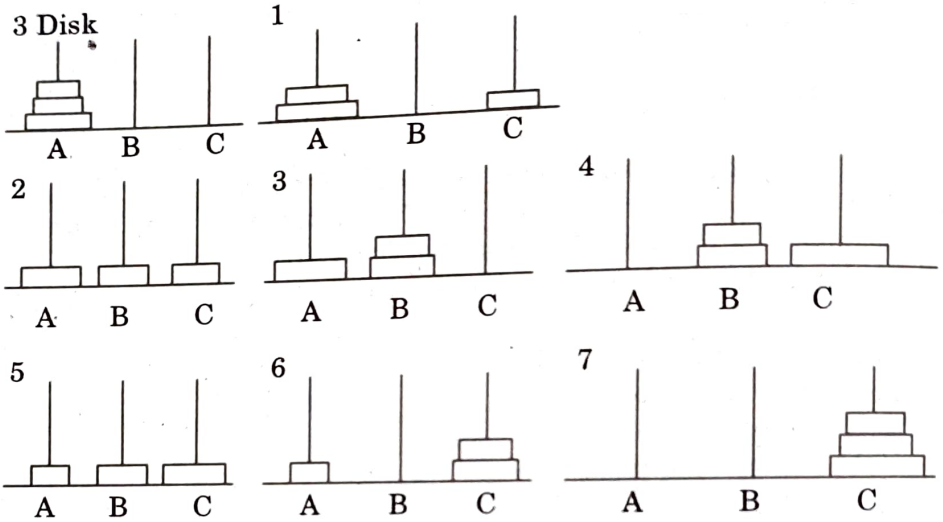

Fig. 5.4.1.

4. Recursive Python function to solve Tower of Hanoi

```
def TowerOfHanoi(n, from_rod, to_rod, aux_rod):
    if n==1:
        print "Move disk 1 from rod",from_rod,"to rod",to_rod
        return
    TowerOfHanoi(n – 1, from_rod, aux_rod, to_rod)
    print "Move disk",n,"from_rod",from_rod,"to rod",to_rod
    TowerOfHanoi(n – 1, aux_rod, to_rod, from_rod)
    n = 4
TowerOfHanoi(n, \'A\', \'C\', \'B\')
# A, C, B are the name of rods
```

Output :

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
```

Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C

Que 5.5. **Differentiate between recursion and iteration.**

OR

Discuss and differentiate iterators and recursion. Write a program for recursive Fibonacci series. **AKTU 2019-20, Marks 10**

Answer

Property	Recursion	Iteration
Definition	Function calls itself.	A set of instruction repeatedly executed.
Application	For functions.	For loops.
Termination	Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
Usage	Used when code size need to be small, and time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
Code size	Smaller code size.	Larger code size.
Time Complexity	Very high (generally exponential) time complexity.	Relatively lower time complexity (generally polynomial logarithmic).
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not use stack.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.
Speed	Slow in execution.	Fast in execution.

Program for recursive Fibonacci series : Refer Q. 5.3, Page 5–3T, Unit-5.

PART-2

Search : Simple Search and Estimating Search Time, Binary Search and Estimating Binary Search Time.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.6. What is simple (linear) search ? Explain with the help of example.

Answer

1. Linear search is a method for finding a particular value in a list.
2. Linear search is good to use when we need to find the first occurrence of an item in an unsorted collection.
3. Linear (Simple) search is one of the simplest searching algorithms, and the easiest to understand.
4. It starts searching the value from the beginning of the list and continues till the end of the list until the value is found.
5. **Code :**

```
def seach(arr, n, x) :
    i = 0
    for i in range(i, n) :
        if (arr[i] == x) :
            return – i
    return – 1
```

For example :

