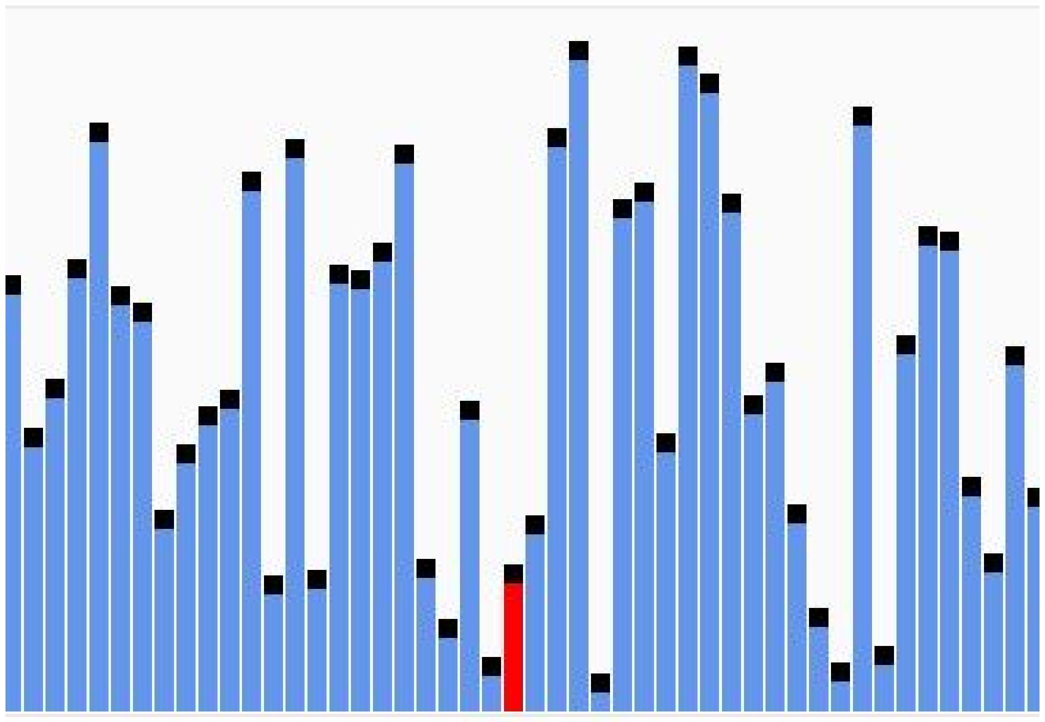
CSN-212: Design & Analysis of Algorithms

CODING PROJECT 1
SORTING ALGORITHMS IMPLEMENTATIONS



Introduction

This project aims to implement all the basic sorting algorithms in the course.

The list includes:

1. Insertion Sort
2. Selection Sort
3. Bubble Sort
4. Shell Sort
5. Merge Sort
6. Quick Sort

7. Heap Sort
8. Count Sort
9. Radix Sort
10. Bucket Sort

Short Descriptions

1.Insertion Sort

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.
Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.
The resulting array after k iterations has the property where the first k + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:
becomes
with each element greater than x copied to the right as it is compared against x.

2. Selection Sort

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

3. Bubble Sort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sort order but may occasionally have some out-of-order elements nearly in position.

4. Shell Sort

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort if smaller value is very far right and have to move to far left.
This algorithm uses insertion sort on widely spread elements first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval.

5. Merge Sort

A merge sort works as follows:
Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list

6. Quick Sort

Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.
The steps are:
Pick an element, called a pivot, from the array.
Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

7. Heap Sort

In computer science, heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. The improvement consists of the use of a heap data structure rather than a linear-time search to find the maximum.

8. Counting Sort

Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items. However, it is often used as a subroutine in another sorting algorithm, radix sort, that can handle larger keys more efficiently.

9. Radix Sort

Each key is first figuratively dropped into one level of buckets corresponding to the value of the rightmost digit. Each bucket preserves the original order of the keys as the keys are dropped into the bucket. There is a one-to-one correspondence between the buckets and the values that can be represented by the rightmost digit. Then, the process repeats with the next neighbouring more significant digit until there are no more digits to process. In other words:
Take the least significant digit (or group of bits, both being examples of radices) of each key.
Group the keys based on that digit, but otherwise keep the original order of keys. (This is what makes the LSD radix sort a stable sort.)
Repeat the grouping process with each more significant digit.
The sort in step 2 is usually done using bucket sort or counting sort, which are efficient in this case since there are usually only a small number of digits.

10. Bucket Sort

Bucket sort, or bin sort, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a distribution sort, and is a cousin of radix sortin the most to least significant digit flavour. Bucket sort is a generalization of pigeonhole sort. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm. The computational complexity estimates involve the number of buckets.

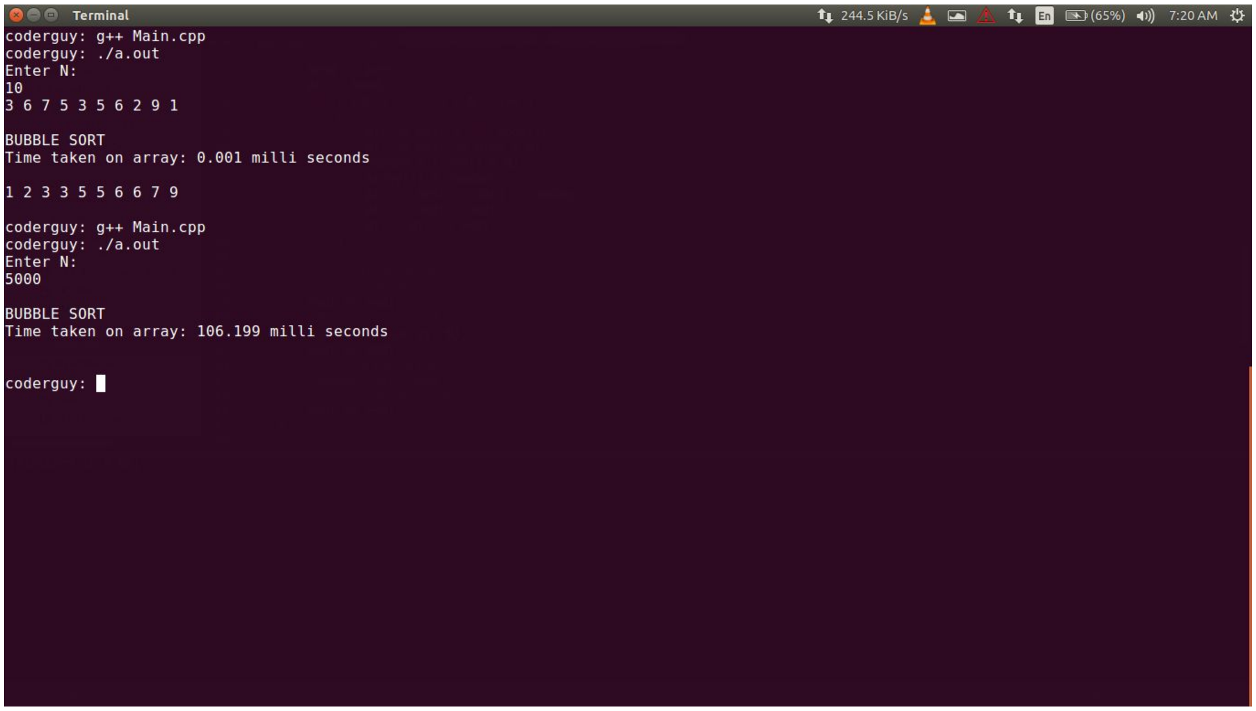
Main File (Code)

```
// GroupID-8 (14114002_14114068) - Abhishek Jaisingh & Tarun Kumar

// Date: February 4, 2016

// Main.cpp - Top Level File.


#include <bits/stdc++.h>

//#include "algorithms/selection_sort.cpp"

//#include "algorithms/counting_sort.cpp"

//#include "algorithms/shell_sort.cpp"

//#include "algorithms/quick_sort.cpp"

//#include "algorithms/bucket_sort.cpp"

//#include "algorithms/bubble_sort.cpp"

//#include "algorithms/insertion_sort.cpp"

//#include "algorithms/radix_sort.cpp"

#include "algorithms/merge_sort.cpp"

//#include "algorithms/heap_sort.cpp"

using namespace std;

void printArray(int* a, int n)

{
	for ( int i = 0 ; i < n ; i++ )
		cout << a[i] << " ";
	cout << endl;
}
void printLinkedList(node* ptr)
{
	if ( ptr == NULL )
		return;
	ptr = ptr -> next;
	while ( ptr != NULL )
		{
			cout << ptr -> data << " ";
			ptr = ptr -> next;
		}
	cout << endl;
}
int main()
{

	//srand((int)time(0));
	int N;
	int random;
	node *head, *ptr;
	cout << "Enter N:" << endl;
```

