Assignment-3

Name – Ujjwal Lehri
Roll No. – 60216403224
Course – B.Tech CSE 4th sem

Q1. List all the sorting algorithms and read about them.
Q2. Write a program in C/C++, show its output, plot the graph, and calculate it's time complexity for:
1. Quick Sort
2. Merge Sort
3. Heap Sort

Q3. Write a program in C/C++, show its output, plot the graph, and calculate it's time complexity for:
1. Counting Sort
2. Radix Sort
3. Bucket Sort

---

1) Sorting Algorithms:

Comparison-Based Sorting Algorithms

1. **Bubble Sort:** Repeatedly swaps adjacent elements if they are in the wrong order; simple but inefficient ($O(n^2)$).
2. **Selection Sort:** Selects the smallest element and swaps it with the current position; $O(n^2)$ time complexity.
3. **Insertion Sort:** Builds a sorted list by inserting elements in their correct position one by one; $O(n^2)$ worst-case, but efficient for nearly sorted data.
4. **Merge Sort:** Recursively splits the array into halves, sorts them, and merges them back together, O(n log n) complexity.
5. **Quick Sort:** Uses a pivot element to partition the array, then recursively sorts each partition, O(n log n) average case, $O(n^2)$ worst case.
6. **Heap Sort:** Converts the array into a heap structure, then repeatedly extracts the maximum/minimum element; O(n log n) complexity.
7. **Shell Sort:** Variation of insertion sort that allows exchanging far-apart elements to reduce swaps; complexity varies, typically O(n^(3/2)).
8. **Tim Sort:** Hybrid of merge and insertion sort; used in Python's built-in sort function; O(n log n) worst case.
9. **Comb Sort:** Improved Bubble Sort with shrinking gap intervals to eliminate turtles (small values near the end).
10. **Cocktail Shaker Sort** – A bidirectional bubble sort that sorts in both directions per pass.

11. **Gnome Sort** – Similar to insertion sort but moves elements by swapping instead of shifting.
12. **Odd-Even Sort (Brick Sort)** – A parallel sorting algorithm that compares and swaps adjacent pairs.
13. **Pancake Sort** – Repeatedly flips the largest unsorted element to the top and then moves it to its correct position.
14. **Stooge Sort** – Recursively sorts the first two-thirds and last two-thirds of the array; very inefficient O(n^(log 3/log 1.5)).
15. **Bogo Sort** – Randomly shuffles elements until they are sorted; extremely inefficient, O(n!) in the worst case.
16. **Tree Sort** – Inserts elements into a Binary Search Tree (BST) and retrieves them in sorted order; O(n log n) average case.
17. **Smooth Sort** – Variant of heap sort with better performance on partially sorted data.
18. **Patience Sorting** – Based on the card game "Patience," used in Longest Increasing Subsequence problems.

Non-Comparison-Based Sorting Algorithms

19. **Counting Sort** – Uses counting of element frequencies to determine their position; O(n + k), where k is the range of input values.
20. **Radix Sort** – Sorts numbers digit by digit (starting from the least significant digit); O(nk), where k is the digit length.
21. **Bucket Sort** – Distributes elements into buckets, sorts each bucket, and then merges them; O(n + k), effective when data is uniformly distributed.
22. **Pigeonhole Sort** – Places elements in their respective "holes" (bins) based on their values; O(n + k).

Hybrid Sorting Algorithms

19. **IntroSort** – Starts with QuickSort, switches to HeapSort when recursion depth is too high, and uses Insertion Sort for small partitions; O(n log n).
23. **Flash Sort** – Uses a combination of distribution and insertion sorting; very fast for large datasets (O(n)).

2) Sorting Algorithms of O(nlogn)