```
arr = [3, 10, 30, 45]
x = 10
n = len(arr)
print(x, "is present at index", search(arr, n, x))
```

Output : 10 is present at index 1

Que 5.7. Explain the time complexity for linear search.

Answer

1. Let $T(n)$ denote the time taken by a search on a sequence of size n.
2. Therefore, recurrence relation is : $T(n) = T(n-1) + C$
3. Solution to the recurrence relation is: $T(n) = Cn$
4. We have three cases to analyse an algorithm:
   - **a. Worst case :**
     - i. In the worst case analysis, we calculate upper bound on running time of an algorithm.
     - ii. For linear search, the worst case happens when the element to be searched is not present in the array.
     - iii. When x is not present, the search() functions compares it with all the elements of arr[] one by one.
     - iv. Therefore, the worst case time complexity of linear search would be $\Theta(n)$.
   - **b. Average case :**
     - i. In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.
     - ii. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So, we sum all the cases and divide the sum by $(n + 1)$.
     - iii. Therefore, the average case time complexity of linear search would be $\Theta(n)$.
   - **c. Best case :**
     - i. In the best case analysis, we calculate lower bound on running time of an algorithm.
     - ii. In the linear search problem, the best case occurs when x is present at the first location.
     - iii. So, time complexity in the best case would be $\Theta(1)$.

Que 5.8. Discuss binary search in Python.

Answer

1. Binary search follows a divide and conquer approach. It is faster than linear search but requires that the array be sorted before the algorithm is executed.
2. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned.
3. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.

4. Otherwise, the item is searched for in the sub-array to the right of the middle item.
5. This process continues on the sub-array as well until the size of the sub-array reduces to zero.
6. **Code :**

```
def binarysearch(arr, 1, r, x) :
    while 1 <= r :
        mid = 1 + (r – 1)/2;
    # Check if x is present at mid
        if arr[mid] == x:
            return mid
    # If x is greater, ignore left half
        elif arr[mid] < x :
            l = mid + 1
    # If x is smaller, ignore right half
        else :
            r = mid – 1
# If we reach here, then the element was not present
    return  – 1
# Test array
arr = [2, 3, 4, 10, 40]
x = 10
# Function call
result = binarySearch(arr, 0, len(arr) – 1, x)
if result ! = – 1:
    print "Element is present at index % d" % result
else :
    print "Element is not present in array"
```

Output :

Element is present at index 3

Que 5.9. Explain the time complexity for binary search.

Answer

1. Let $T(n)$ denote the time taken by a search on a sequence of size n.
2. Therefore, recurrence relation is : $T(n) <= T(n/2) + C$
3. Solution to the recurrence relation is : $T(n) = \log(n)$

Recursive function of binary search :

$T(n) = T(n/2) + 1$

$T(n/2) = T(n/4) + 1 + 1$

Put the value of $T(n/2)$ in above so

$T(n) = T(n/4) + 1 + 1 \ldots T(n/2^k) + 1 + 1$

$= T(2^k/2^k) + 1 + 1 \ldots + 1$ upto k

$= T(1) + k$

As we taken $2^k = n$

$k = \log n$

So time complexity is $O(\log n)$

4. We have three cases to analyse an algorithm :

   a. Best case :

   i. In the best case, the item x is the middle in the array A. A constant number of comparisons (actually just 1) are required.

   ii. Time complexity in best case is $O(1)$.

   b. Worst case :

   i. In the worst case, the item x does not exist in the array A at all. Through each recursion or iteration of binary search, the size of the admissible range is halved.

   ii. Time complexity in worst case is $O(\log n)$.

   c. Average case :

   i. To find the average case, take the sum over all elements of the product of number of comparisons required to find each element and the probability of searching for that element.

   ii. Time complexity in average case is $O(\log n)$.

PART-3

Sorting and Merging : Selection Sort, Merge List, Merge Sort, Higher Order Sort.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.10. What do you mean by selection sort ? Discuss in detail.

Answer

1. The selection sort algorithm sorts an array by repeatedly finding the smallest element (considering ascending order) from unsorted list and swapping it with the first element of the list.
2. The algorithm maintains two sub-arrays in a given array:
   i. The sub-array which is already sorted.
   ii. Remaining sub-array which is unsorted.
3. In every iteration of selection sort, the smallest element from the unsorted sub-array is picked and moved to the sorted sub-array.
4. **Code :**