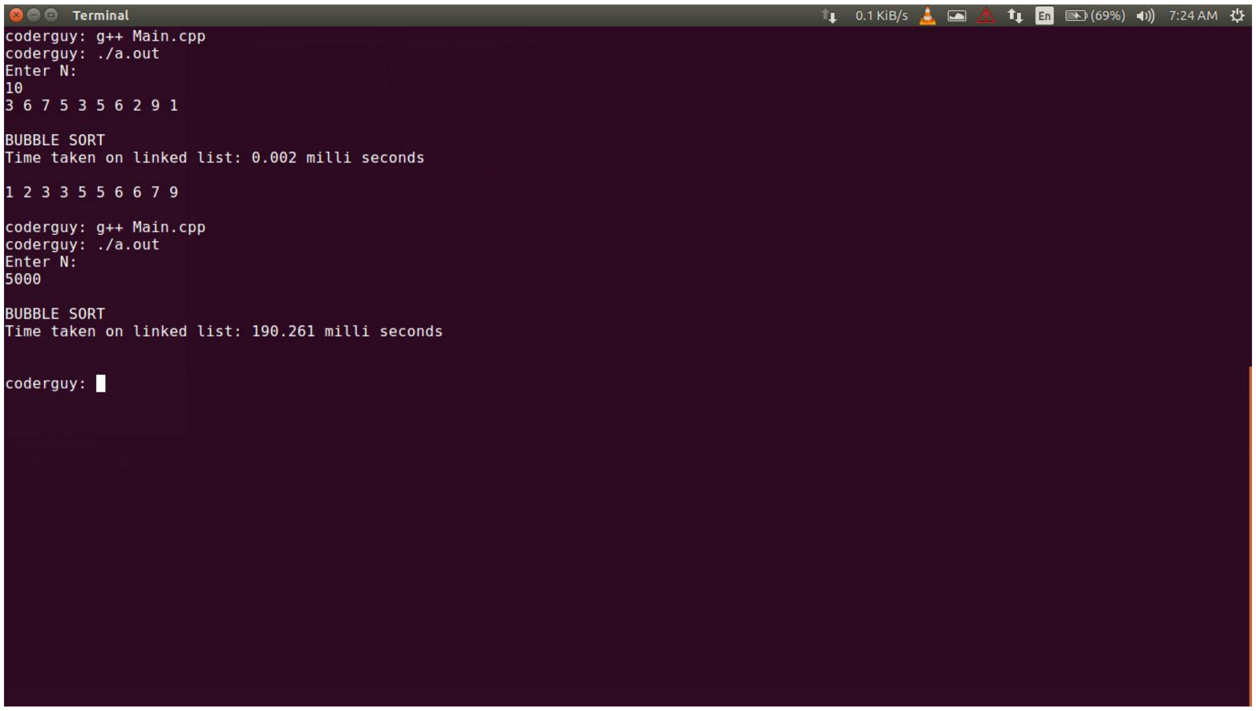
```
N = 10;

cin >> N;

int* array = new int[N];

head = new node();

head -> data = -1;

head -> prev = NULL;

ptr = head;

for ( int i = 0 ; i < N ; i++ )

       {

              ptr -> next = new node();

              ptr -> next -> prev = ptr;

              random = (rand() % N);

              array[i] = random;

              ptr -> next -> data = random;

              ptr -> next-> next = NULL;

              ptr = ptr -> next;

       }


printArray(array, N);

//printLinkedList(head);

cout << endl;

//bucket_sort(head, N);

//bubble_sort(head, N);

merge_sort(array, N);
```

```
        //heap_sort(head, N);

        cout << endl;

        printArray(array, N);

        //bucket_sort(head, N);

        //printLinkedList(head);

        cout << endl;

}
```