Code:

```
#include<stdio.h>
#include<windows.h>
#include<stdlib.h>
#include<time.h>

void swap(int* arr, int i, int j);
void printArray(int arr[], int size);
int getRandom(int min, int max);
int* GenArr(int size);

//Quick Sort
int pivot(int *arr, int left, int right);
void quickSort(int *arr, int left, int right);

//MergeSort
void mergeSort(int arr[], int i, int j);
void merge(int arr[], int i, int mid, int j);

//HeapSort
void HeapSort(int Arr[], int n);
void buildHeap(int Arr[], int n);
void Heapify(int Arr[], int n, int i);

int main(){
   LARGE_INTEGER freq, start, end;
   QueryPerformanceFrequency(&freq);
   srand(time(0));
   int Sizes[] = {100, 1000, 10000, 100000, 1000000};
   int n = sizeof(Sizes)/sizeof(Sizes[0]);

   printf("quick sort: \n");
   printf("Input size \t   time taken\n");
   printf("-----------------------------------\n");
   for(int i=0; i<n; i++){
      int size = Sizes[i];
      int *Arr = GenArr(size);
      QueryPerformanceCounter(&start);
      quickSort(Arr, 0, size - 1);
      QueryPerformanceCounter(&end);
      double time_taken = (double)(end.QuadPart - start.QuadPart) * 1e9 / freq.QuadPart;
      printf("%-18d  %-5.2lf ns\n", size, time_taken);

      free(Arr);
   }
```

```
    printf("-------------------------------------------------------------------------------\n");
    printf("merge sort: \n");
    printf("Input size \t   time taken\n");
    printf("-----------------------------------\n");
    for(int i=0; i<n; i++){
        int size = Sizes[i];
        int *Arr = GenArr(size);

        QueryPerformanceCounter(&start);
        mergeSort(Arr, 0, size-1);
        QueryPerformanceCounter(&end);
        double time_taken = (double)(end.QuadPart - start.QuadPart) * 1e9 / freq.QuadPart;
        printf("%-18d  %-5.2lf ns\n", size, time_taken);
        free(Arr);
    }


    printf("-------------------------------------------------------------------------------\n");
    printf("Heap sort: \n");
    printf("Input size \t   time taken\n");
    printf("-----------------------------------\n");

    for(int i=0; i<n; i++){
        int size = Sizes[i];
        int *Arr = GenArr(size);

        QueryPerformanceCounter(&start);
        HeapSort(Arr, size);
        QueryPerformanceCounter(&end);

        double time_taken = (double)(end.QuadPart - start.QuadPart) * 1e9 / freq.QuadPart;
        printf("%-18d  %-5.2lf ns\n", size, time_taken);

        free(Arr);
    }
}
```

```
/*****************************************************************************/
/*                                  Quick Sort                                 */
/*****************************************************************************/

int pivot(int *arr, int left, int right){
   int pivotIndex = left;
   int swapIndex = pivotIndex;

   for(int i = pivotIndex + 1; i<= right ; i++){
      if(arr[i] < arr[pivotIndex]){
         swapIndex++;
         swap(arr, swapIndex , i);
      }
   }

   swap(arr, swapIndex, pivotIndex);
   return swapIndex;
}

void quickSort(int *arr, int left, int right){
   if(left < right){
      int index = pivot(arr, left, right);
      quickSort(arr, left, index-1);
      quickSort(arr,index+1, right);
   }
}
/*****************************************************************************/
/*                                  Merge Sort                                 */
/*****************************************************************************/

void merge(int arr[], int i, int mid, int j){
   int mergedArray[j-i+1];
   int Index=0;
   int x = i, y = mid+1;

   while(x <= mid && y <= j){
      if(arr[x] <= arr[y]){
         mergedArray[Index] = arr[x];
         x++;
      } else{
         mergedArray[Index] = arr[y];
         y++;
      }
      Index++;
   }

   while(x<=mid){
      mergedArray[Index] = arr[x];
```

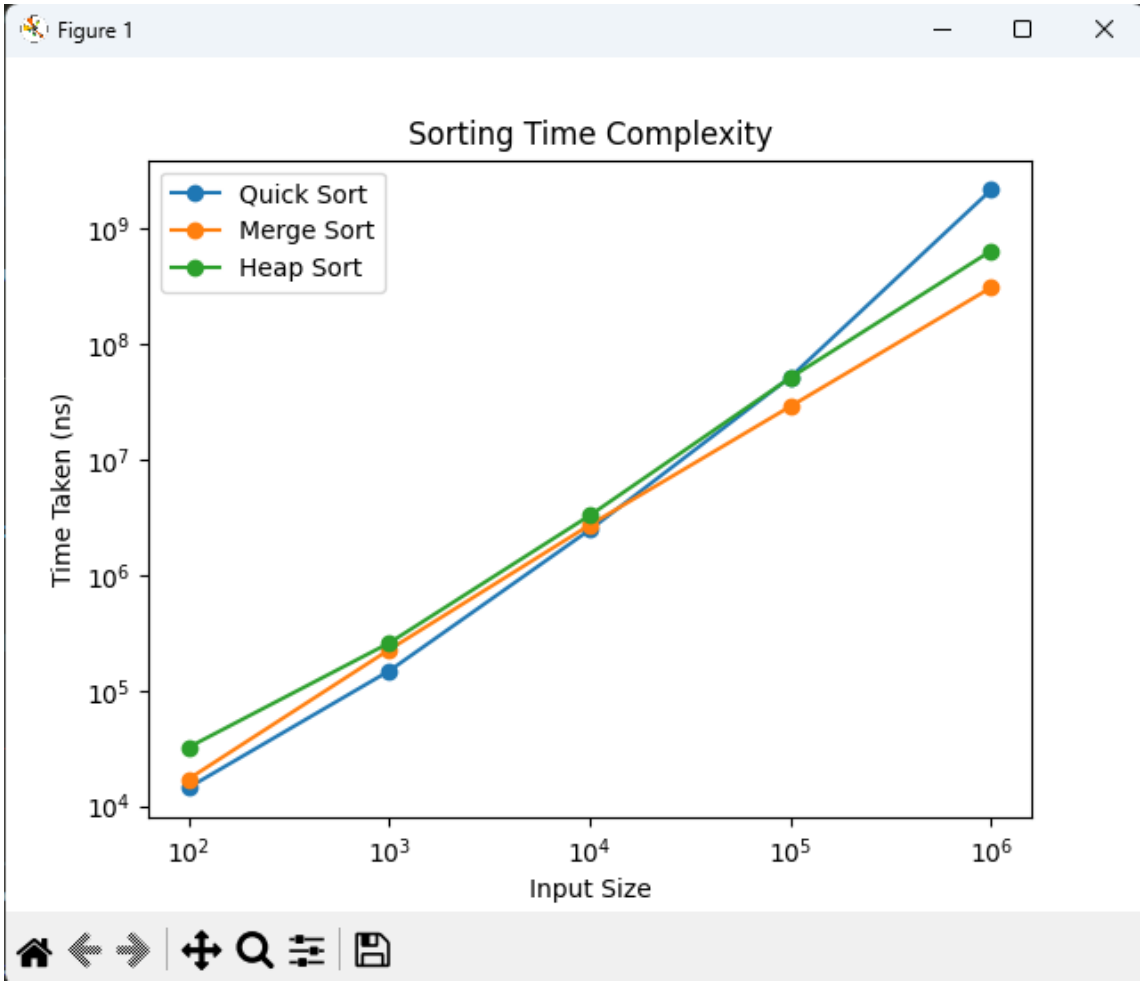
```
      Index++;
      x++;
   }

   while(y<=j){
      mergedArray[Index] = arr[y];
      Index++;
      y++;
   }

   for (int k = 0; k < Index; k++) {
      arr[i + k] = mergedArray[k];
   }

}

void mergeSort(int arr[], int i, int j){
   if(i>=j)  return;

   int mid = i + (j-i)/2;

   mergeSort(arr, i, mid);
   mergeSort(arr, mid+1, j);
   merge(arr, i, mid, j);
}


/*****************************************************************************/
/*                                Heap Sort (using Max Heap)                 */
/*****************************************************************************/

void Heapify(int Arr[], int n, int i){
   int l,r,max;

   l = 2*i+1;
   r = 2*i+2;
   max = i;
   if(l<n && Arr[l] > Arr[max]){
      max = l;
   }
   if(r<n && Arr[r] > Arr[max]){
      max = r;
   }
   if( max != i){
      swap(Arr, i, max);
      Heapify(Arr, n, max);
   }
}
```

```
void buildHeap(int Arr[], int n){
   for(int i = n/2; i >= 0; i--){
      Heapify(Arr, n, i);
   }
}

void HeapSort(int Arr[], int n){
   buildHeap(Arr, n);
   for(int i = n-1; i>=0; i--){
      swap(Arr, 0, i);
      Heapify(Arr, i, 0);
   }
}



/*****************************************************************************/
/*                                Helper Functions                           */
/*****************************************************************************/

void swap(int* arr, int i, int j) {
   int temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
}

void printArray(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}

int getRandom(int min, int max){
   return min + rand()%(max-min);
}

int* GenArr(int size){
   int min = 1, max = 999;
   int* Arr = (int*)malloc(sizeof(int) * size);
   for (int i = 0; i < size; i++) {
      Arr[i] = getRandom(min, max);
   }
   return Arr;
}
```