```
def slectionSort(nlist) :
    for fillslot in range(len(nlist) – 1, 0, – 1) :
        maxpos = 0
        for location in range(1, fillslot + 1) :
            if nlist[location]>nlist[maxpos] :
                maxpos = location
        temp = nlist[fillslot]
        nlist[fillslot] = nlist[maxpos]
        nlist[maxpos] = temp
nlist = [14, 46, 43, 27, 57, 41, 45, 21, 70]
selectionSort(nlist)
print(nlist)
```

Output :

[14, 21, 27, 41, 43, 45, 46, 57, 70]

5. **Time complexity :**
   i. **Best case :** $O(n^2)$
   ii. **Worst case :** $O(n^2)$
   iii. **Average case :** $O(n^2)$

Que 5.11. Explain merge list.

Answer

1. Merging is defined as the process of creating a sorted list/array of data items from two other sorted array/list of data items.
2. Merge list means to merge two sorted list into one list.

Code for merging two lists and sort it :

```
a=[ ]
c=[ ]
```

```
n1=int(input("Enter number of elements:"))
for i in range(1, n1+1):
    b=int(input("Enter element:"))
    a.append(b)
n2=int(input("Enter number of elements:"))
for i in range(1, n2+1):
    d=int(input("Enter element:"))
    c.append(d)
new=a+c
new.sort( )
print("Sorted list is:", new)
```

Que 5.12. Explain merge sort with the help of example.

Answer

1. Merge sort is a divide and conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.
2. The merge() function is used for merging two halves.
3. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[$m + 1$..r] are sorted and merges the two sorted sub-arrays into one.
4. **Code :**

```
def mergeSort(arr)
    if len(arr) >1:
        mid = len(arr)//2 #Finding the mid of the array
        L = arr[:mid] # Dividing the array elements
        R = arr[mid:] # into 2 halves
        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half
        i = j = k = 0
# Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i + = 1
            else :
                arr[k] = R[j]
```

```
                j + = 1
            k + = 1
    # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i + = 1
            k + = 1
        while j < len(R):
            arr[k] = R[j]
            j + = 1
            k + = 1
    # Code to print the list
    def printList(arr):
        for i in range(len(arr)):
            print(arr[i],end=" ")
        print()
    # driver code to test the above code
    if __name__ == '__main__' :
        arr = [12, 11, 13, 5, 6, 7 ]
        print ("Given array is", end = "\n")
        printList(arr)
        mergeSort(arr)
        print("Sorted array is: ", end = "\n")
        printList(arr)
```

Output :

Given array is

12, 11, 13, 5, 6, 7

Sorted array is

5, 6, 7, 11, 12, 13

5. **Time complexity :** Recurrence relation of merge sort is given by

$T(n) = 2T(n/2) + Cn$

$= 2(2T(n/4) + Cn/2) + Cn = 2^2T(n/4) + 2Cn$

$= 2^2(2T(n/8) + Cn/4) + Cn = 2^3T(n/8) + 3Cn$

$= \ldots$ // keep going for k steps

$= 2^kT(n/2^k) + k*Cn$

Assume $n = 2^k$ for some k.

$k = \log_2 n$

Then, $T(n) = n*T(1) + Cn*\log_2 n$

i. Time complexity of Merge sort is $O(n \log n)$ in all three cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

Que 5.13. Discuss higher order sort.

Answer

1. Python also supports higher order functions, meaning that functions can accept other functions as arguments and return functions to the caller.
2. **Sorting of higher order functions :**
   a. In order to defined non-default sorting in Python, both the sorted() function and .sort() method accept a key argument.
   b. The value passed to this argument needs to be a function object that returns the sorting key for any item in the list or iterable.
3. **For example :** Consider the given list of tuples, Python will sort by default on the first value in each tuple. In order to sort on a different element from each tuple, a function can be passed that return that element.