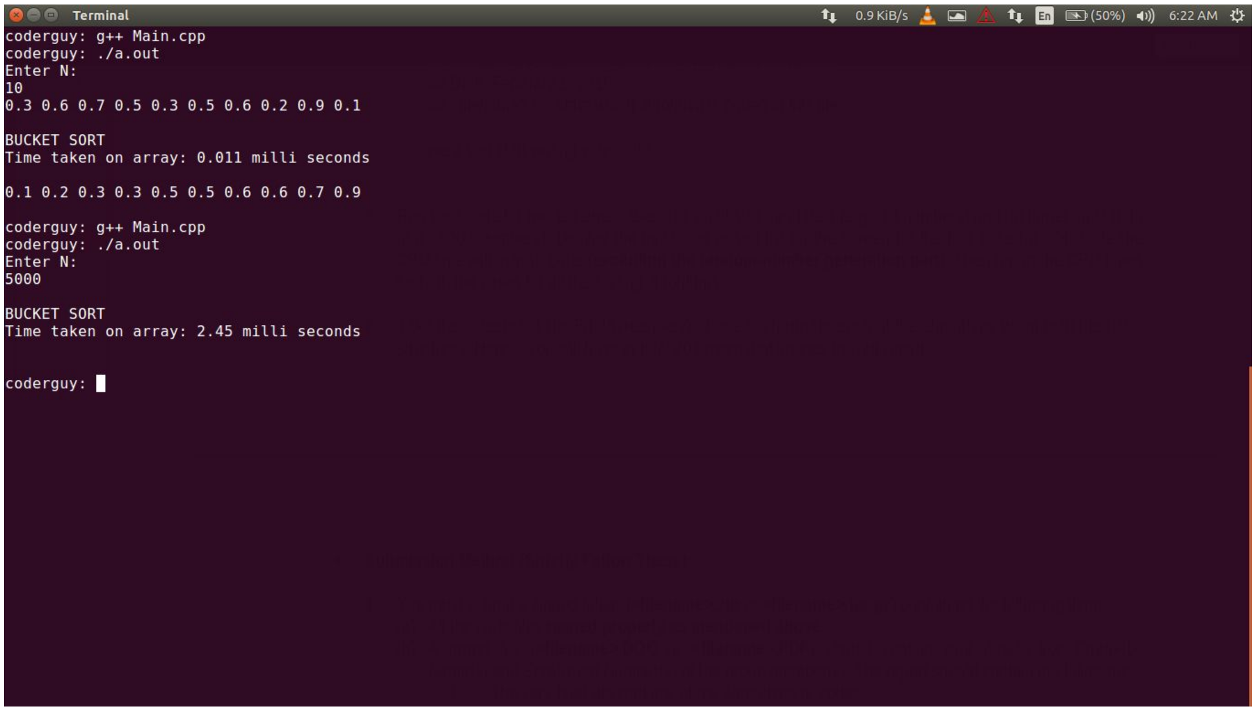
Screenshots



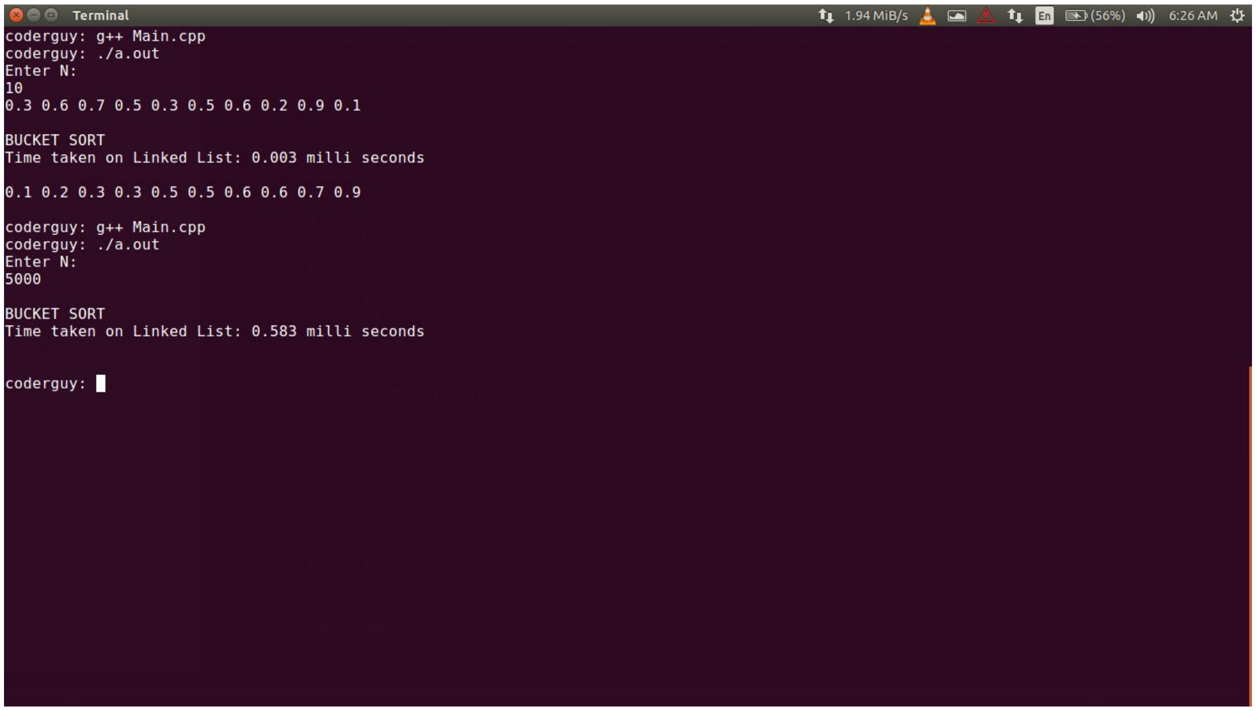
```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

BUBBLE SORT
Time taken on array: 0.001 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

BUBBLE SORT
Time taken on array: 106.199 milli seconds


coderguy: 
```



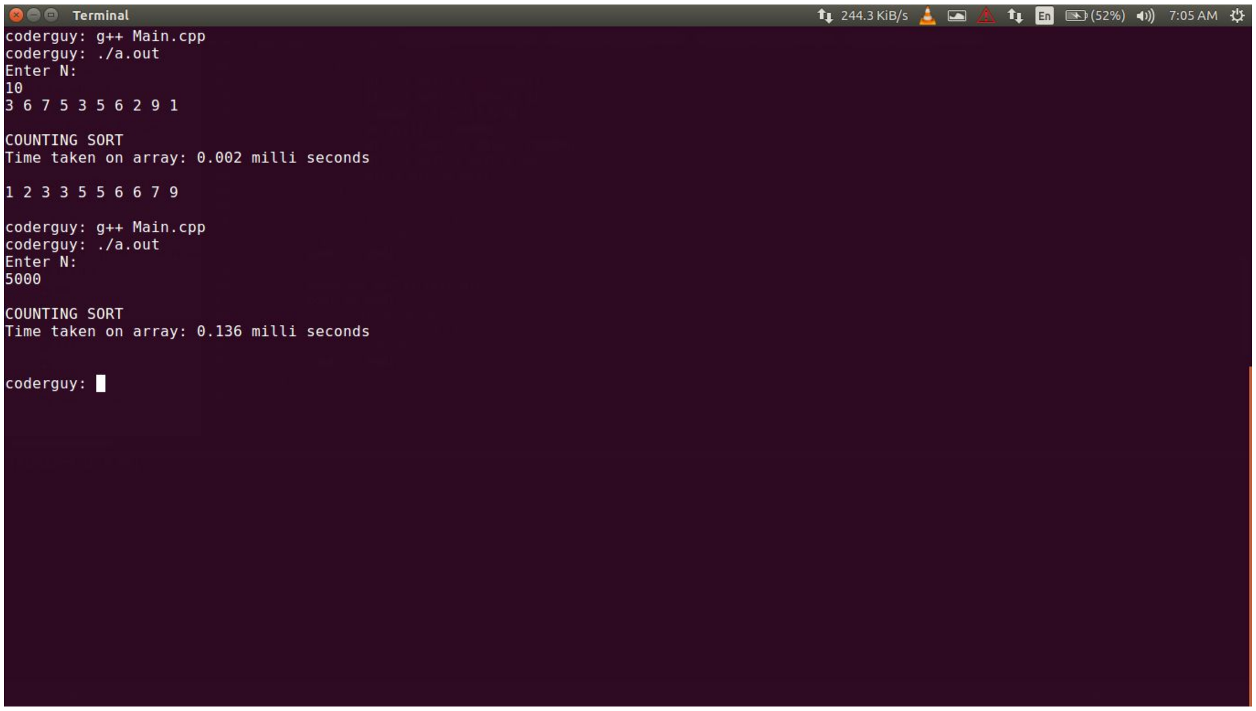
```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

BUBBLE SORT
Time taken on linked list: 0.002 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

BUBBLE SORT
Time taken on linked list: 190.261 milli seconds


coderguy: 
```



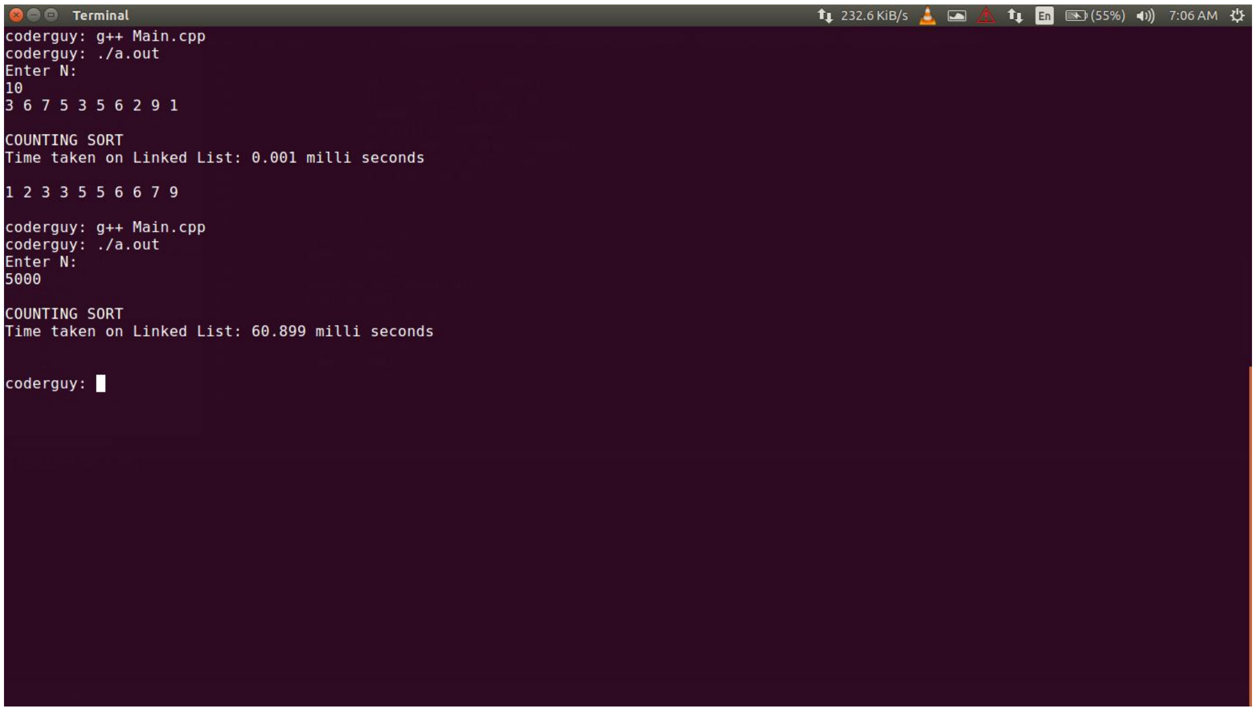
```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
0.3 0.6 0.7 0.5 0.3 0.5 0.6 0.2 0.9 0.1

BUCKET SORT
Time taken on array: 0.011 milli seconds

0.1 0.2 0.3 0.3 0.5 0.5 0.6 0.6 0.7 0.9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

BUCKET SORT
Time taken on array: 2.45 milli seconds


coderguy: 
```