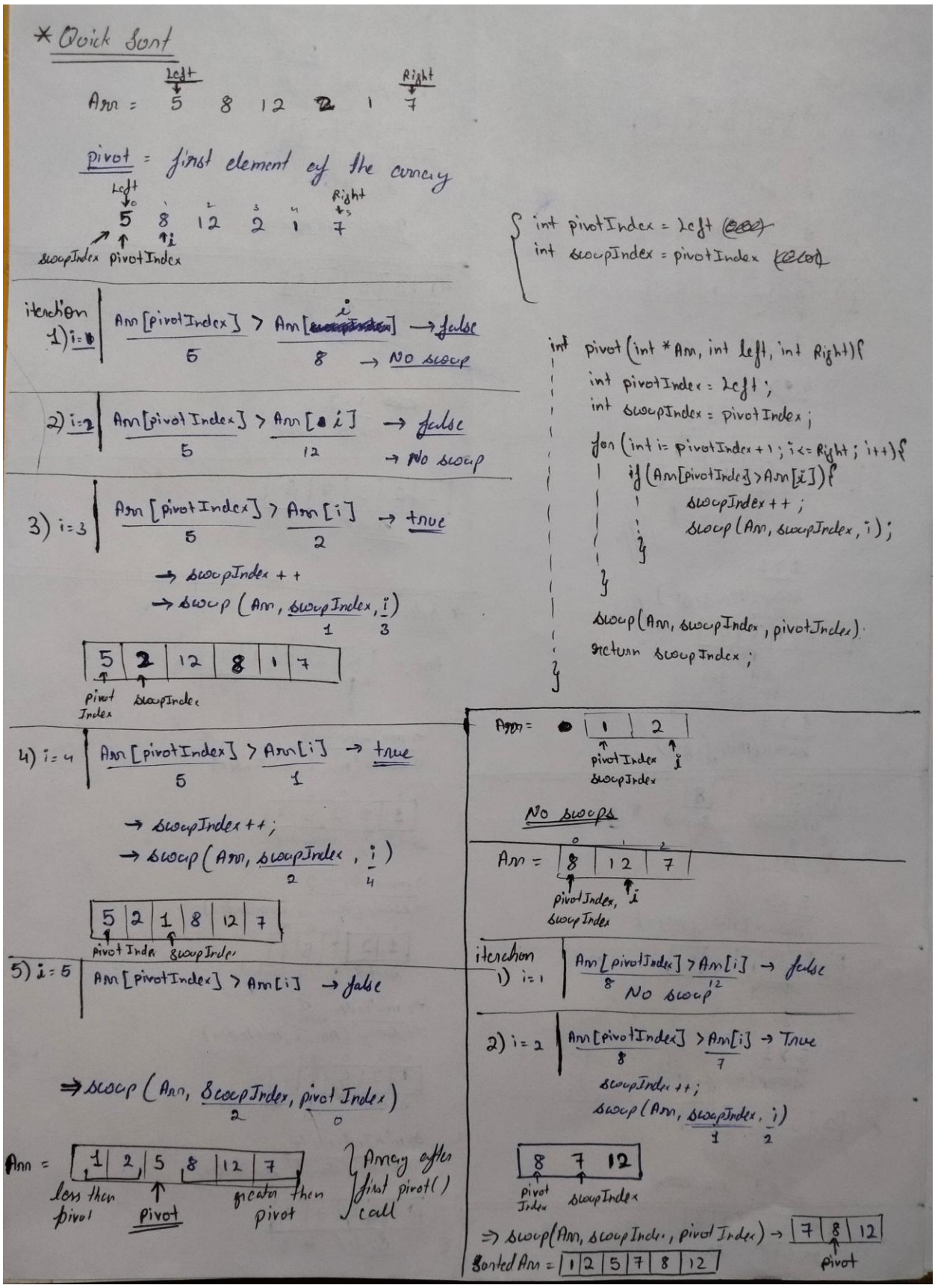
Output:

```
c:\Users\Ujjwal\Desktop\C\DAA_assignment\3_SortingAlgo\O(nlogn)>gcc SortingAlgo.c
quick sort:
Input size        time taken
-----------------------------------
100               14400.00 ns
1000              148000.00 ns
10000             2473000.00 ns
100000            52151200.00 ns
1000000           2186975900.00 ns
-------------------------------------------------------------------------------
merge sort:
Input size        time taken
-----------------------------------
100               17100.00 ns
1000              226000.00 ns
10000             2725500.00 ns
100000            29087700.00 ns
1000000           311189000.00 ns
-------------------------------------------------------------------------------
Heap sort:
1000              258400.00 ns
10000             3296000.00 ns
100000            51551600.00 ns
1000000           647639800.00 ns
```

