**Selection Sort:**

**Description of Algorithm:**

Selection sort is the simplest sorting algorithm. In This Algorithm the entire list is divided into two parts, the sorted part is at the left side and the unsorted part is at the right side. Initially, the sorted part is empty and the unsorted part is the entire list.

This algorithm will first find the smallest element in the array and swap it with the element in the first position, then it will find the second smallest element and swap it with the element in the second position, and it will keep on doing this until the entire array is fully sorted.

**Pseudo Code:**

|  |
| --- |
| procedure selection sort  list : array of items  n : size of list  for i = 1 to n - 1  /\* set current element as minimum\*/  min = i  /\* check the element to be minimum \*/  for j = i+1 to n  if list[j] < list[min] then  min = j;  end if  end for  /\* swap the minimum element with the current element\*/  if indexMin != i then  swap list[min] and list[i]  end if  end for  end procedure |

**Code in Python:**

def selectionSort(array, size):

    for step in range(size):

        min\_idx = step

        for i in range(step + 1, size):

            if array[i] < array[min\_idx]:

                min\_idx = i

        (array[step], array[min\_idx]) = (array[min\_idx], array[step])

data = [-2, 45, 0, 11, -9]

size = len(data)

selectionSort(data, size)

print('Sorted Array in Ascending Order:')

print(data)

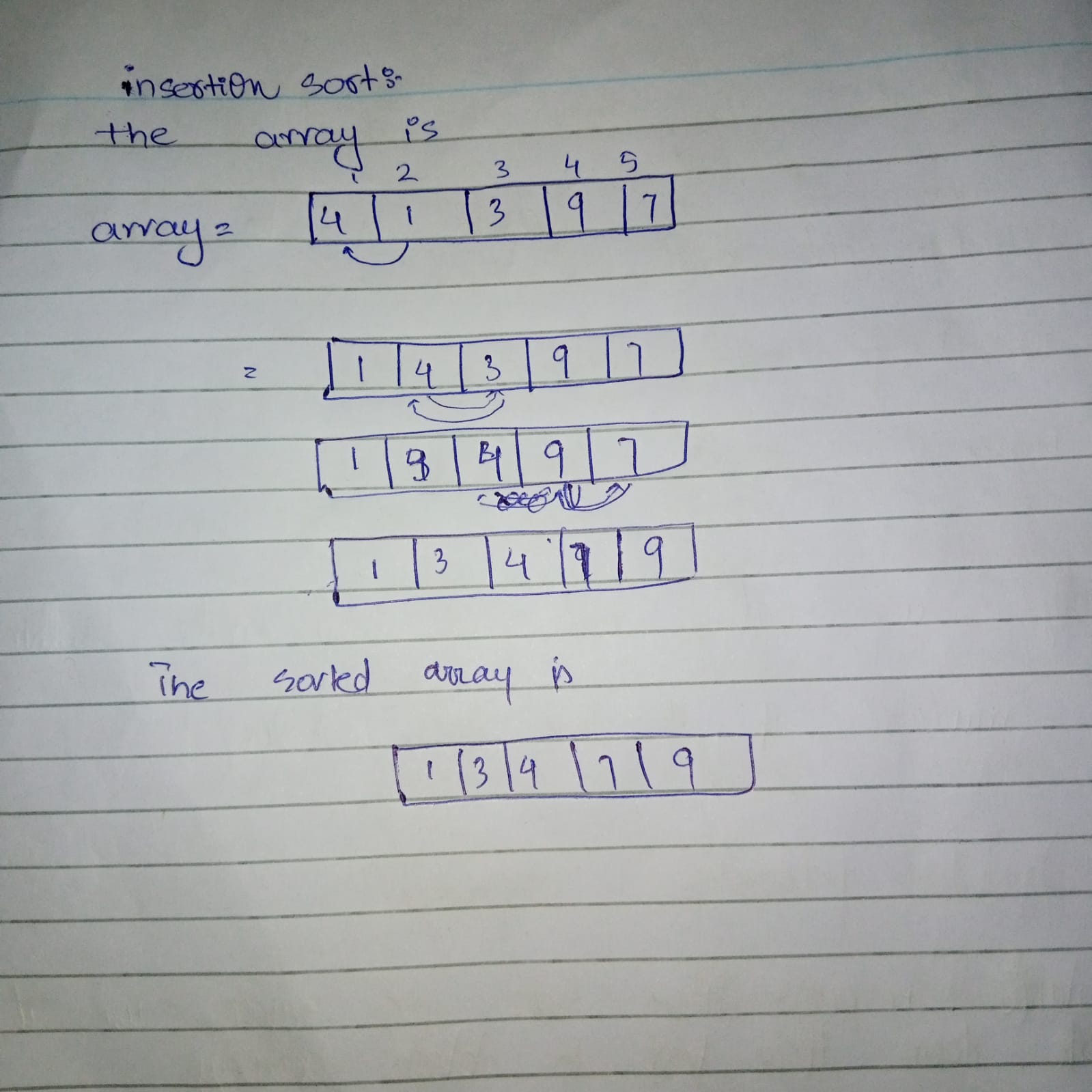
**Time Complexity:**

Selection Sort requires two nested for loops to complete itself, one for loop is in the function selectionSort, and inside the first loop we are making a call to another function indexOfMinimum, which has the second (inner) for loop.

The Time complexity for selection sort algorithm is given below:

* Worst Case Time Complexity [ Big-O ]: **O(n2)**
* Best Case Time Complexity [Big-omega]: **Ω (n2)**
* Average Case Time Complexity[Big-theta]: **Θ(n2)**

**Dry Run:**

****

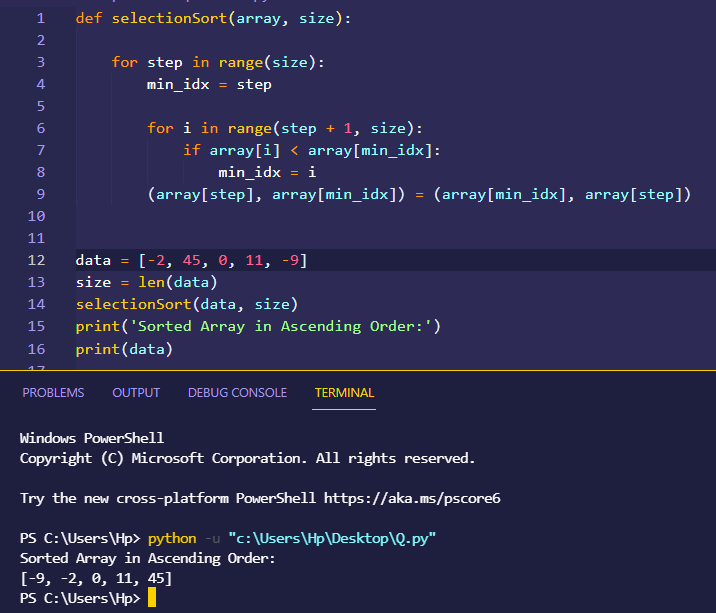
**3 Strengths:**

* It is very efficient
* It works very well for small data sets
* It Is very easy to implement

**3 Weaknesses:**

* It is slower comparative to other sorting algorithms
* Selection sort requires n-squared number of steps for sorting n elements.
* It is not very efficient when we are dealing with a huge list of items.

**Dry Run on Small Input:**

****

**Merge Sort:**

**Description of Algorithm:**

Merge sort is one of the most efficient sorting algorithm based on divide and conquer rule. It first divides the array into equal halves and then combines them in a sorted manner. It repeatedly breaks down the array into several sub arrays until each sub array consists of a single element and then it merges those sub arrays in a manner that results into a sorted array list.

**Pseudo Code:**

|  |
| --- |
| MergeSort(arr[], l, r)  If r > l  1. Find the middle point to divide the array into two halves:  middle m = l+ (r-l)/2  2. Call mergeSort for first half:  Call mergeSort(arr, l, m)  3. Call mergeSort for second half:  Call mergeSort(arr, m+1, r)  4. Merge the two halves sorted in step 2 and 3:  Call merge(arr, l, m, r) |

**Code in Python:**

def mergeSort(array):

    if len(array) > 1:

        r = len(array)//2

        L = array[:r]

        M = array[r:]

        mergeSort(L)

        mergeSort(M)

        i = j = k = 0

        while i < len(L) and j < len(M):

            if L[i] < M[j]:

                array[k] = L[i]

                i += 1

            else:

                array[k] = M[j]

                j += 1

            k += 1

        while i < len(L):

            array[k] = L[i]

            i += 1

            k += 1

        while j < len(M):

            array[k] = M[j]

            j += 1

            k += 1

def printList(array):

    for i in range(len(array)):

        print(array[i], end=" ")

    print()

if \_\_name\_\_ == '\_\_main\_\_':

    array = [4, 1, 3, 9, 7,]

    mergeSort(array)

    print("Sorted array is: ")

    printList(array)

**Time Complexity:**

Merge Sort is an efficient, stable sorting algorithm with an average, best-case, and worst-case time complexity of O (n log n)*.*

The Time complexity for merge sort algorithm is given below:

* Worst Case Time Complexity [ Big-O ]: **O(n log n)**
* Best Case Time Complexity [Big-omega]: **Ω (n log n)**
* Average Case Time Complexity[Big-theta]: **Θ(n log n)**

**Proof Of Correctness:**

**Initialization:**

In the insertion sort the subarray start with the first element of the array and then it is sorted the begin with.