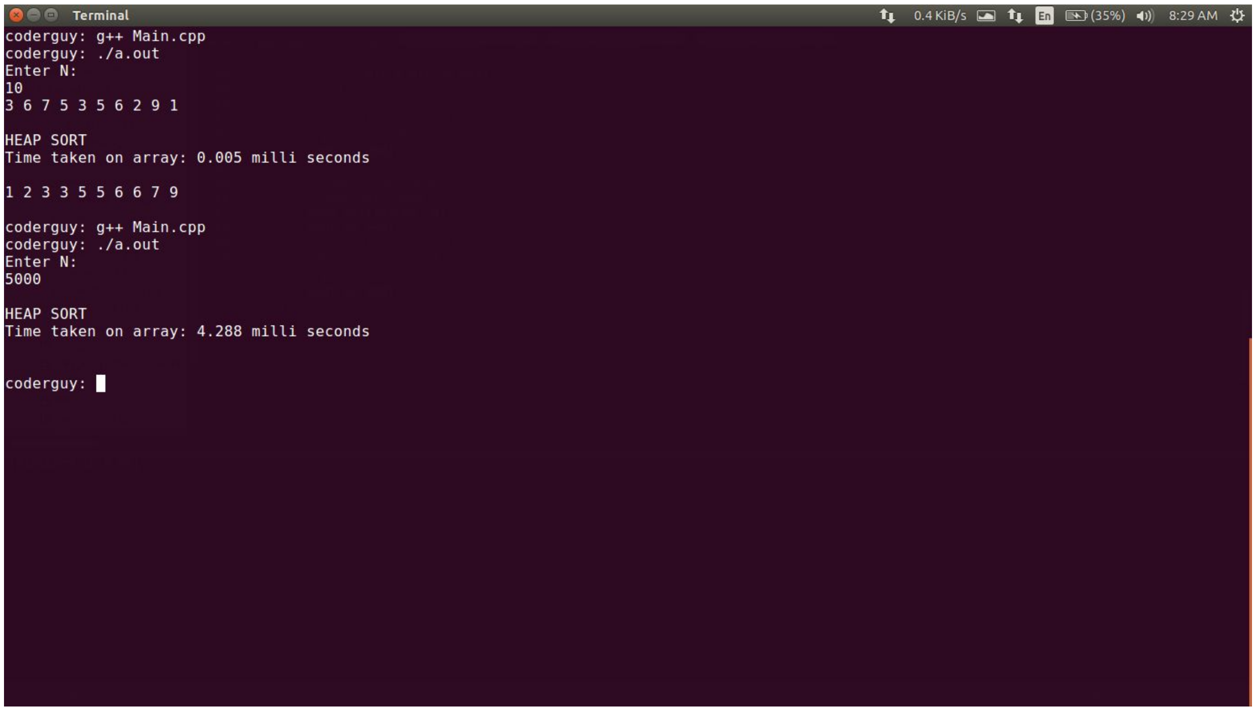
```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
0.3 0.6 0.7 0.5 0.3 0.5 0.6 0.2 0.9 0.1

BUCKET SORT
Time taken on Linked List: 0.003 milli seconds

0.1 0.2 0.3 0.3 0.5 0.5 0.6 0.6 0.7 0.9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

BUCKET SORT
Time taken on Linked List: 0.583 milli seconds


coderguy: 
```



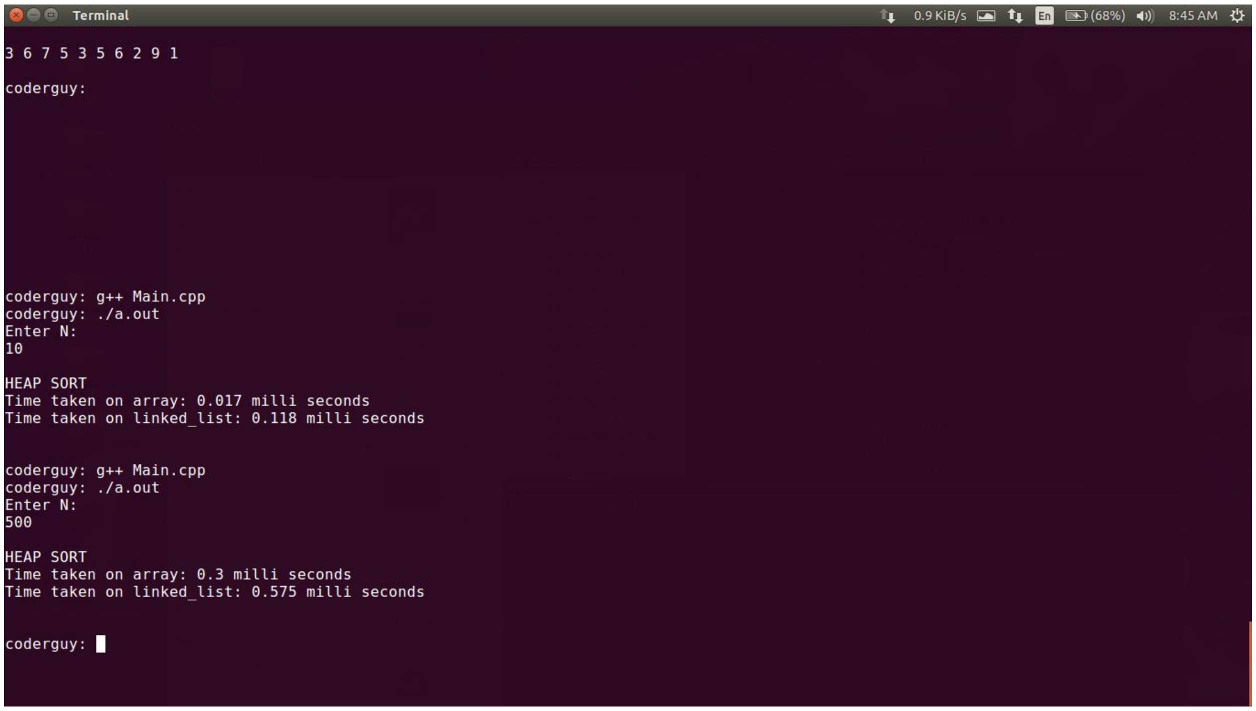
```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

COUNTING SORT
Time taken on array: 0.002 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

COUNTING SORT
Time taken on array: 0.136 milli seconds


coderguy: 
```



```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

COUNTING SORT
Time taken on Linked List: 0.001 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

COUNTING SORT
Time taken on Linked List: 60.899 milli seconds


coderguy: 
```

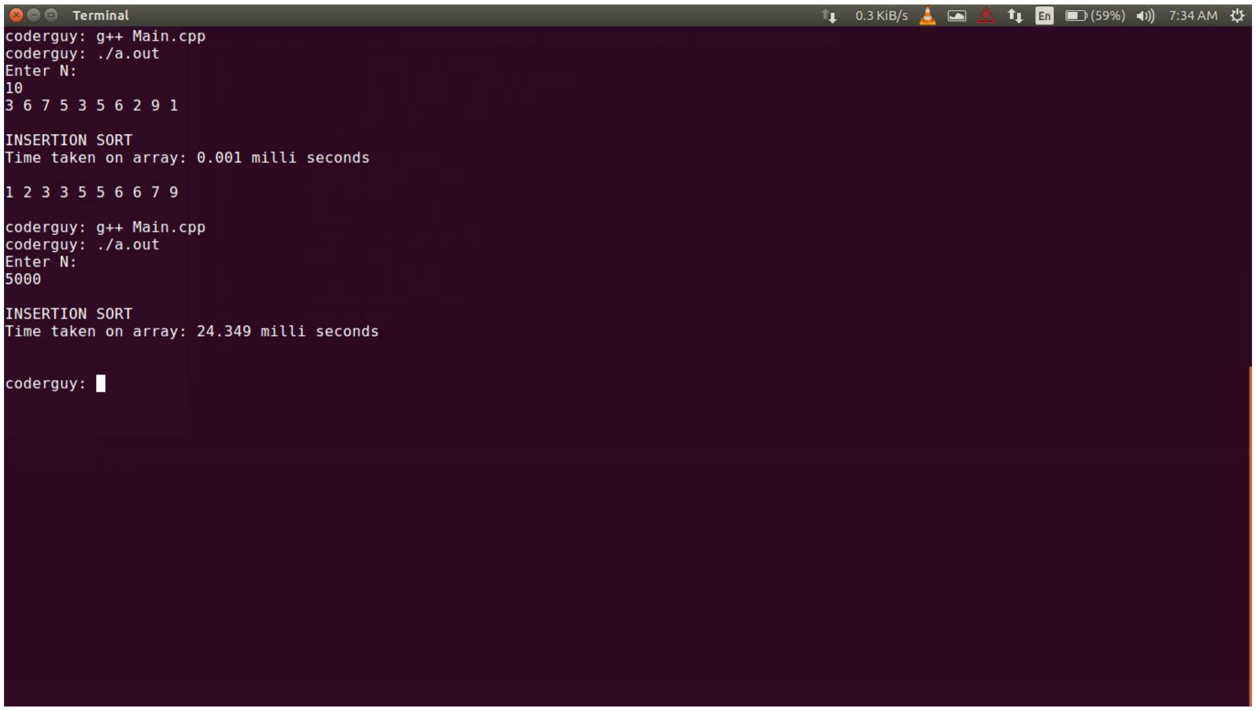


```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

HEAP SORT
Time taken on array: 0.005 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

HEAP SORT
Time taken on array: 4.288 milli seconds


coderguy: 
```