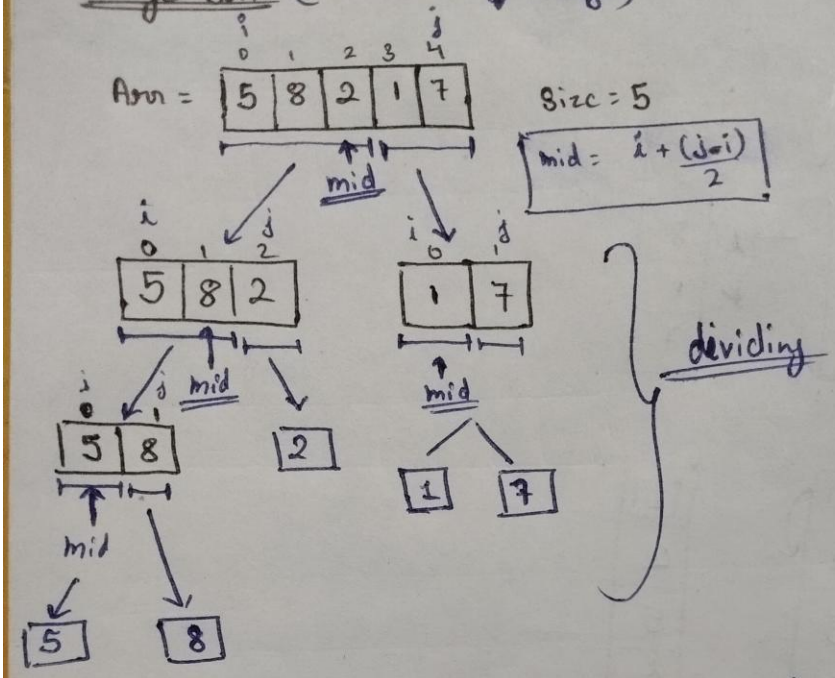
Graph: the graph compares the performance of Quick Sort, Merge Sort, and Heap Sort across different array sizes.

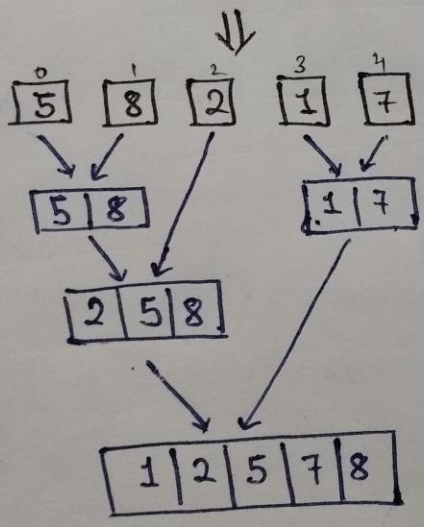


Dry Run: Dry run on array {5,8,2,1,7} using Quick Sort, Merge Sort and Heap Sort.



* Merge Sort (Divide & conquer algo)





Time Complexity = O(n log n)