**maintenance:**

The iteration of loop increase the size of sub array and when new element enters the array it will maintain the sorting property of sub array it is inserted where it is larger than its left one element.

**Terminology:**

The loop will be terminated when the sub array which is sorted to the size of actual array so the i indicated the size of original array*.*

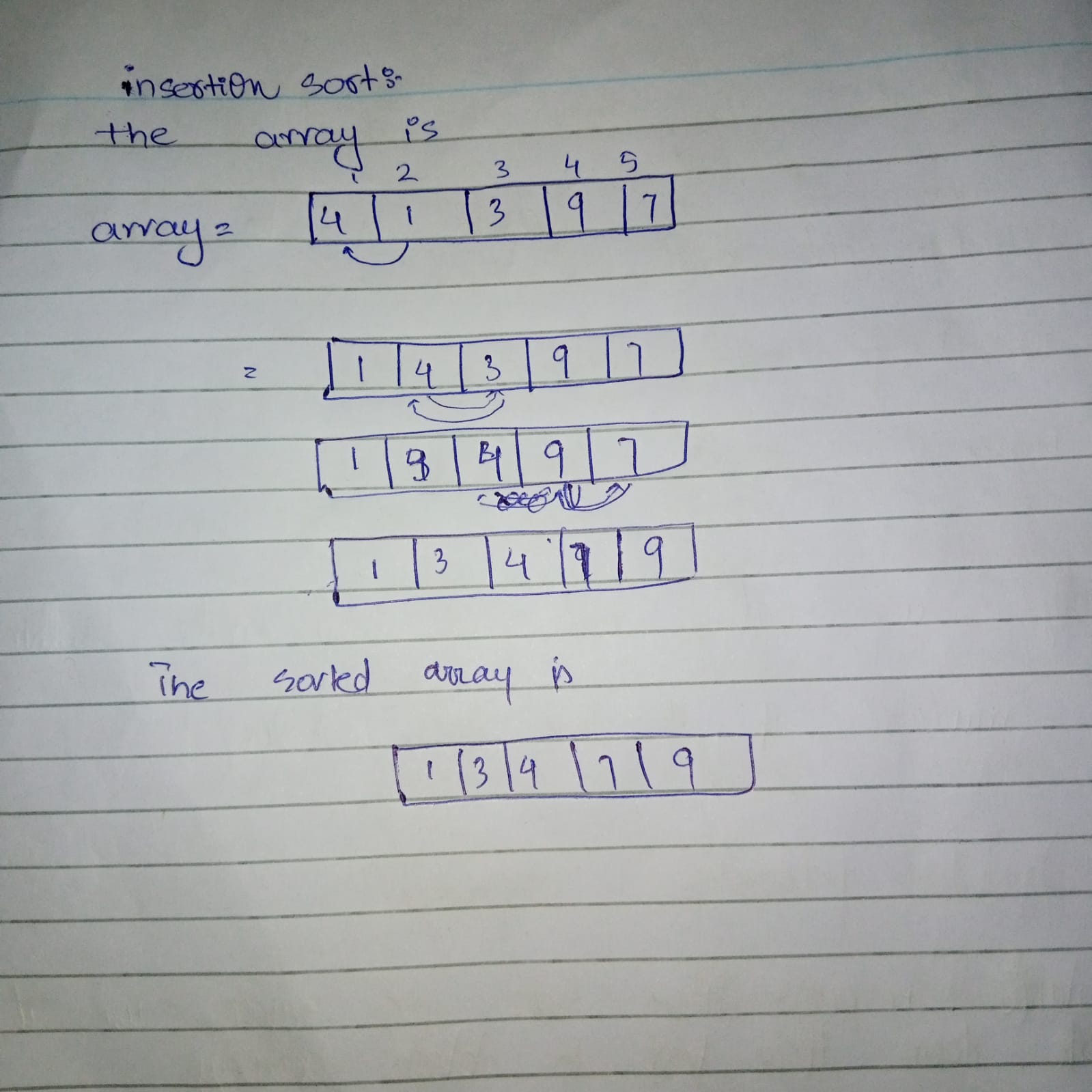
**3 Strengths:**

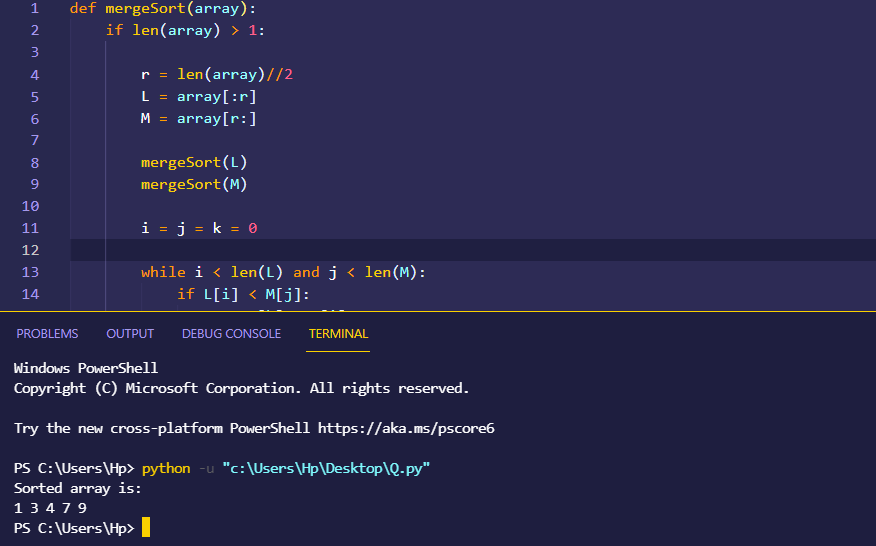
* It is very efficient
* It can be applied to files of any size.
* It works very well for larger inputs.

**3 Weaknesses:**

* It is slower comparative than other sorting algorithms for smaller inputs.
* Memory sort requires additional space.
* It goes through the whole process even if the array is sorted.

**Dry Run:**

****



**Insertion Sort:**

**Description of Algorithm:**Insertion Sort Algorithm is an in-place comparison based algorithm In Insertion sort, The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).It Is the most simplest sorting algorithm and works efficiently.

**Pseudo Code:**

|  |
| --- |
| INSERTION-SORT(A)  for i = 1 to n  key ← A [i]  j ← i – 1  while j > = 0 and A[j] > key  A[j+1] ← A[j]  j ← j – 1  End while  A[j+1] ← key  End for |

**Code in Python:**

def insertionSort(array):

    for step in range(1, len(array)):

        key = array[step]

        j = step - 1

        while j >= 0 and key < array[j]:

            array[j + 1] = array[j]

            j = j - 1

        array[j + 1] = key

data = [ 5, 7, -8, 9, 10, 4, -7, 0,-12, 1, 6, 2, 3, -4, -15, 12 ]

insertionSort(data)

print('Sorted Array in Ascending Order:')

print(data)

**Time Complexity:**

Insertion Sort is an easy-to-implement, stable sorting algorithm with time complexity of O (n²) in the average and worst case*,* and O (n)in thebest case*.*

The Time complexity for insertion sort algorithm is given below:

* Worst Case Time Complexity [ Big-O ]: **O(n2)**
* Best Case Time Complexity [Big-omega]: **Ω (n)**
* Average Case Time Complexity[Big-theta]: **Θ(n2)**