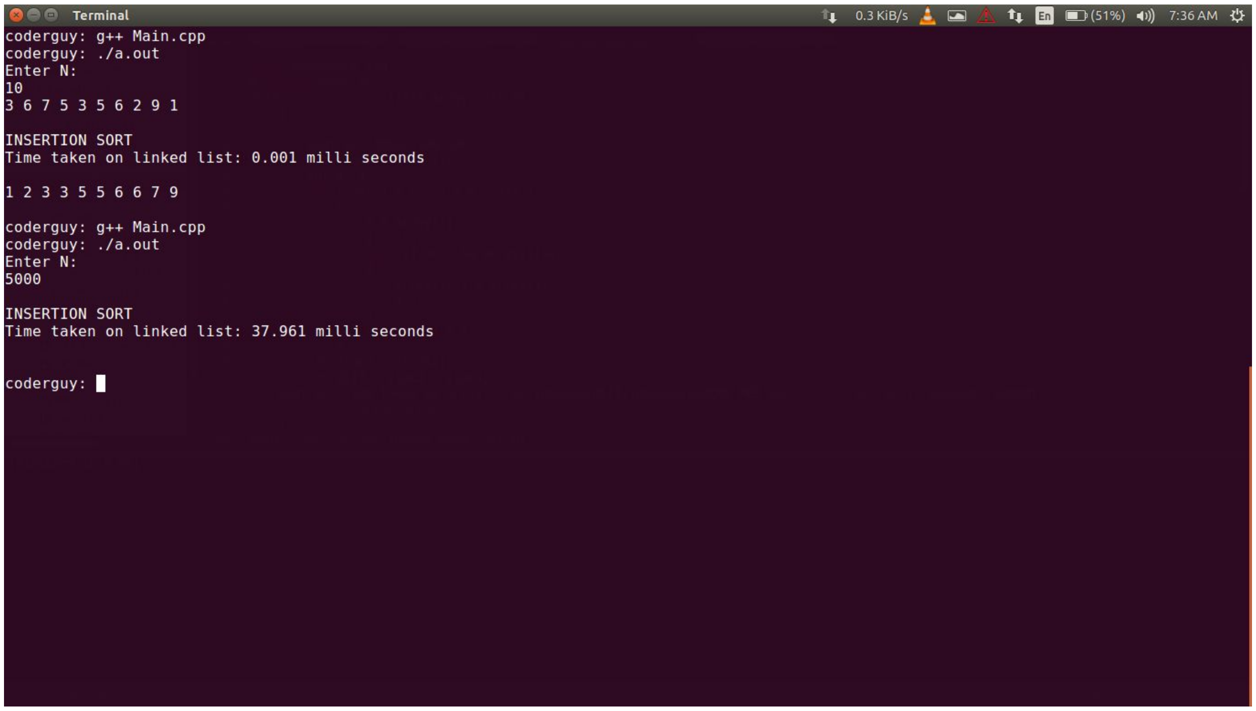
Terminal   0.9 KiB/s   En   (68%)   8:45 AM

```
3 6 7 5 3 5 6 2 9 1

coderguy:











coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10

HEAP SORT
Time taken on array: 0.017 milli seconds
Time taken on linked_list: 0.118 milli seconds


coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
500

HEAP SORT
Time taken on array: 0.3 milli seconds
Time taken on linked_list: 0.575 milli seconds


coderguy: █
```



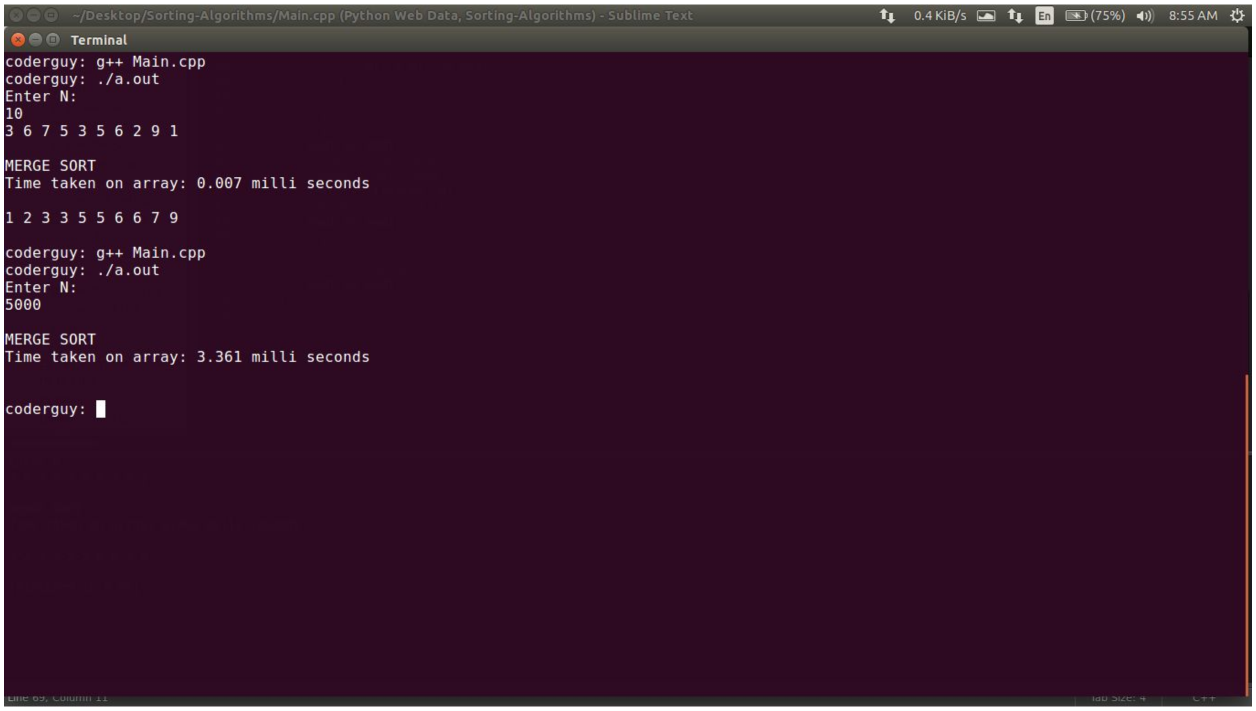
Terminal   0.3 KiB/s   En   (59%)   7:34 AM

```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

INSERTION SORT
Time taken on array: 0.001 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

INSERTION SORT
Time taken on array: 24.349 milli seconds


coderguy: █
```