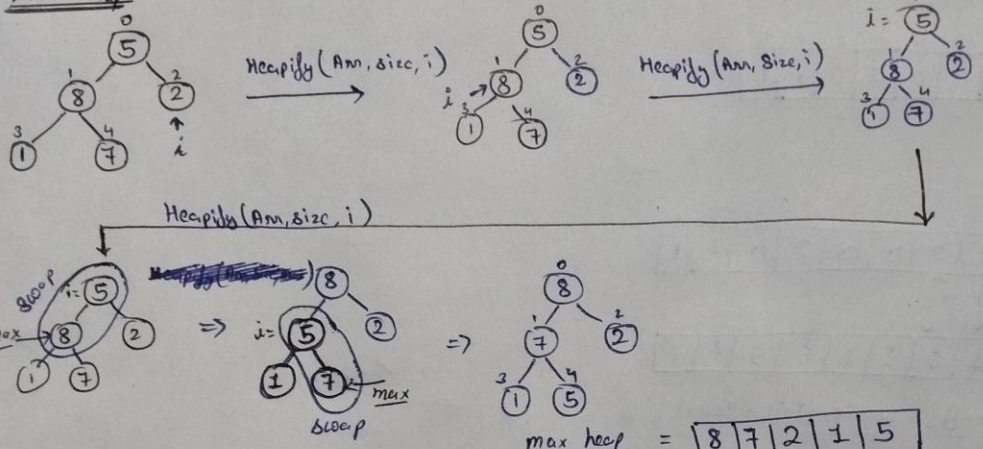
* Heap Sort using max Heap

Arr = | 0:5 | 1:8 | 2:2 | 3:1 | 4:7 | size = 5

leaf nodes

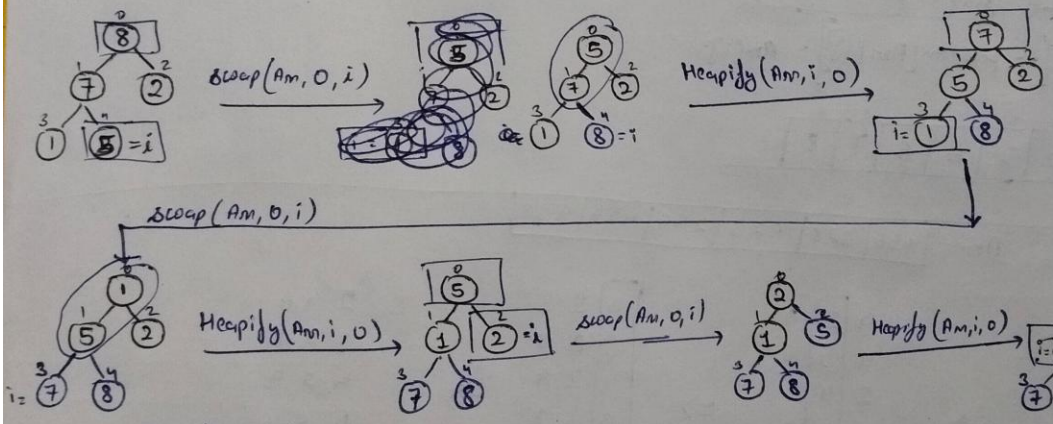
leaf node = $\lfloor \frac{size}{2} \rfloor = 2$

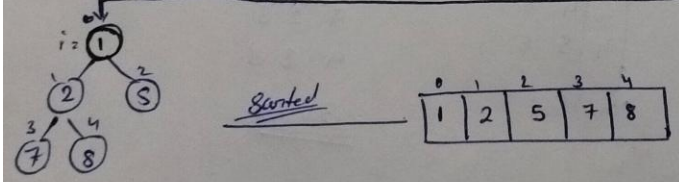
⇒ Build Heap



max heap = | 8 | 7 | 2 | 1 | 5 |

* Heap Sort





Sorted | 0:1 | 1:2 | 2:5 | 3:7 | 4:8 |

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

```
* Heapify (Array, size, start_index)
* Build Heap
    for( i = size/2 to 0){ Heapify(Arr, size, i); }
* HeapSort
    for( i = size-1 to 1){
        swap(Arr, 0, i)
        Heapify(Arr, i, 0)
    }
```

3) Sorting Algorithms of O(n)

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<windows.h>
#include<time.h>

typedef struct Bucket{
   float val;
   struct Bucket *next;
} Bucket;

int getRandom(int min, int max);
int* GenArr(int size);

float getMinF(float *Arr, int size);
float getMaxF(float *Arr, int size);
int getMax(int *Arr, int size);

void printArrayF(float *arr, int size);
void printArray(int *arr, int size);


void CountSort(int *Arr, int size);
void R_CountSort(int *Arr, int size, int exp);

void RadixSort(int *Arr, int size);

Bucket *createNode(float val);
void InsertInBucket(Bucket** head, float val);
void BucketSort(float *Arr, int size);


/*************************************************************************/
/*                                Main Sort                              */
/*************************************************************************/

int main(){
   LARGE_INTEGER freq, start, end;
   QueryPerformanceFrequency(&freq);
   srand(time(0));


   int Sizes[] = {100, 1000, 10000, 100000, 1000000};
   int n = sizeof(Sizes)/sizeof(Sizes[0]);
```

```
    printf("Counting Sort: \n");
    printf("Input size \t   time taken\n");
    printf("-----------------------------------\n");
    for(int i=0; i<n; i++){
        int size = Sizes[i];
        int *Arr = GenArr(size);

        QueryPerformanceCounter(&start);
        CountSort(Arr, size);
        QueryPerformanceCounter(&end);
        double time_taken = (double)(end.QuadPart - start.QuadPart) * 1e9 / freq.QuadPart;
        printf("%-18d  %-5.2lf ns\n", size, time_taken);

        free(Arr);
    }
    printf("-------------------------------------------------------------------------------\n");
    printf("Radix Sort: \n");
    printf("Input size \t   time taken\n");
    printf("-----------------------------------\n");
    for(int i=0; i<n; i++){
        int size = Sizes[i];
        int *Arr = GenArr(size);
        QueryPerformanceCounter(&start);
        RadixSort(Arr,size);
        QueryPerformanceCounter(&end);
        double time_taken = (double)(end.QuadPart - start.QuadPart) * 1e9 / freq.QuadPart;
        printf("%-18d  %-5.2lf ns\n", size, time_taken);
        free(Arr);
    }
    printf("-------------------------------------------------------------------------------\n");
    printf("Bucket Sort: \n");
    printf("Input size \t   time taken\n");
    printf("-----------------------------------\n");
    for(int i=0; i<n; i++){
        int size = Sizes[i];
        float min = 1.0f, max = 999.0f;
        float* Arr = (float*)malloc(sizeof(float) * size);
        for (int i = 0; i < size; i++){
            Arr[i] = min + (float)rand() * (max - min);
        }
        QueryPerformanceCounter(&start);
        BucketSort(Arr, size);
        QueryPerformanceCounter(&end);
        double time_taken = (double)(end.QuadPart - start.QuadPart) * 1e9 / freq.QuadPart;
        printf("%-18d  %-5.2lf ns\n", size, time_taken);
        free(Arr);
    }
}
```