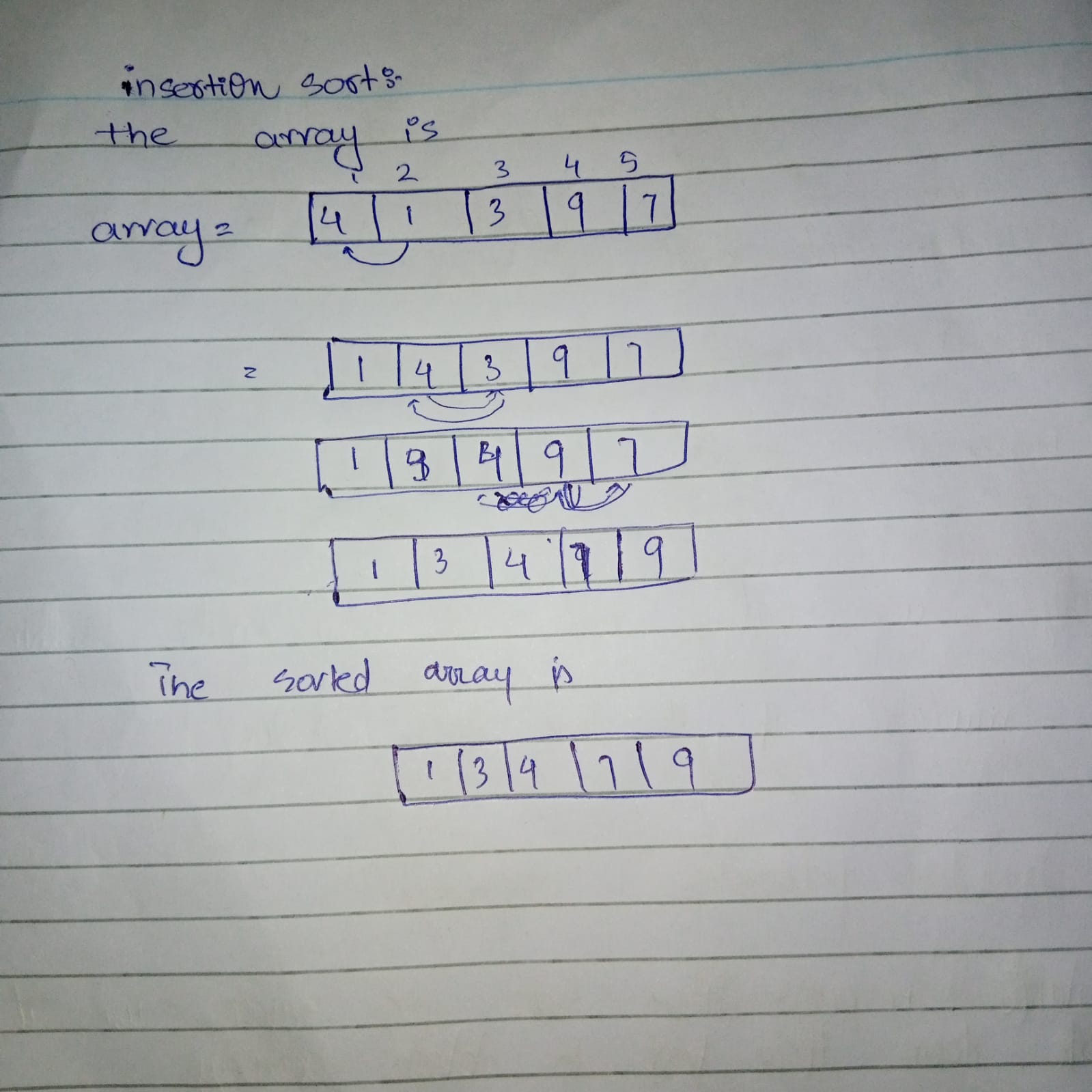
**3 Strengths:**

* It is very easy to implement. One of the simplest sorting algorithm.
* It is an in-place sorting algorithm so the memory requirement is minimum.
* It works very well for smaller inputs.

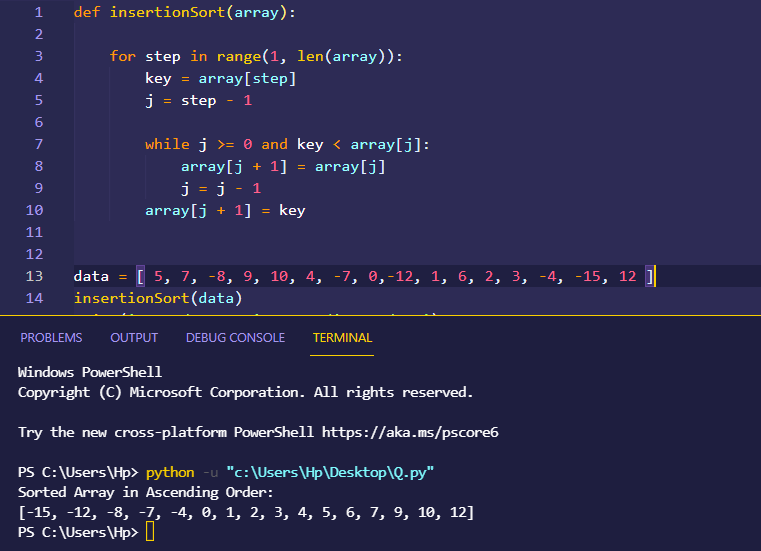
**3 Weaknesses:**

* Many Sorting algorithms are better in performance in comparison to insertion sort.
* The insertion sort is particularly useful only when sorting a list of few items.
* Insertion sort does not deal well with a huge input.

**Dry Run:**

****

**Dry run on small input:**



**Quick Sort:**

**Description of Algorithm:**

Quick sort is a highly efficient sorting algorithm. It is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value. It partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

**Pseudo Code:**

|  |
| --- |
| quickSort(arr[], low, high)  {  if (low < high)  {  /\* pi is partitioning index, arr[pi] is now  at right place \*/  pi = partition(arr, low, high);  quickSort(arr, low, pi - 1); // Before pi  quickSort(arr, pi + 1, high); // After pi  }  }  partition (arr[], low, high)  {  // pivot (Element to be placed at right position)  pivot = arr[high];  i = (low - 1) // Index of smaller element and indicates the  // right position of pivot found so far  for (j = low; j <= high- 1; j++)  {  // If current element is smaller than the pivot  if (arr[j] < pivot)  {  i++; // increment index of smaller element  swap arr[i] and arr[j]  }  }  swap arr[i + 1] and arr[high])  return (i + 1)  } |

**Code in Python:**

def partition(arr, low, high):

    i = (low-1)

    pivot = arr[high]

    for j in range(low, high):

        if arr[j] <= pivot:

            i = i+1

            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]

    return (i+1)

def quickSort(arr, low, high):

    if len(arr) == 1:

        return arr

    if low < high:

        pi = partition(arr, low, high)

        quickSort(arr, low, pi-1)

        quickSort(arr, pi+1, high)

    return arr

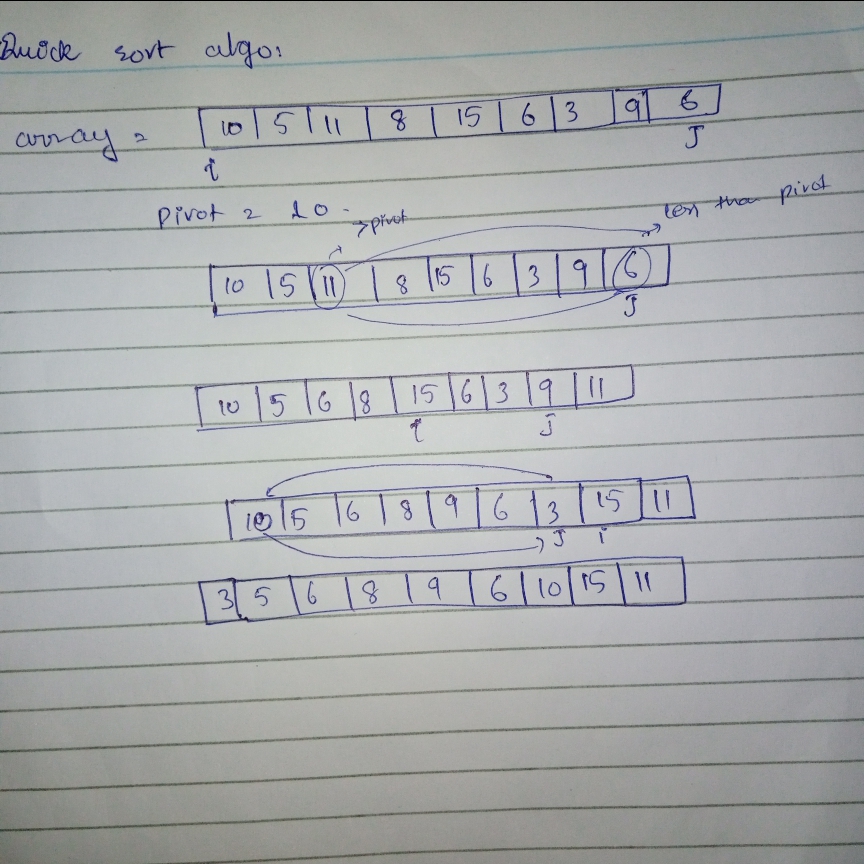
arr = [4, 1, 3, 9, 7]

n = len(arr)

quickSort(arr, 0, n-1)

print(arr)

**Dry Run:**

****

**Time Complexity:**

The Time complexity for quick sort algorithm is given below:

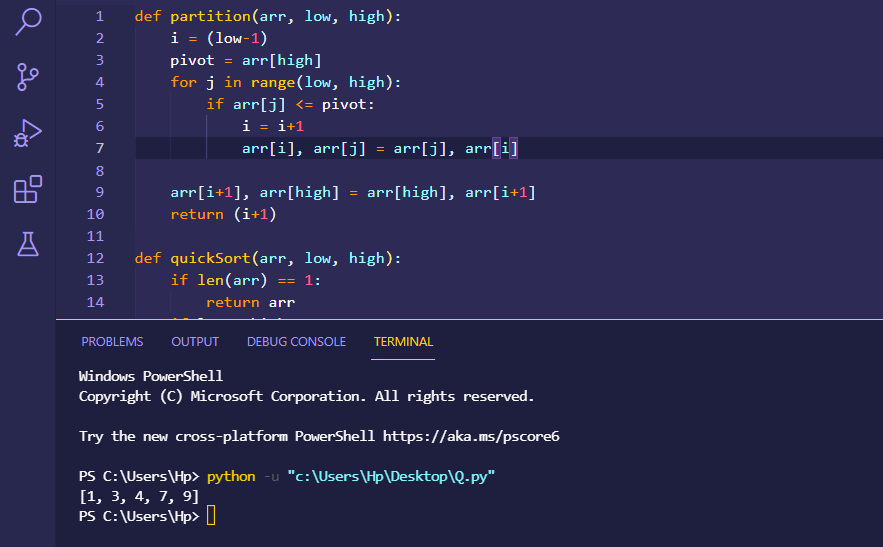
* Worst Case Time Complexity [ Big-O ]: **O(n2)**
* Best Case Time Complexity [Big-omega]: **Ω (n log(n))**
* Average Case Time Complexity[Big-theta]: **Θ(n log(n))**

**3 Strengths:**

* The quick sort is regarded as the best sorting algorithm.
* It can deal well with a huge list of items.
* As it sorts in place, so no additional storage is required.