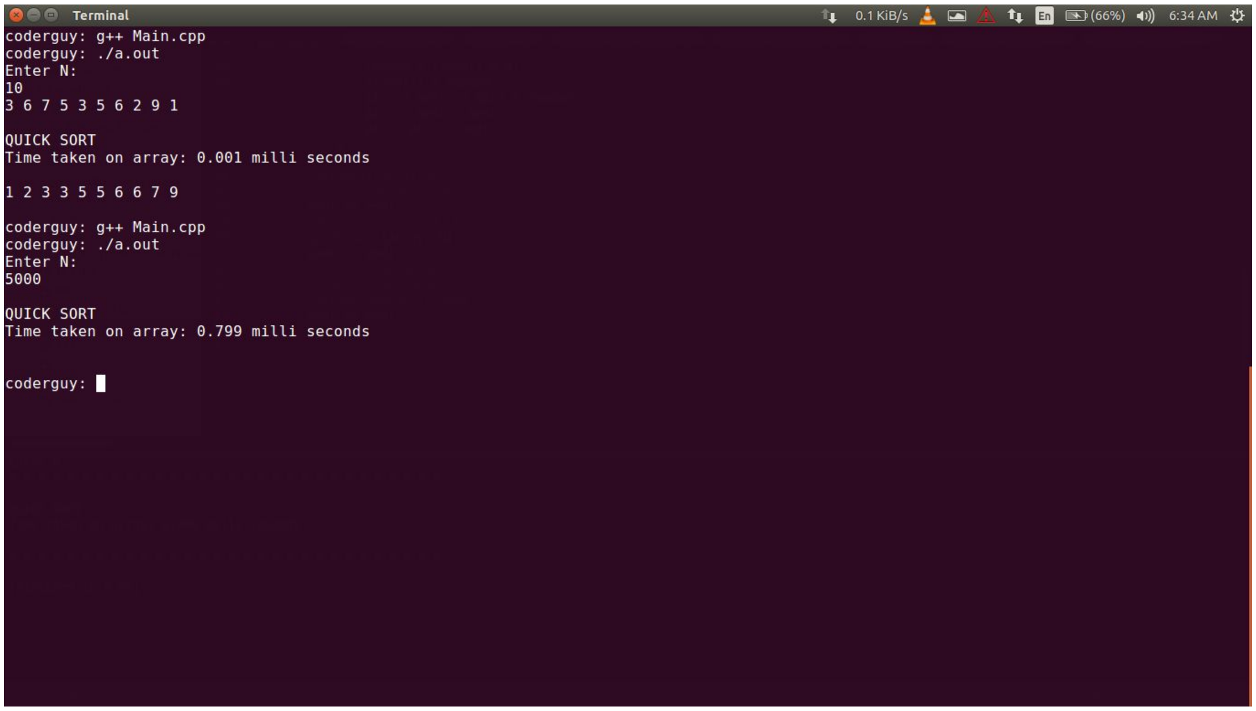
Terminal

```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

INSERTION SORT
Time taken on linked list: 0.001 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

INSERTION SORT
Time taken on linked list: 37.961 milli seconds


coderguy:
```

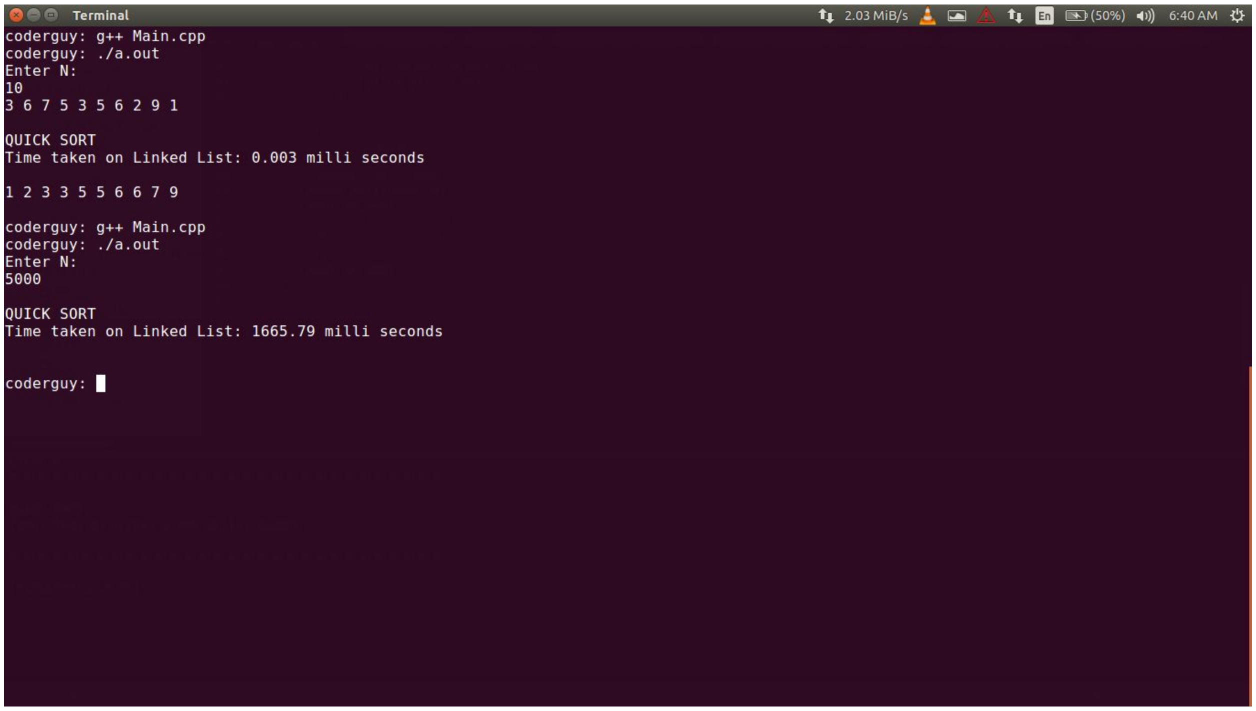


Terminal

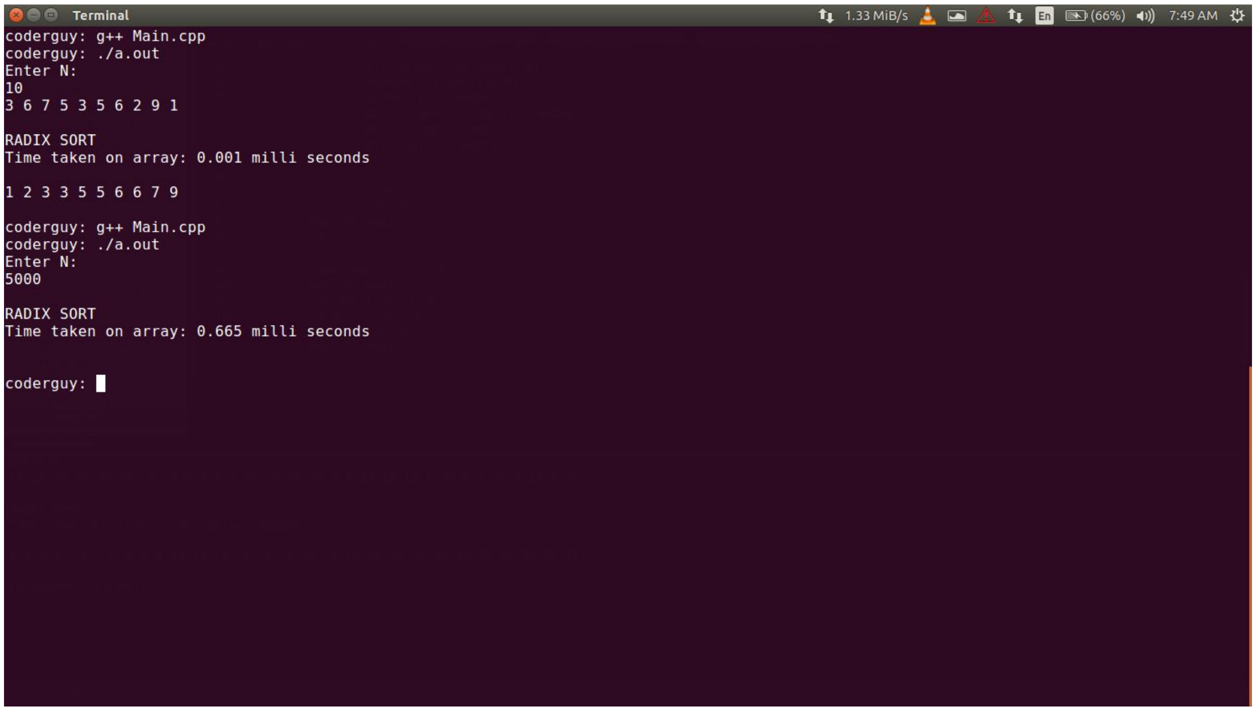
```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

MERGE SORT
Time taken on array: 0.007 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

MERGE SORT
Time taken on array: 3.361 milli seconds


coderguy:
```