```
/*****************************************************************************/
 /*                                  Counting Sort                            */
/*****************************************************************************/

void CountSort(int *Arr, int size){
   int max = getMax(Arr, size);

   int *count = (int*)calloc(max + 1, sizeof(int));
   if(count == NULL){
      printf("Memory allocation failed\n");
      return;
   }

   for(int i=0; i < size; i++){
      count[Arr[i]]++;
   }

   for(int i=1; i <= max; i++){
      count[i] += count[i-1];
   }

   int *output = (int*)calloc(size, sizeof(int));
   if(output == NULL){
      printf("Memory allocation failed\n");
      return;
   }

   for(int i = size-1; i >= 0; i--){
      output[--count[Arr[i]]] = Arr[i];
   }

   for(int i = 0; i < size; i++){
      Arr[i] = output[i];
   }

   free(count);
   free(output);

}
```

```
/*****************************************************************************/
/*                                  Redix Sort                               */
/*****************************************************************************/

void R_CountSort(int *Arr, int size, int exp){
   int count[10];
   for (int i = 0; i < 10; i++) {
      count[i] = 0;
   }

   for(int i=0; i < size; i++){
      count[((Arr[i] / exp) % 10 )]++;
   }

   for(int i=1; i < 10; i++){
      count[i] += count[i-1];
   }

   int *output = (int*)calloc(size, sizeof(int));
   if(output == NULL){
      printf("Memory allocation failed\n");
      return;
   }

   for(int i = size-1; i >= 0; i--){
      output[--count[((Arr[i] / exp) % 10 )]] = Arr[i];
   }

   for(int i = 0; i < size; i++){
      Arr[i] = output[i];
   }

   free(output);
}

void RadixSort(int *Arr, int size){
   int max = getMax(Arr, size);
   for(int exp = 1; max / exp > 0; exp*=10){
      R_CountSort(Arr, size, exp);
   }

}
```

```
/*************************************************************************/
/*                                Bucket Sort                            */
/*************************************************************************/

Bucket *createNode(float val){
   Bucket *node = (Bucket*)malloc(sizeof(Bucket));
   node->val = val;
   node->next = NULL;

   return node;
}

void InsertInBucket(Bucket** head, float val){
   Bucket *node = createNode(val);
   if((*head) == NULL || (*head)->val >= val){
      node->next = *head;
      *head = node;
      return;
   }

   Bucket *cur = *head;
   while(cur->next != NULL && cur->next->val < val){
      cur = cur->next;
   }
   node->next = cur->next;
   cur->next = node;
}