**3 Weaknesses:**

* Its worst-case performance is similar to average performances of the bubble, insertion or selections sorts.
* If the list is already sorted than quick sort is not much efficient.
* It is fragile, i.e. a simple mistake in the implementation can go unnoticed and cause it to perform badly.

**Dry Run on Small Input:**

**Bubble Sort:**

**Description of Algorithm:**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

**Pseudo Code:**

|  |
| --- |
| procedure bubbleSort( list : array of items )  loop = list.count;  for i = 0 to loop-1 do:  swapped = false  for j = 0 to loop-1 do:  /\* compare the adjacent elements \*/  if list[j] > list[j+1] then  /\* swap them \*/  swap( list[j], list[j+1] )  swapped = true  end if  end for  /\*if no number was swapped that means  array is sorted now, break the loop.\*/  if(not swapped) then  break  end if  end for  end procedure return list |

**Code in Python:**

def bubbleSort(arr):

    n = len(arr)

    for i in range(n-1):

        for j in range(0, n-i-1):

            if arr[j] > arr[j + 1] :

                arr[j], arr[j + 1] = arr[j + 1], arr[j]

arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print ("Sorted array is:")

for i in range(len(arr)):

    print ("% d" % arr[i]),

**Proof Of Correctness:**

**Initialization:**

In the insertion sort the subarray start with the first element of the array and then it is sorted the begin with.

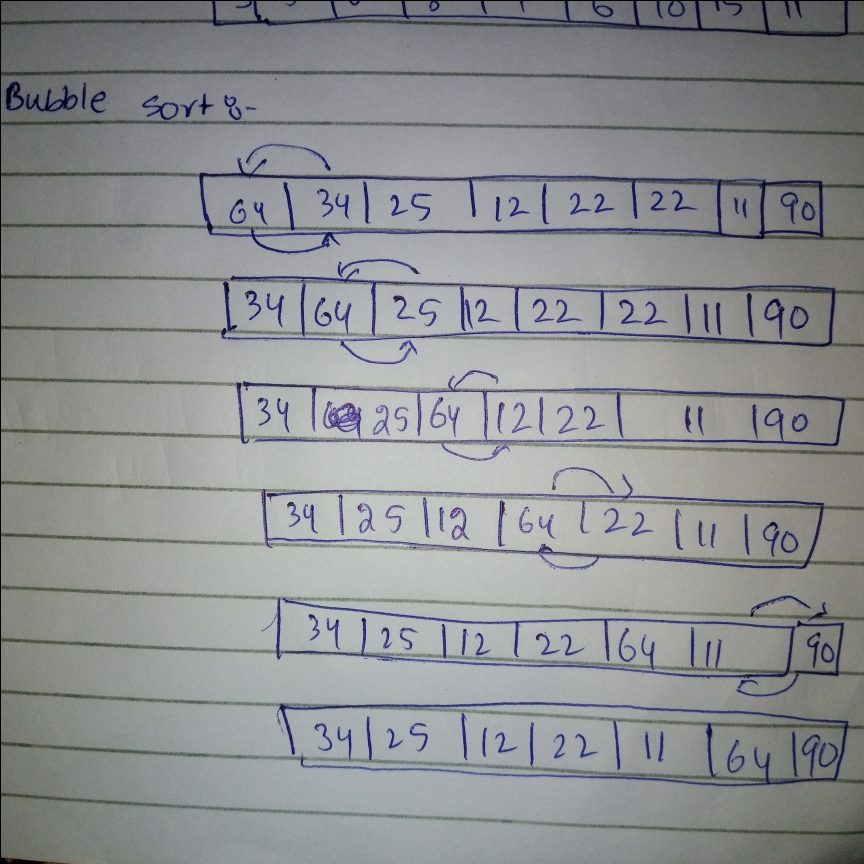
**maintenance:**

The iteration of loop increase the size of sub array and when new element enters the array it will maintain the sorting property of sub array it is inserted where it is larger than its left one element.

**Terminology:**

The loop will be terminated when the sub array which is sorted to the size of actual array so the i indicated the size of original array*.*

**Dry Run:**

****

**Time Complexity:**

Bubble Sort is an easy-to-implement, stable sorting algorithm with a time complexity given below.

The Time complexity for bubble sort algorithm is given below:

* Worst Case Time Complexity [ Big-O ]: **O(n2)**
* Best Case Time Complexity [Big-omega]: **Ω (n)**
* Average Case Time Complexity[Big-theta]: **Θ(n2)**

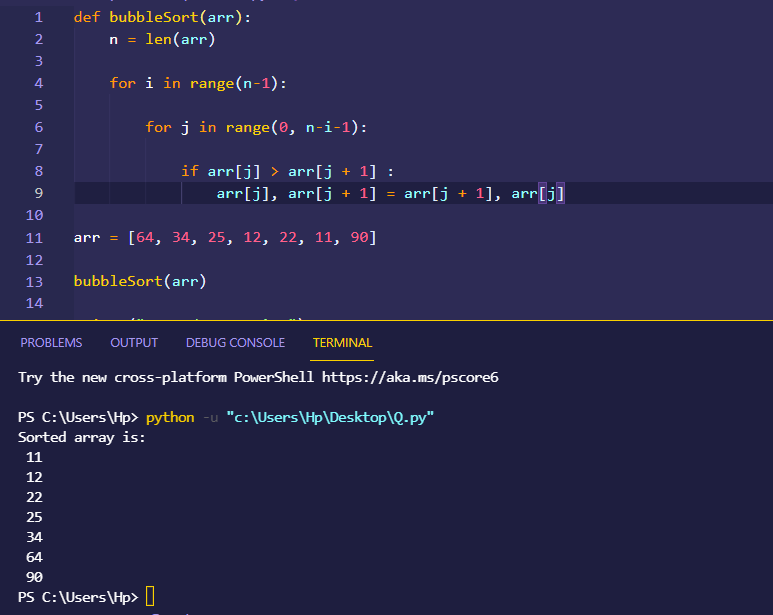
**3 Strengths:**

* The space requirement is at a minimum.
* Elements are swapped in place without using additional temporary storage.
* Bubble sort is very popular and easy to implement.

**3 Weaknesses:**

* The main disadvantage of the bubble sort is the fact that it does not deal well with a list containing a huge number of items.
* The bubble sort requires n-squared processing steps for every n number of elements to be sorted.
* The bubble sort is mostly suitable for academic teaching but not for real-life applications.

**Dry Run on Small Input:**



**Cube Sort:**

**Description of Algorithm:**

Cube Sort is a parallel sorting algorithm that builds a self-balancing multi-dimensional array from the keys to be sorted. As the axes are of similar length the structure resembles a cube. After each key is inserted the cube can be rapidly converted to an array.

**Code in Python:**

def sortArr(arr, n):

    arr = [(i \* i \* i, i) for i in arr];

    arr.sort()

    for i in range(n):

        print(arr[i][1], end = " ");

if \_\_name\_\_ == "\_\_main\_\_" :

    arr = [ 4, -1, 0, -5, 6 ];

    n = len(arr);

    sortArr(arr, n);

**Time Complexity:**

The Time complexity for Cube sort algorithm is given below:

* Worst Case Time Complexity [ Big-O ]: **O(n log n)**
* Best Case Time Complexity [Big-omega]: **Ω (n)**
* Average Case Time Complexity[Big-theta]: **Θ(nlog n)**

**3 Strengths:**

* It is a parallel sorting algorithm
* It uses a specialized binary search.
* It is very popular

**3 Weaknesses:**

* When an axis grows too large it is split.
* It is costly
* It is not easier to implement

****

**Heap Sort:**

**Description of Algorithm:**

Heap Sort is a popular and efficient sorting algorithm. It works by visualizing the elements of the array as a special kind of complete binary tree known as heap.

**Pseudo Code:**

|  |
| --- |
| void heapify(int arr[], int n, int i) {  // Find largest among root, left child and right child  int largest = i;  int left = 2 \* i + 1;  int right = 2 \* i + 2;    if (left < n && arr[left] > arr[largest])  largest = left;  if (right < n && arr[right] > arr[largest])  largest = right;  // Swap and continue heapifying if root is not largest  if (largest != i) {  swap(&arr[i], &arr[largest]);  heapify(arr, n, largest);  }  } |

**Code in Python:**

def heapify(arr, n, i):

*# Find largest among root and children*

      largest = i

      l = 2 \* i + 1

      r = 2 \* i + 2

      if l < n and arr[i] < arr[l]:

          largest = l

      if r < n and arr[largest] < arr[r]:

          largest = r

*# If root is not largest, swap with largest and continue heapifying*

      if largest != i:

          arr[i], arr[largest] = arr[largest], arr[i]

          heapify(arr, n, largest)

def heapSort(arr):

      n = len(arr)

*# Build max heap*

      for i in range(n//2, -1, -1):

          heapify(arr, n, i)

      for i in range(n-1, 0, -1):

*# Swap*

          arr[i], arr[0] = arr[0], arr[i]

*# Heapify root element*

          heapify(arr, i, 0)

arr = [1, 12, 9, 5, 6, 10]

heapSort(arr)

n = len(arr)

print("Sorted array is")

for i in range(n):

    print("%d " % arr[i], end='')

**Time Complexity:**

Bubble Sort is an easy-to-implement, stable sorting algorithm with a time complexity given below.