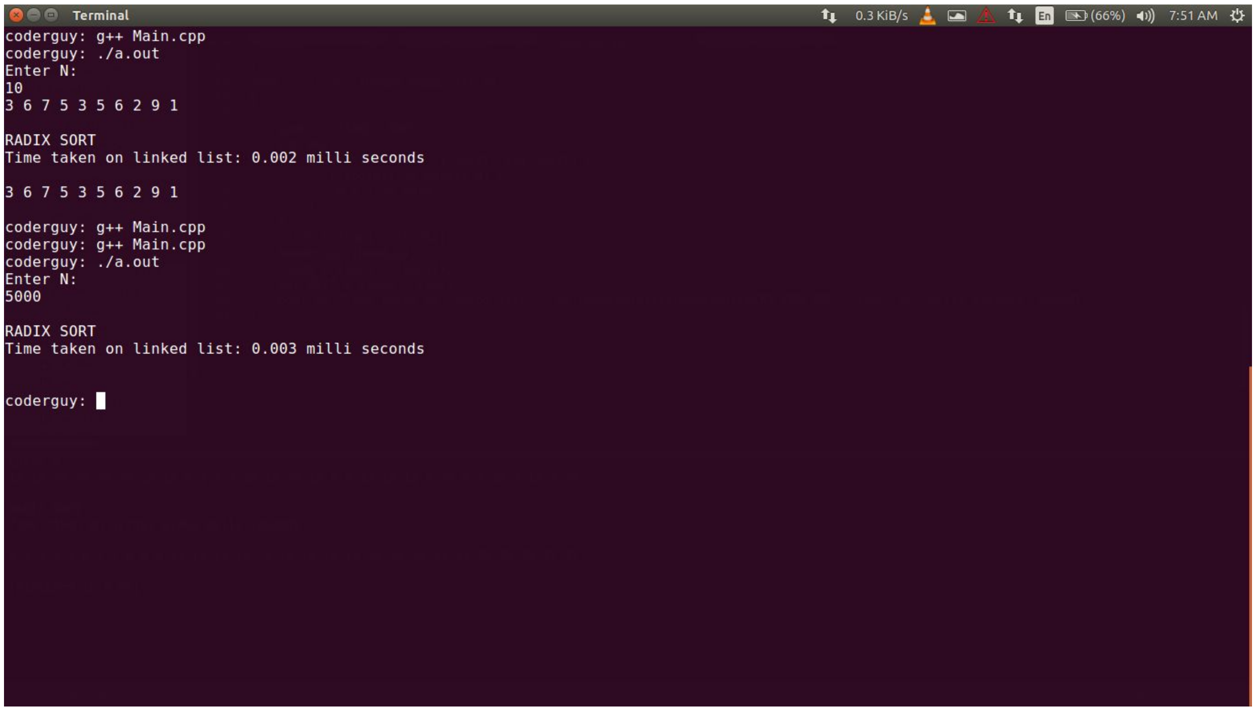
```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

QUICK SORT
Time taken on array: 0.001 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

QUICK SORT
Time taken on array: 0.799 milli seconds


coderguy: 
```



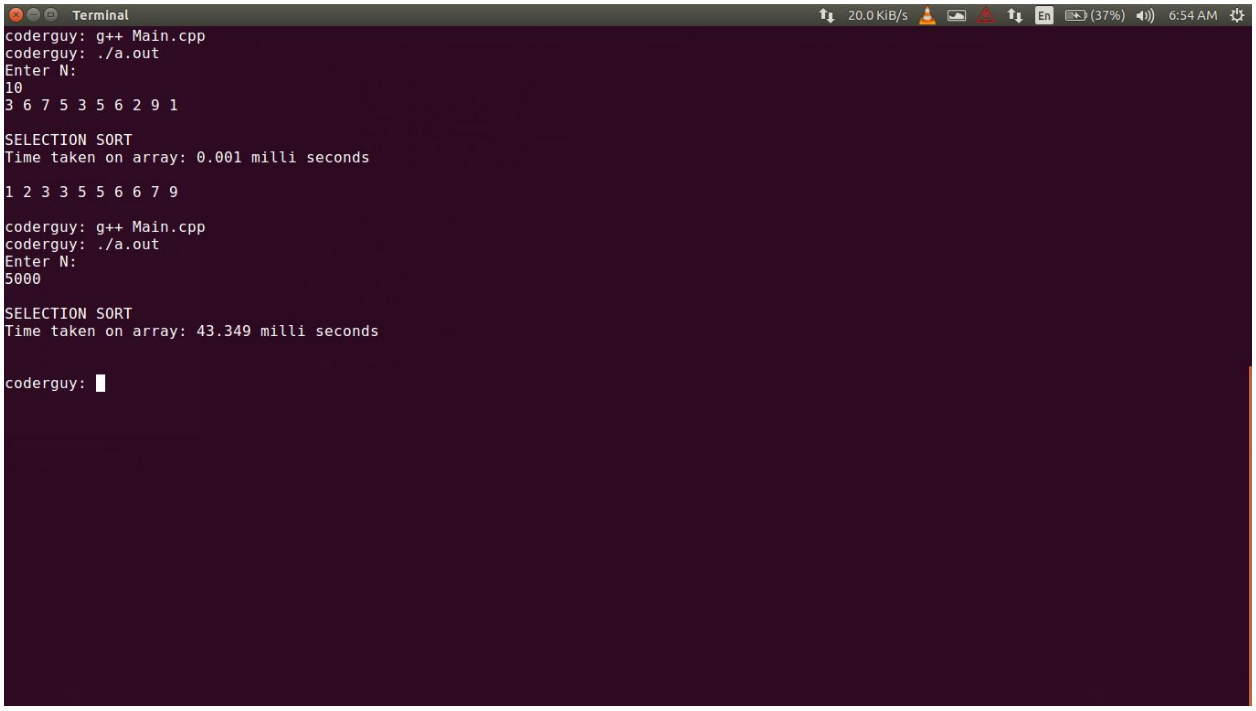
```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

QUICK SORT
Time taken on Linked List: 0.003 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

QUICK SORT
Time taken on Linked List: 1665.79 milli seconds


coderguy: 
```