void BucketSort(float *Arr, int size){
   Bucket **buckets = (Bucket**)calloc(size, sizeof(Bucket*));
   float min = getMinF(Arr, size);
   float max = getMaxF(Arr, size);
   for(int i=0; i<size; i++){
      int index = (int)((Arr[i] - min) / (max - min) * (size - 1));
      InsertInBucket(&buckets[index], Arr[i]);
   }
   int j = 0;
   for(int i=0; i<size; i++){
      Bucket *temp = buckets[i];
      while(temp != NULL && j < size){
         Arr[j++] = temp->val;
         Bucket *node = temp;
         temp = temp->next;
         free(node);
      }
   }
  free(buckets);
}
```

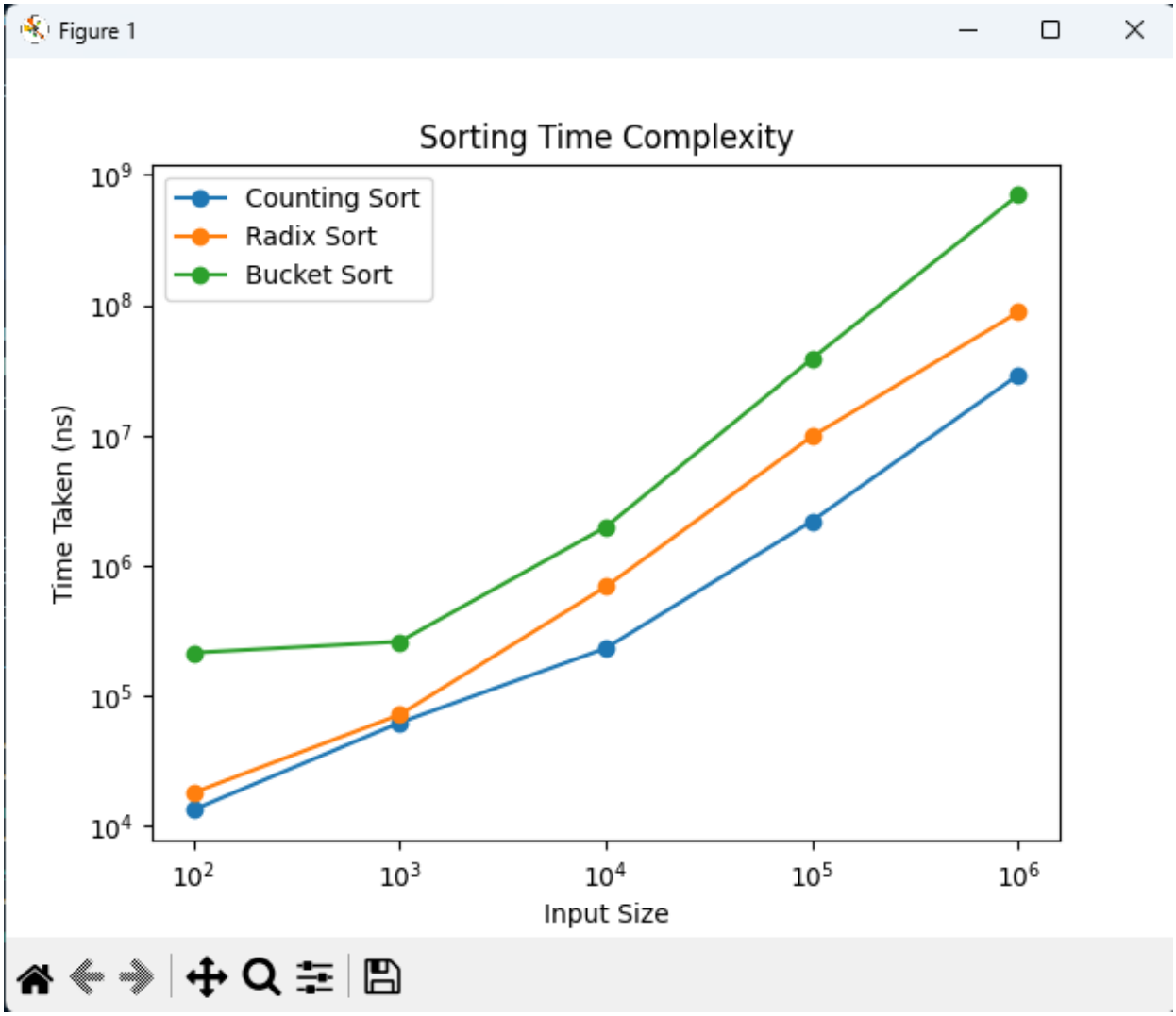
```
/*****************************************************************************/
/*                                Helper Functions                           */
/*****************************************************************************/

int getRandom(int min, int max){
   return min + rand()%(max-min);
}

int* GenArr(int size){
   int min = 1, max = 999;
   int* Arr = (int*)malloc(sizeof(int) * size);
   if(Arr == NULL){
      printf("Memory Allocation Error\n");
      exit(1);
   }
   for (int i = 0; i < size; i++) {
      Arr[i] = getRandom(min, max);
   }

   return Arr;
}

void printArray(int *arr, int size){
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}

int getMax(int *Arr, int size){
   int max = Arr[0];
   for(int i= 1; i < size; i++){
      if(max < Arr[i]) max = Arr[i];
   }

   return max;
}

float getMaxF(float *Arr, int size){
   float max = Arr[0];
   for(int i=0; i<size; i++){
      if(max < Arr[i]) max = Arr[i];
   }
   return max;
}

float getMinF(float *Arr, int size){
   float min = Arr[0];
```

```
    for(int i=0; i<size; i++){
        if(min > Arr[i]) min = Arr[i];
    }
    return min;
}