The Time complexity for bubble sort algorithm is given below:

* Worst Case Time Complexity [ Big-O ]: **O(nlog n)**
* Best Case Time Complexity [Big-omega]: **Ω (n log n)**
* Average Case Time Complexity[Big-theta]: **Θ(nlog n)**

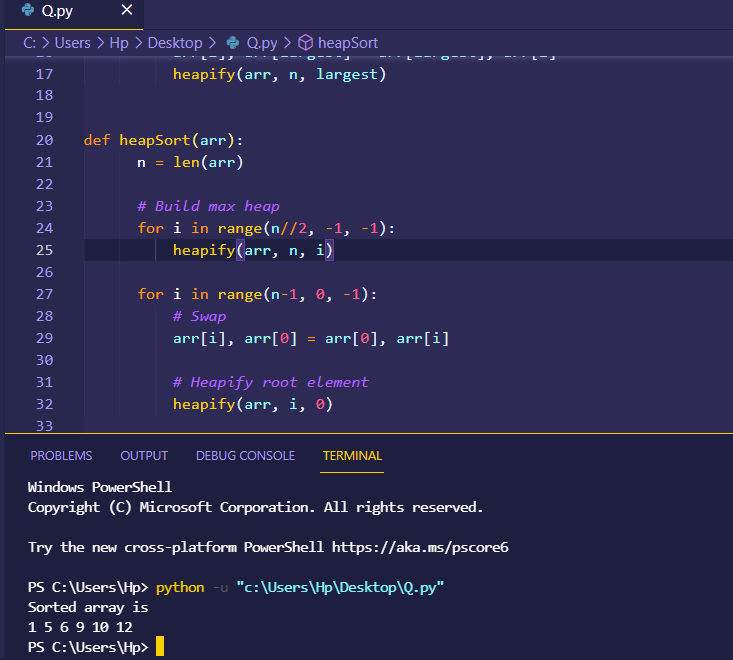
**3 Strengths:**

* Heap sort algorithm is very efficient.
* In-place sorting algorithm, less memory required.
* It is simpler to understand and can be implemented consistently.

**3 Weaknesses:**

* It is not much stable and the values may vary.
* It uses 0(1) memory space for the sorting operation.
* It takes more time

**Dry Run:**



**Counting sort:**

Counting sort is a popular and efficient sorting algorithm. Counting sort algorithm is used to sort the small input by dealing them as a small keys. Counting sort is non-comparing algorithm. An algorithm in which we cannot simply compare the two objects.

**Pseudo coded:**

COUNTING-SORT(A, B, k)

let C(0….. k) be a new array

for i =0 to k:

C[i]= 0

4 for j = 1 to A.length

5 C[A[j]] =C[A[j]]+1

6 C[i] now contains the number of elements equal to i.

7 for i=1 to k

8 C[i]=C[i]=C[i]+ C[i] -1

9 // C[i] now contains the number of elements less than or equal to i.

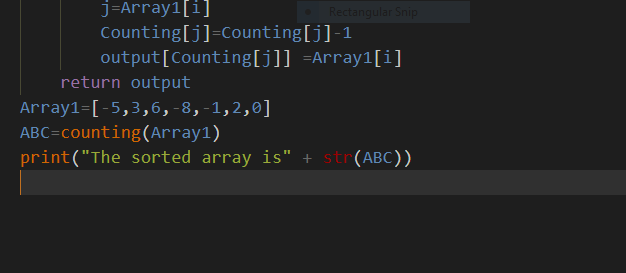
10 for j =A.length downto 1

11 B[C[A[j]]] = A[j]

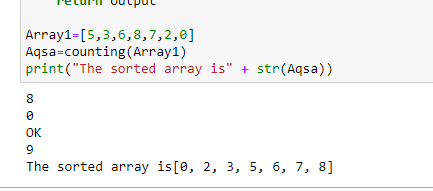
12 C[A[j]]= C[A[j]]-1

**Code in python:**

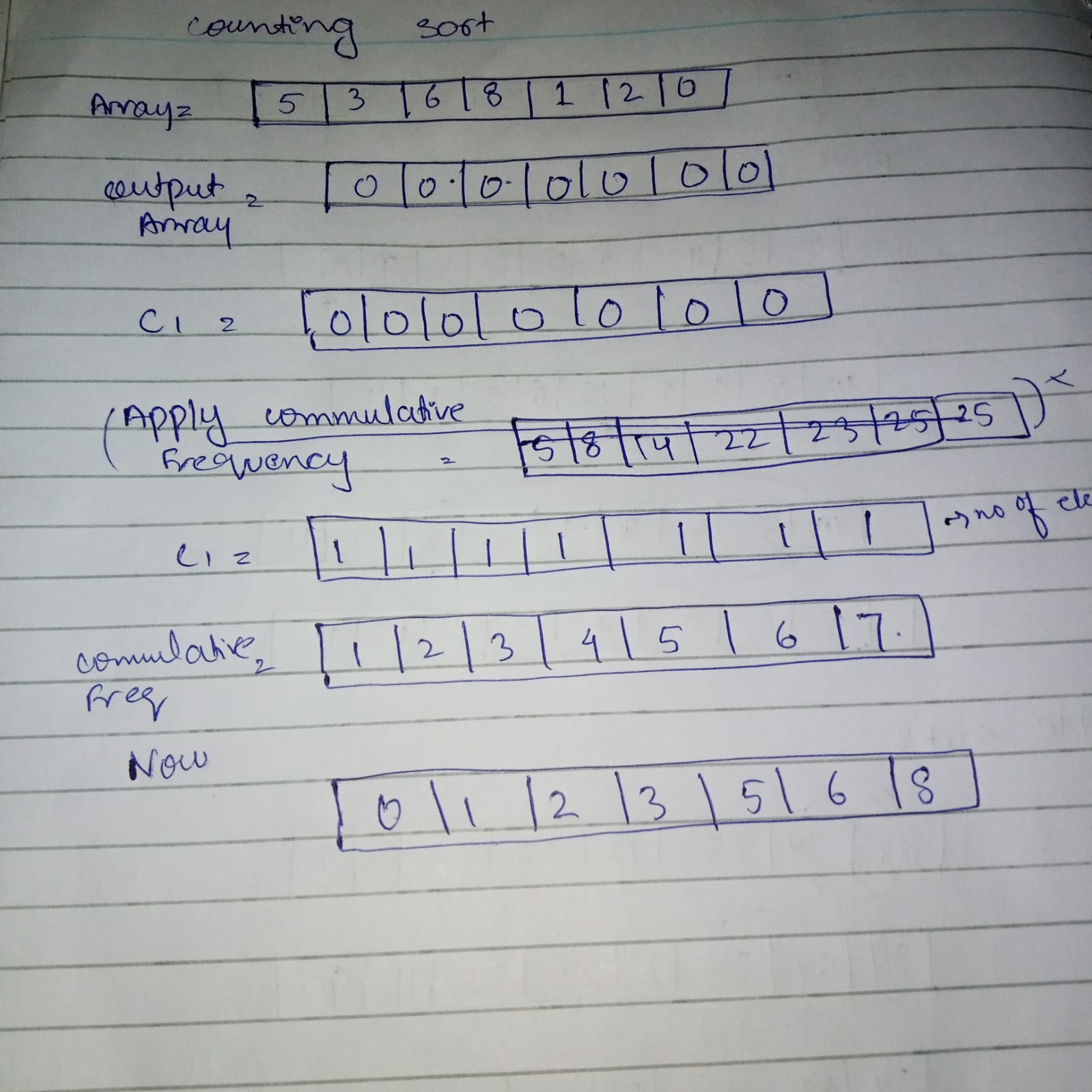
****

****

**Dry Run :**

****

**Dry Run:**

****

**Time Complexity:**

The Time complexity of the counting sort I O(n+k). Where n is the number of elements in the array and the k is the range of the elements. The Counting Sort is the best and an efficient algorithm to solve if the number of elements in the array almost equal to the range of the array.

* Time complexity of the counting sort is O(n+k)

**3 Strengths:**

* Stable Sorting Technique
* Efficient enough if the range of the array and the number of elements in the array is almost equal
* Best for the elements if the difference between the elements is small

**3 Weeknessess:**

* Non Comparison Algorithm
* Has big time complexity
* Has bid space complexity

**Proof of correctness:**

**Initialization:**

After the for loop of initializes the array C to all zeros, the for loop of lines 4–5 inspects each input element.

**Maintenance:**

If the value of an input element is i, we increment C[i]. Thus, C[i]. holds the number of input elements equal to.

**Bucket Sort:**

Bucket sort algorithm is a sorting algorithm which is done by dealing the elements if the array in the form of buckets. Firstly it deal with the elements and store al the elements in the different buckets and then you can solve the problem either by another sorting algorithm or by simply recursively calling the bucket algorithm.

