Name: Harsh Jaiswal

Enrollment number: BT22CSH013

Branch: CSH(HCI & GT)

Semester: 3

QUESTION 1

```c
#include <stdio.h>

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

void countingsort(int arr[], int size, int exp)
{
    int output[size];
    int count[10] = {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];
    }

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

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

void radixsort(int arr[], int size)
{
    int max = getMax(arr, size);

    for (int exp = 1; max / exp > 0; exp *= 10)
    {
        countingsort(arr, size, exp);
    }
}

int main()
{
    int arr[10] = {136, 487, 358, 469, 570, 247, 598, 639, 205, 609};
    int size = sizeof(arr) / sizeof(arr[0]);

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

    radixsort(arr, size);

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

    return 0;
}
```

The result of the following code:

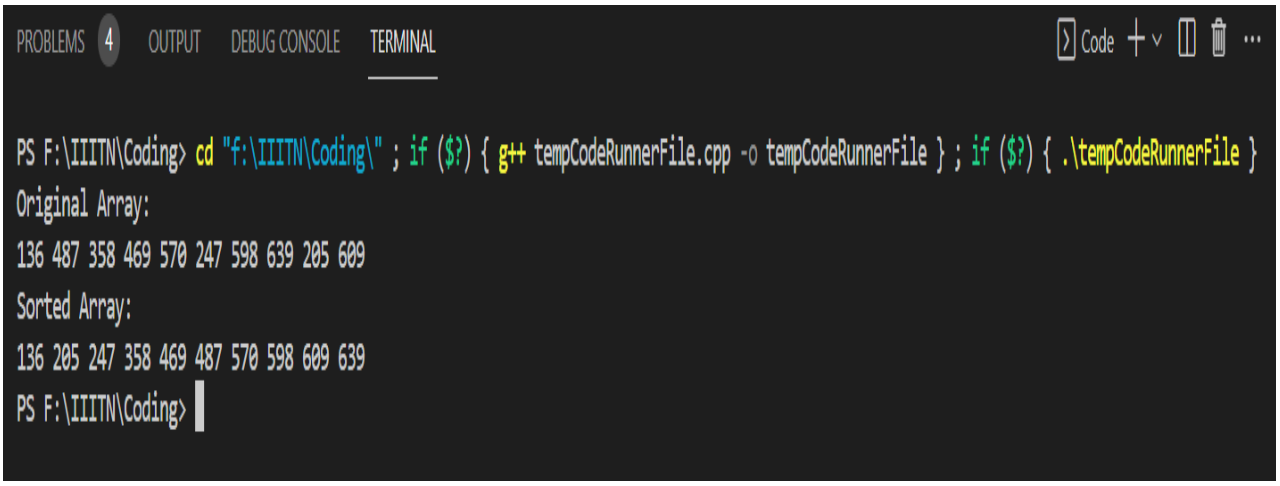
Original Array:

136 487 358 469 570 247 598 639 205 609

Sorted Array:

136 205 247 358 469 487 570 598 609 639

Screenshot:



```
PS F:\IIITN\Coding> cd "f:\IIITN\Coding\" ; if ($?) { g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Original Array:
136 487 358 469 570 247 598 639 205 609
Sorted Array:
136 205 247 358 469 487 570 598 609 639
PS F:\IIITN\Coding>
```

TIME COMPLEXITY

The time complexity of Radix Sort is determined by both the number of digits in the largest number and the quantity of elements in the array. Assuming the maximum number has k digits and there are n elements in the array, the Counting Sort procedure will be invoked k times within the Radix Sort function. Inside this Counting Sort operation, a loop will iterate n times.

Consequently:

- In the Best Case scenario: O(k * n)
- In the Average Case scenario: O(k * n)
- In the Worst Case scenario: O(k * n)

In these expressions, the underlying essence remains consistent. The Radix Sort's performance is intrinsically linked to both the magnitude of the largest number's digits and the total number of elements in the array.

Question 2

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node
{
    int data;
    struct Node *next;
} Node;

void insert(Node **head, int data)
{
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;

    if (*head == NULL)
    {
        *head = newNode;
    }
    else
    {
        Node *current = *head;
        while (current->next != NULL)
        {
            current = current->next;
        }
        current->next = newNode;
    }
}

void printList(Node *head)
{
    Node *current = head;
    while (current != NULL)
    {
        printf("%d ", current->data);
        current = current->next;
    }
}

void clearList(Node **head)
{
    Node *current = *head;
    while (current != NULL)
    {
        Node *temp = current;
        current = current->next;
        free(temp);
    }
    *head = NULL;
}

void radixSort(int arr[], int n)
{
    int max = arr[0];
    for (int i = 1; i < n; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
        }
    }
    int digits = 0;
    while (max > 0)
    {
        digits++;
        max /= 10;
    }
    for (int exp = 1; digits > 0; exp *= 10, digits--)
    {
        Node *buckets[10] = {NULL};

        for (int i = 0; i < n; i++)
        {
            int index = (arr[i] / exp) % 10;
            insert(&buckets[index], arr[i]);
        }

        printf("Sorted Array (Phase %d): \n", exp);
        int index = 0;
        for (int i = 0; i < 10; i++)
        {
            Node *current = buckets[i];
            while (current != NULL)
            {
                arr[index++] = current->data;
                Node *temp = current;
                current = current->next;
                free(temp);
            }
        }
        for (int i = 0; i < n; i++)
        {
            printf("%d ", arr[i]);
        }
        printf("\n");
    }