void printArrayF(float *arr, int size){
    for (int i = 0; i < size; i++) {
        printf("%f ", arr[i]);
    }
    printf("\n");
}
```

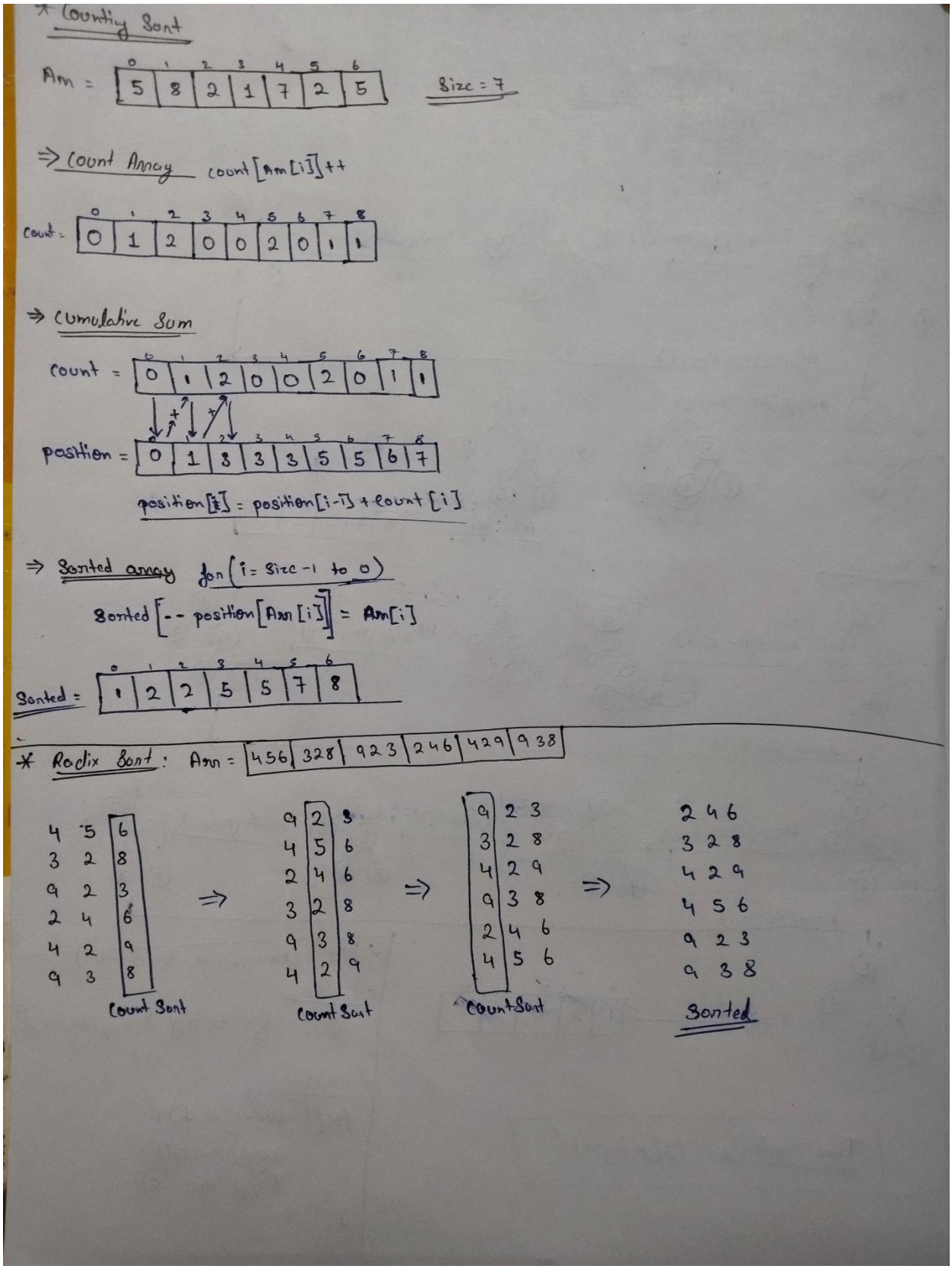
OUTPUT:

```
c:\Users\Ujjwal\Desktop\C\DAA_assignment\3_SortingAlgo\O(n)>gcc Algo.c -Wl,--stack,268
Algo.exe && Algo.exe
Counting Sort:
Input size         time taken
-----------------------------------
100                 17000.00 ns
1000                91700.00 ns
10000               225800.00 ns
100000              2308600.00 ns
1000000             27414700.00 ns
-------------------------------------------------------------------------------
Radix Sort:
Input size         time taken
-----------------------------------
100                 16700.00 ns
1000                95400.00 ns
10000               941400.00 ns
100000              9562700.00 ns
1000000             89344100.00 ns
-------------------------------------------------------------------------------
Bucket Sort:
Input size         time taken
-----------------------------------
100                 216300.00 ns
1000                282200.00 ns
10000               2554200.00 ns
100000              38131400.00 ns
1000000             585439200.00 ns
```

Graph: the graph compares the performance of Counting Sort, Radix Sort, and Bucket Sort across different array sizes.



Dry Run: Dry run on array {5,8,2,1,7,2,5} using counting Sort, Radix Sort and Bucket Sort.



* Bucket Sort

	0	1	2	3	4	5	6
Arr =	0.43	0.17	0.13	0.58	0.35	0.78	0.86

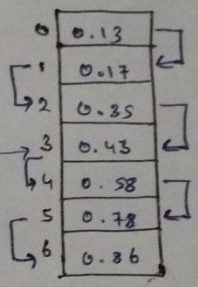
Size = 7

⇒ Bucket_index = size × Arr[i]

Buckets [bucket_index] = Arr[i]

→ 0.43 × 7 = 3.01 → 3

→ 0.17 × 7 = 1.19 → 1



Buckets

Combining the Buckets

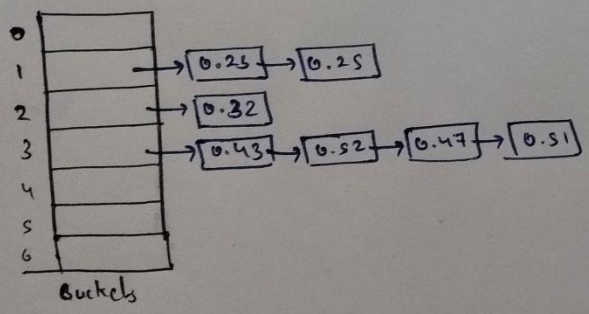
0.13	0.17	0.35	0.43	0.58	0.78	0.86

Sorted

Ex

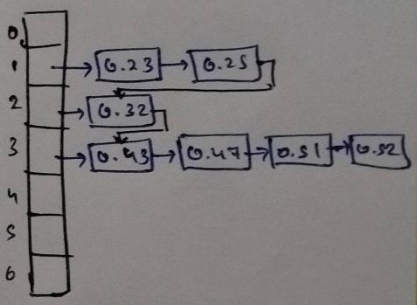
	0	1	2	3	4	5	6
Arr =	0.43	0.32	0.23	0.52	0.25	0.47	0.51

index = size × Arr[i]



Buckets

Sort each bucket using any sorting Algo



combine the buckets

0	1	2	3	4	5	6
0.25	0.25	0.32	0.43	0.47	0.51	0.52