**Pseudo Code:**

BUCKET-SORT(A):

1. let B(0 ….. n – 1)be a new array

2 .n = A.length

3. for i = 0 to n- 1:

4 make B[i] asan empty list

5 for i = 1 to n:

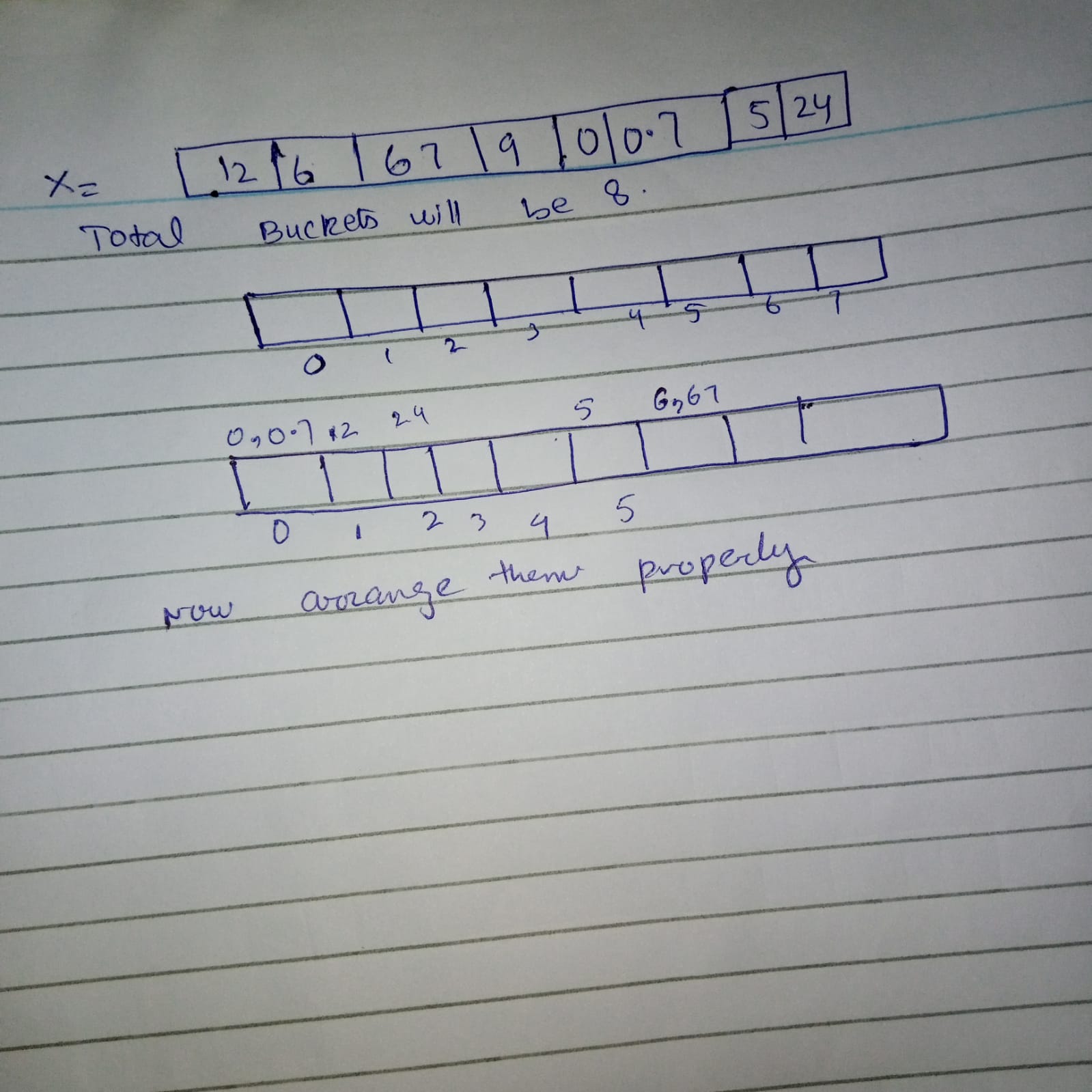
6 insert A[i]into list B[[nA[i]]]

7 for i = 0 to n – 1:

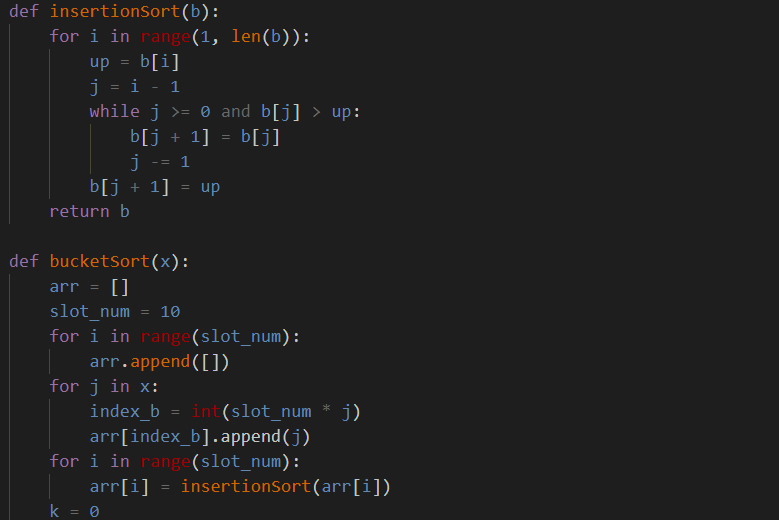
8 sort list B[i] with insertion sort

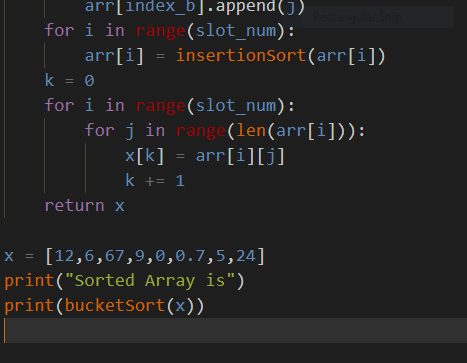
9 concatenate the lists B[0]; B[1]; : : : ; B[n - 1]together in order

**Dry Run:**

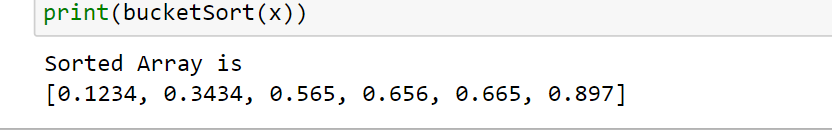
****

**Code in python:**

****

****

**Dry Run:**

****

**Time Complexity:**

Time Complexity is time taken by an algorithm to solve the given problem and produce a sorted output

* The best case time complexity of the bucket sort is O(n+k)
* The Worst time complexity of the bucket sort is O(n^2)

**3 Strengths:**

* Stable Sorting Technique
* Bucket sort id quicker than the bubble sort
* Putting the elements in to different buckets and than solve the problem reduce the number of the comparisons take place. Which reduce the time to run the program

**3 Weeknessess:**

* Non Comparison Algorithm
* Can not apply on all the data types you need a best bucketing scheme for this
* Has big time complexity

**Redix Sort Algorithm:**

Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value). Radix sort uses counting sort as a subroutine to sort an array of numbers.Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. In a typical computer, which is a sequential random-access machine, we sometimes use radix sort to sort records of information. Intuitively, you might sort numbers on their most significant digit, sort each of the resulting bins recursively, and then combine the decks in order.

**Proof of correctness:**

**Initlization:**

The correctness of radix sort follows by induction on the column being sorted Maintenance

**Maintenance:**

The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to k, so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time. n C k.

**Termination:**

There are d passes, and so the total time for radix sort is d\*n C k. When d is constant and k D O.(n), we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits

**Pseudo Cord:**

Radix-Sort(A, d)

for j = 1 to d do

int count[10] = {0};

for i = 0 to n do

count[key of(A[i]) in pass j]++

for k = 1 to 10 do

count[k] = count[k] + count[k-1]

for i = n-1 downto 0 do

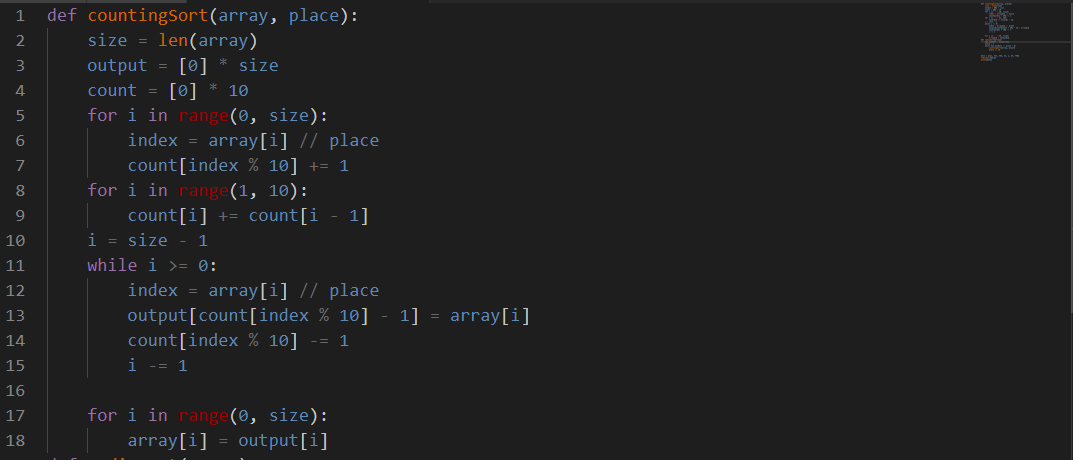
result[ count[key of(A[i])] ] = A[j]