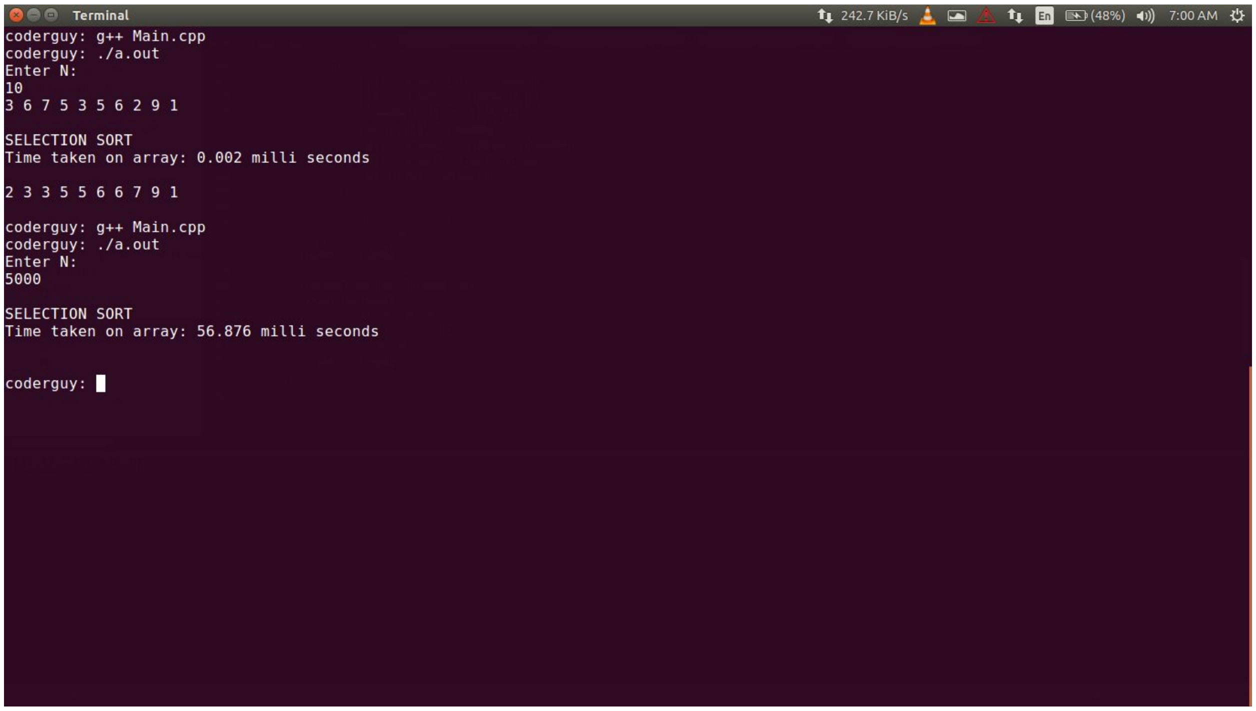


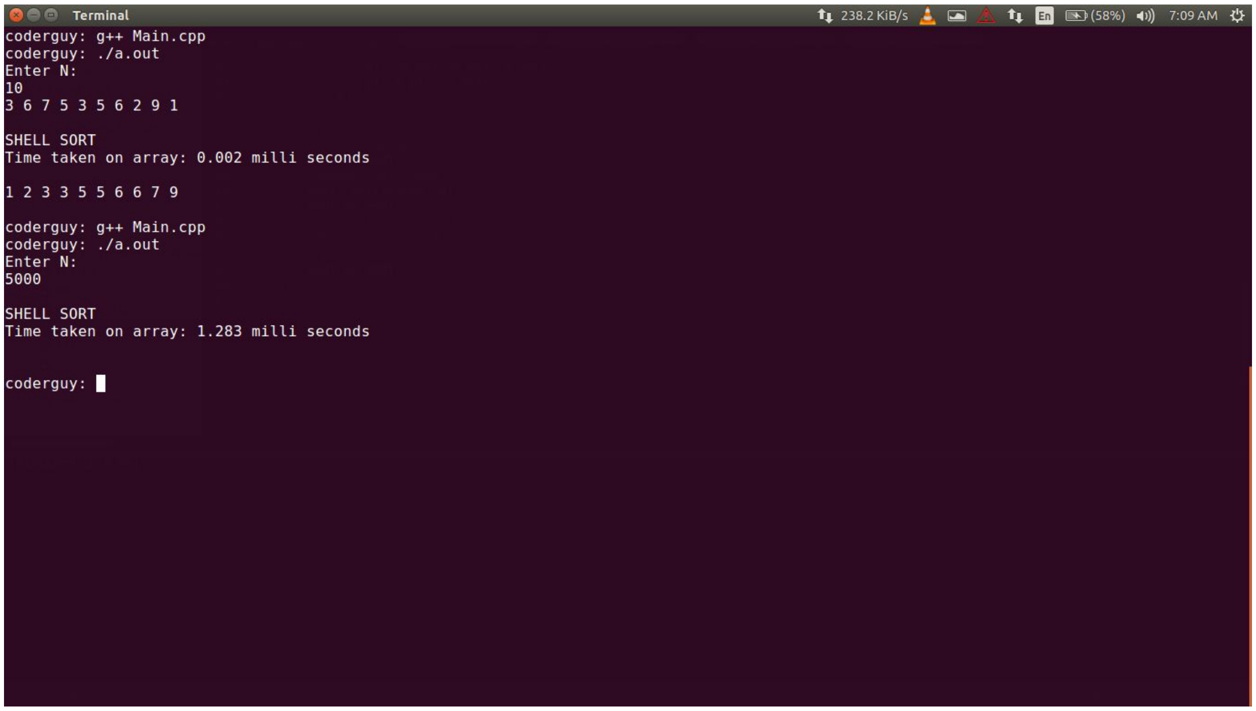
```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

RADIX SORT
Time taken on array: 0.001 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

RADIX SORT
Time taken on array: 0.665 milli seconds


coderguy: 
```



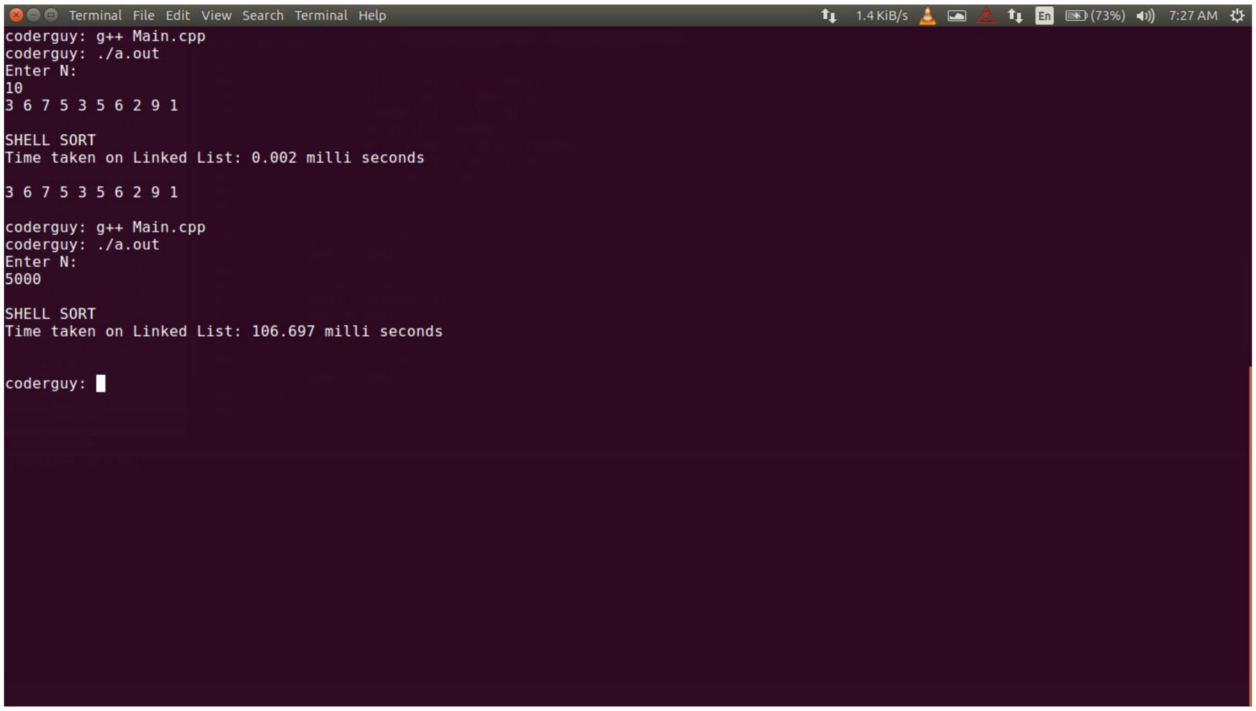
```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

RADIX SORT
Time taken on linked list: 0.002 milli seconds

3 6 7 5 3 5 6 2 9 1

coderguy: g++ Main.cpp
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

RADIX SORT
Time taken on linked list: 0.003 milli seconds


coderguy: 
```


```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

SELECTION SORT
Time taken on array: 0.001 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

SELECTION SORT
Time taken on array: 43.349 milli seconds


coderguy: 
```


```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

SELECTION SORT
Time taken on array: 0.002 milli seconds

2 3 3 5 5 6 6 7 9 1

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

SELECTION SORT
Time taken on array: 56.876 milli seconds


coderguy: 
```



```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

SHELL SORT
Time taken on array: 0.002 milli seconds

1 2 3 3 5 5 6 6 7 9

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

SHELL SORT
Time taken on array: 1.283 milli seconds


coderguy: 
```



```
coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
10
3 6 7 5 3 5 6 2 9 1

SHELL SORT
Time taken on Linked List: 0.002 milli seconds

3 6 7 5 3 5 6 2 9 1

coderguy: g++ Main.cpp
coderguy: ./a.out
Enter N:
5000

SHELL SORT
Time taken on Linked List: 106.697 milli seconds


coderguy: 
```

Group Members

1. Abhishek Jaisingh (14114002)
2. Tarun Kumar (14114068)