```
>>> def second_element (t) :
...         return t[1]
...
>>> zepp = [('Guitar', 'Jimmy'), ('Vocals', 'Robert'), ('Bass', 'John Paul'), ('Drums', 'John')]
>>> sorted(zepp)
[('Bass', 'John Paul'), ('Drums', 'John'), ('Guitar', 'Jimmy'), ('Vocals', 'Robert')]
>>> sorted(zepp, key = second_element)
[('Guitar', 'Jimmy'), ('Drums', 'John'), ('Bass', 'John Paul'), ('Vocals', 'Robert')]
```

Que 5.14. Discuss sorting and merging. Explain different types of sorting with example. Write a Python program for Sieve of Eratosthenes. **AKTU 2019-20, Marks 10**

Answer

Sorting :

1. Sorting refers to arranging data in a particular order.
2. Most common orders are in numerical or lexicographical order.
3. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner.
4. Sorting is also used to represent data in more readable formats.

Merging : Refer 5.11, Page 5–11T, Unit-5.

Different types of sorting are :

1. **Bubble sort :** It is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

   For example :

```
def bubblesort(list):
# Swap the elements to arrange in order
    for iter_num in range(len(list) – 1,0, – 1):
        for idx in range(iter_num):
        if list[idx]>list[idx+1]:
                temp = list[idx]
                list[idx] = list[idx+1]
                list[idx+1] = temp
list = [19,2,31,45,6,11,121,27]
bubblesort(list)
print(list)
```

   Output :

   [2, 6, 11, 19, 27, 31, 45, 121]

2. **Merge sort :** Refer 5.12, Page 5–12T, Unit-5.
3. **Selection sort :** Refer 5.10, Page 5–10T, Unit-5.
4. **Higher order sort :** Refer 5.13, Page 5–14T, Unit-5.
5. **Insertion sort :**
   a. Insertion sort involves finding the right place for a given element in a sorted list. So in beginning we compare the first two elements and sort them by comparing them.
   b. Then we pick the third element and find its proper position among the previous two sorted elements.
   c. This way we gradually go on adding more elements to the already sorted list by putting them in their proper position.

For example :

```
def insertion_sort(InputList):
    for i in range(1, len(InputList)):
        j = i - 1
        nxt_element = InputList[i]
# Compare the current element with next one
        while (InputList[j] > nxt_element) and (j >= 0):
        InputList[j+1] = InputList[j]
          j=j - 1
        InputList[j+1] = nxt_element
list = [19,2,30,42,28,11,135,26]
insertion_sort(list)
print(list)
```

Output :

[2, 11, 19, 26, 28, 30, 42, 135]

Program : Refer Q. 4.3, Page 4–3T, Unit-4.

Que 5.15. **What will be the output after the following statements ?**

```
x = [25, 14, 53, 62, 11]
x.sort()
print(x)
```

Answer

[11, 14, 25, 53, 62]

Que 5.16. **What will be the output after the following statements ?**

```
x = ['25', 'Today', '53', 'Sunday', '15']
x.sort()
print(x)
```

Answer

['15', '25', '53', 'Sunday', 'Today']

Que 5.17. **What will be the output after the following statements ?**

```
x = {5:4, 8:8, 3:16, 9:32}
print(sorted(x.items()))
```

Answer

[(3, 16), (5, 4), (8, 8), (9, 32)]

Que 5.18. **What will be the output after the following statements ?**

```
x = ['a', 'b', 'c', 'A', 'B', 'C']
x.sort()
print(x)
```

Answer

['A', 'B', 'C', 'a', 'b', 'c']

Que 5.19. **What will be the output after the following statements ?**

```
x = ['a', 'b', 'c', 'A', 'B', 'C']
x.sort(key=str.lower)
print(x)
```

Answer

['a', 'A', 'b', 'B', 'c', 'C']

Que 5.20. **What will be the output after the following statements ?**

```
x = ['a', 'b', 'c', 'A', 'B', 'C']
x.sort(key=str.swapcase)
print(x)
```

Answer

['a', 'b', 'c', 'A', 'B', 'C']

Que 5.21. **What will be the output after the following statements ?**

```
x = ['a', 'b', 1, 2, 'A', 'B']
x.sort()
print(x)
```

Answer

TypeError

Que 5.22. **What will be the data type of the output after the following statements ?**

```
x = 'Python'
y = list(x)
print(y)
```

Answer

List

Que 5.23. **What will be the data type of the output after the following statements ?**

```
x = 'Python'
y = tuple(x)
print(y)
```

Answer

Tuple