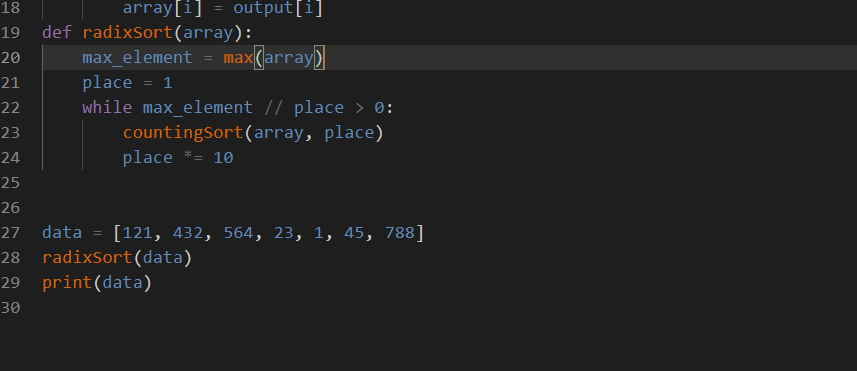
count[key of(A[i])]--

for i=0 to n do

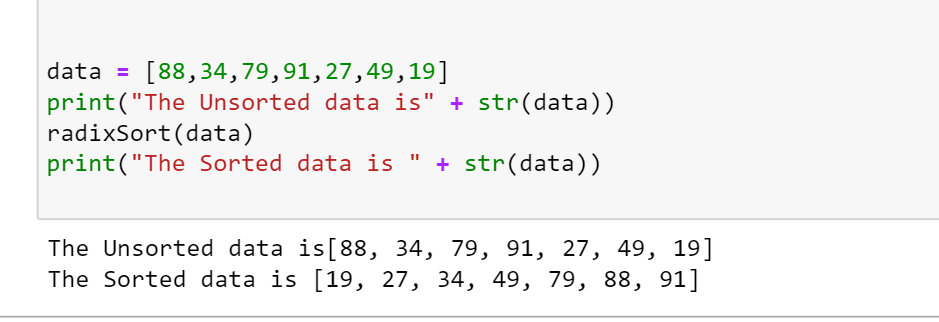
A[i] = result[i]

**Python code:**

****

****

**Dry Run:**

****

**Three weaknesses:**

* A redix sort is cannot be used for the non –integers. It can only be applicable for the integer numbers
* Redix sort take more time than the Quick sort (space also)
* Redix sort is harder to generalize than all other algorithms

**Three Strengthes:**

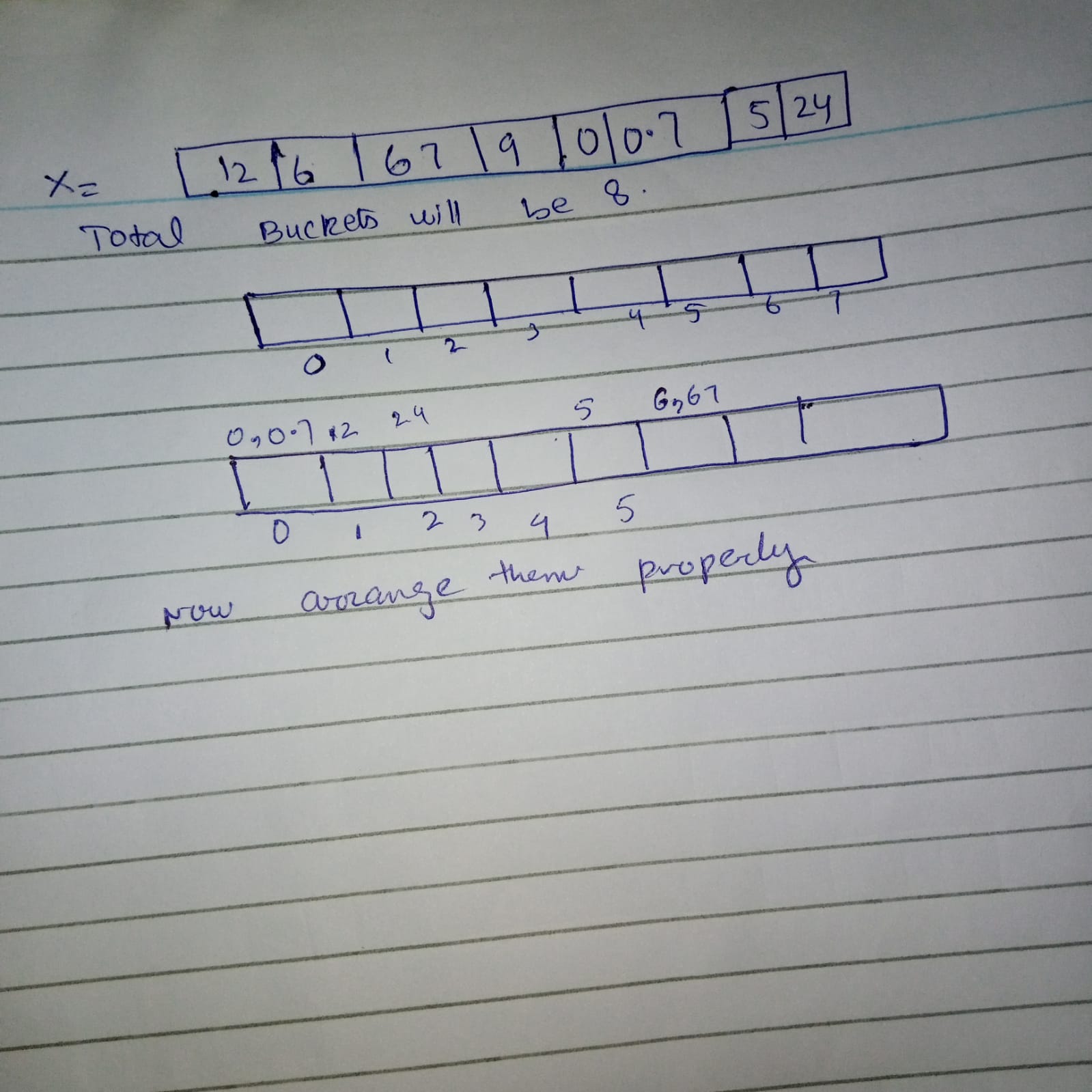
* Non comparative sorting Algorithm
* One of the most popular algorithm
* Having One time complexity for all inputs
* Space complexity of the radix sort is better than the counting sort.

**Time complexity:**

Time Complexity is time taken by an algorithm to solve the given problem and produce a sorted output

* Worst Case Time Complexity [ Big-O ]: **O(n)**
* Best Case Time Complexity [Big-omega]: **Ω (n log n)**
* Average Case Time Complexity[Big-theta]: **Θ(nlog n)**

**Dry Run:**

****

**Shell Sort:**

Shell sort algorithm is also known as shellsort or shell method. Shell sort algorithm is used to solve the algorithms in the form of the group form far a part from each other and than progreesivly start reducing the space between them.

**Pseudo Code:**

foreach (gap in gaps)

for (i = gap; i < n; i += 1)

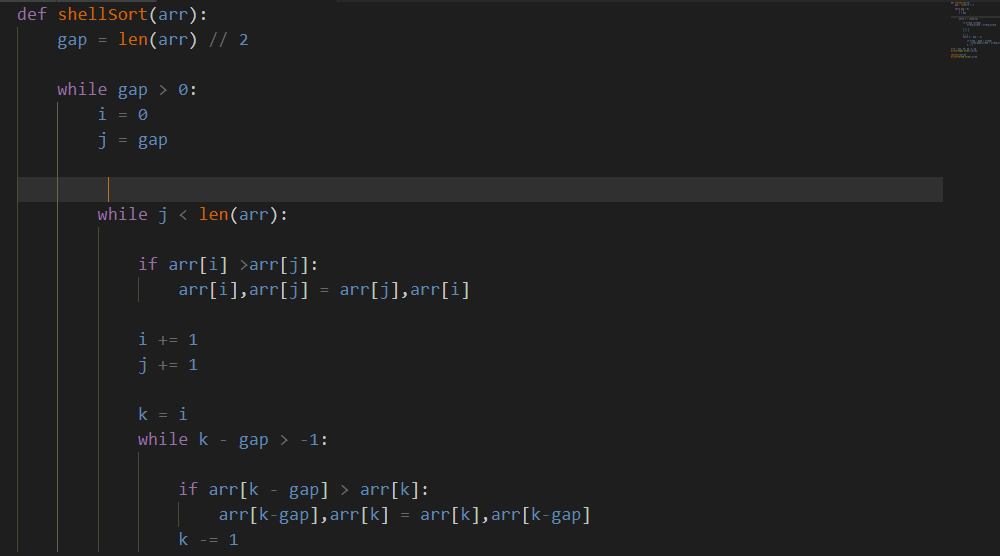
temp = a[i]

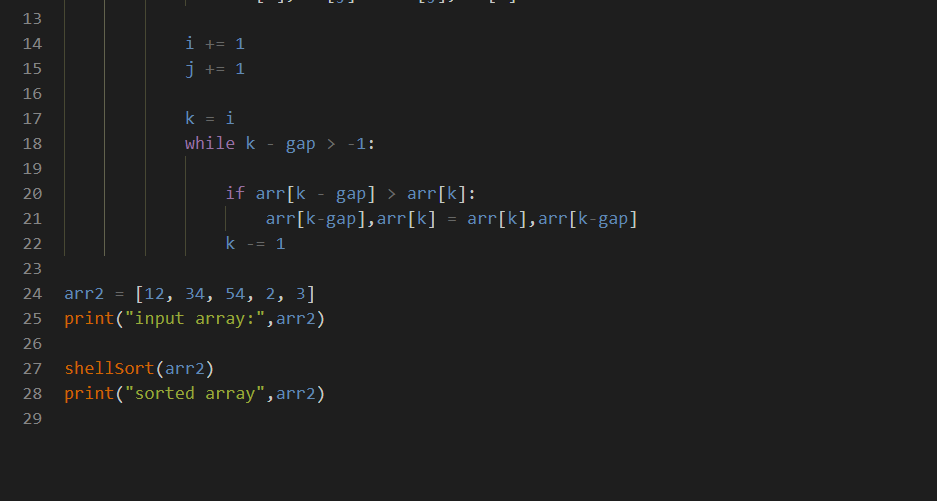
for (j = i; j >= gap and a[j - gap] > temp; j -= gap)

a[j] = a[j - gap]