    printf("Final Sorted Array: \n");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
}

int main()
{
    int n;
    printf("Enter the number of elements: \n");
    scanf("%d", &n);

    int arr[n];
    printf("Enter the array elements:\n");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }

    radixSort(arr, n);

    return 0;
}
```

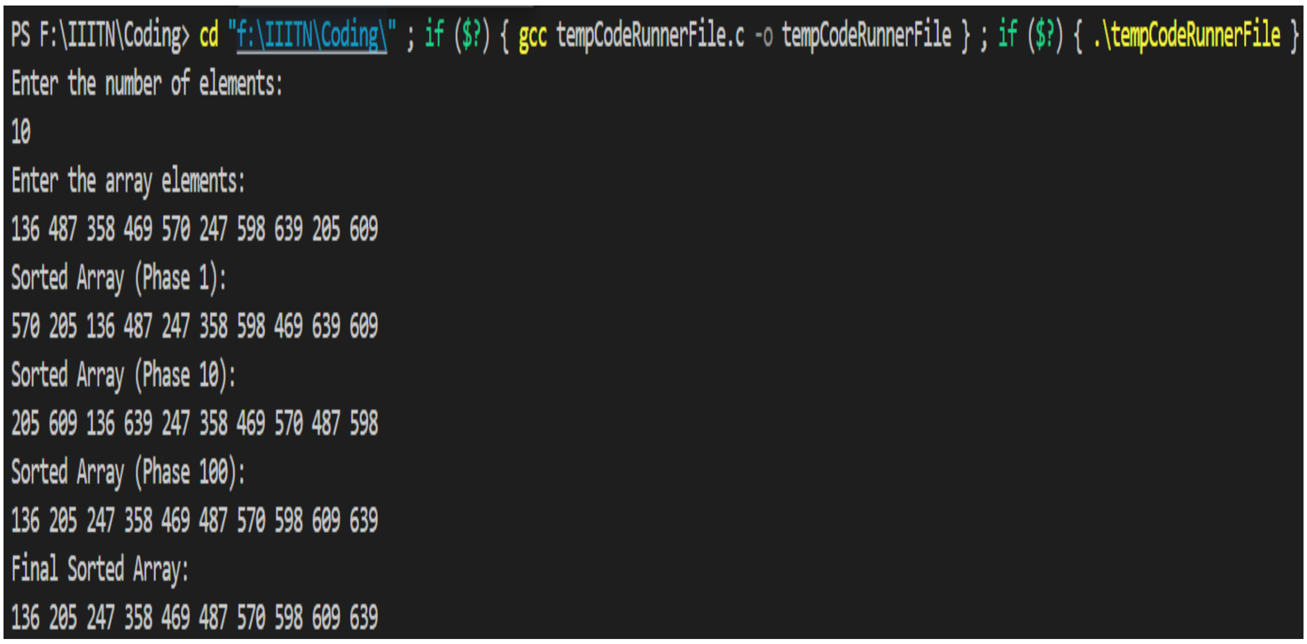
The output for the following code:--

```
Enter the number of elements:

10
```

```
Enter the array elements:
136 487 358 469 570 247 598 639 205 609
Sorted Array (Phase 1):
570 205 136 487 247 358 598 469 639 609
Sorted Array (Phase 10):
205 609 136 639 247 358 469 570 487 598
Sorted Array (Phase 100):
136 205 247 358 469 487 570 598 609 639
Final Sorted Array:
136 205 247 358 469 487 570 598 609 639
```

Screenshot:



```
PS F:\IIITN\Coding> cd "f:\IIITN\Coding\" ; if ($?) { gcc tempCodeRunnerFile.c -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Enter the number of elements:
10
Enter the array elements:
136 487 358 469 570 247 598 639 205 609
Sorted Array (Phase 1):
570 205 136 487 247 358 598 469 639 609
Sorted Array (Phase 10):
205 609 136 639 247 358 469 570 487 598
Sorted Array (Phase 100):
136 205 247 358 469 487 570 598 609 639
Final Sorted Array:
136 205 247 358 469 487 570 598 609 639
```

TIME COMPLEXITY:

The time complexity analysis for this program, which employs linked lists to implement Radix Sort, is outlined as follows:

- **Best Case:** The time complexity of the linked-list-based Radix Sort persists at O(k * n) in the best-case scenario. This is due to the requirement of iterating through each of the **k** digits for every one of the **n** elements. It's essential to highlight that in this context, 'k' represents the count of digits in the largest number.
- **Average Case:** Similarly, the average-case time complexity maintains O(k * n). This is because, on average, the algorithm must still traverse through all **k** digits for each of the **n** elements, resulting in the same O(k * n) time complexity.
- **Worst Case:** The worst-case time complexity, once again, remains O(k * n). This scenario arises when the distribution of digits maximizes the number of iterations through the digits. In every iteration, all **n** elements need to be processed, categorized into linked lists, and subsequently reassembled. The count of iterations is fundamentally determined by the quantity of digits in the largest number.

In all of these cases, the underlying meaning remains consistent: the time complexity of this linked-list-based Radix Sort is inherently bound by the number of digits in the largest number and the total number of elements in the array.