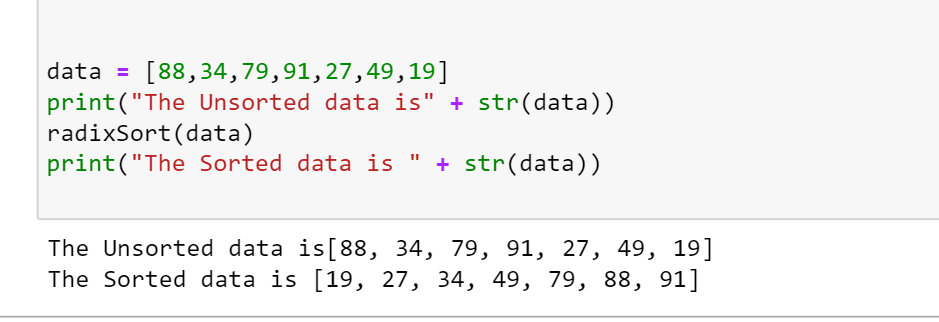
a[j] = temp

**Code in python:**





**DRY RUN:**



**Time Complexity:**

Time Complexity is time taken by an algorithm to solve the given problem and produce a sorted output

* Worst Case Time Complexity [ Big-O ]: **O(n)**
* Best Case Time Complexity [Big-omega]: **Ω ( n)**
* Average Case Time Complexity[Big-theta]: **Θ( n)**

**Three strengths:**

* Best efficient algorithm
* This algorithm avoids large shifts as in case of insertion sort
* One of the most popular algorithm

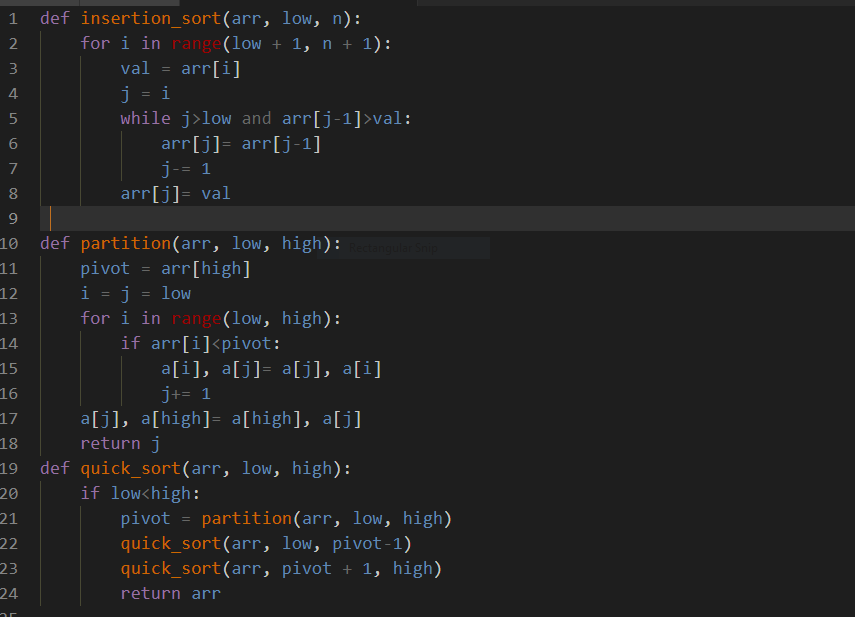
**Three weeknesses:**

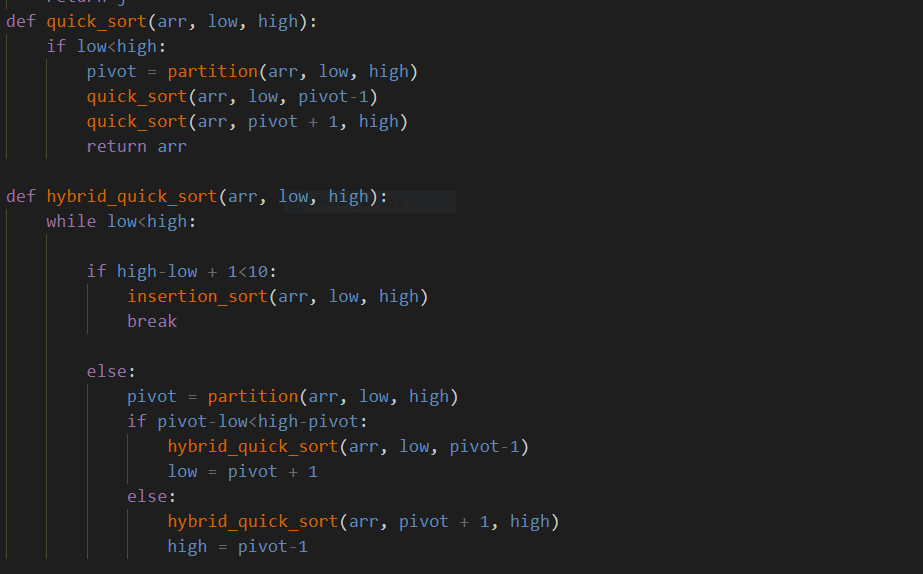
1. Shell sort is that its only efficient for medium size lists
2. For bigger lists, the algorithm is not the best choice
3. Has large time complexity

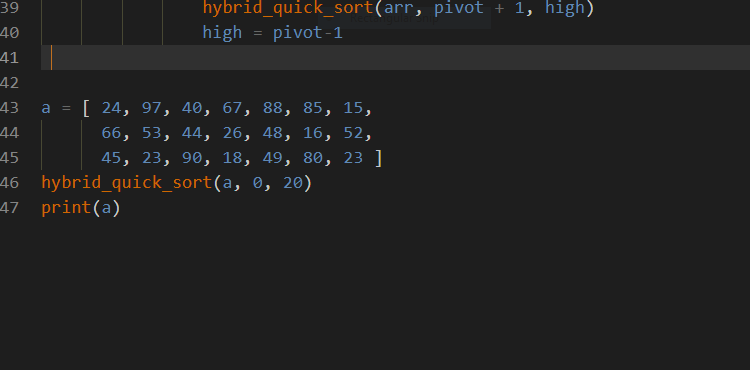
**Hybrid Sort:**

A Hybrid Algorithm is an algorithm that combines two or more other algorithms that solve the same problem, either choosing one (depending on the data) or switching between them throughout the algorithm.In computer science, hybrid algorithms are very common in optimized real-world implementations of recursive algorithms, particularly implementations of divide and conquer or decrease and conquer algorithms, where the size of the data decreases as one moves deeper in the recursion. In this case, one algorithm is used for the overall approach but deep in the recursion, it switches to a different algorithm, which is more efficient on small data.

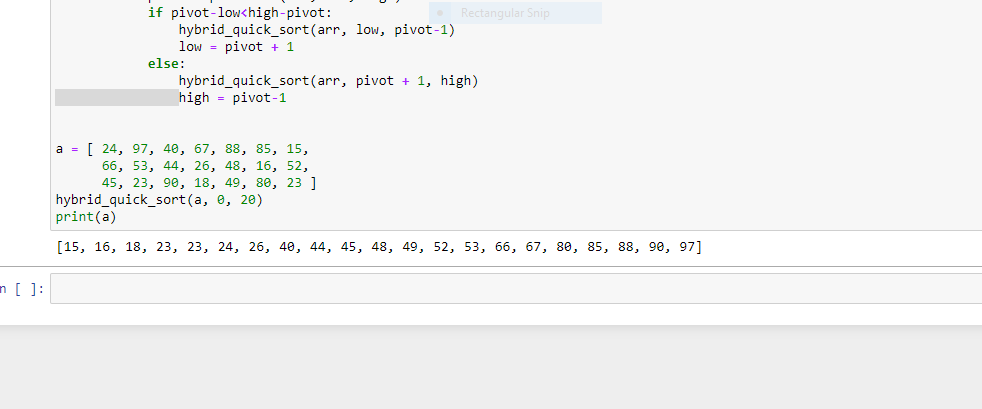
**Pseudo code:**







**Dry Run:**

****

**Time Complexity:**

Time Complexity is time taken by an algorithm to solve the given problem and produce a sorted output

* Worst Case Time Complexity [ Big-O ]: **O(n^2)**
* Best Case Time Complexity [Big-omega]: **Ω (n)**
* Average Case Time Complexity[Big-theta]: **Θ(n)**

**3 Weeknessess:**

* Non Comparison Algorithm
* Has big time complexity
* Has bid space complexity

**3 Strengths:**

* Stable Sorting Technique
* Hybrid sort id quicker than the bubble sort
* This algorithm avoids large shifts as in case of insertion sort