Genesis Roman
Group #5

HOMEWORK #1

4. **Compare the times it takes to sort a random array vs a linked list with a bubble sort.**

Random Array

Explanation:

The program begings with a couple of declarations and also the clock function which is going to generate the time the program takes so we can make the comparison:

```
    float a;
   clock_t time_req;
int rand_0toN1(int n);
int hits[10];
```

The rand_0toN1 function is declared here because it is going to be called by main. The declaration of hits creates an array of 10 integers, ranging in index from 0 to 9. Because this array is global (declared outside of any function), all its elements are initialized to 0.

Technically, the array is initialized to all-zero values because it has a static storage class. Local variables can also be declared static, causing them to retain values between calls even though they are not visible outside the function. The main function begins by defining two integer variables, n and r, and by setting the seed for the sequence of random numbers. This needs to be done in every program that uses random-number generation.

```
srand(time(nullptr));             //set seed for randomizing
```

The program then prompts for the value of n. This should look familiar by now:

```
cout<<"Enter how many trials and press enter: ";
cin>>n;
```

The next part of the program is a for loop that carries out the requested number of trials and stores results in the hits array.

```
//Run n trials. For each trial, get  a num 0 to 9
//and then increment the corresponding element
//in the hits array

for(int i=0; i<n; ++i){
```

Genesis Roman
Group #5

```
        r=rand_0toN1(10);
        ++hits[r];
}
```

Note that r could actually be defined locally to the loop, which would make sense. That is left as an exercise. Each time through, the loop gets a random number r between 0 and 9 and then records this as a "hit" for the number chosen by adding 1 to the appropriate array element. At the end of the process, the element hits[0] contains the number of Os generated, hits[1] contains the number of 1s generated, and so on.

The rest of main consists of a loop that prints all the elements of the array. This action reports the results and is run after all the trials have been performed, As before, this code is much more concise than would be the case if weren't using an array:

```
//Print all elements in the hits array, along
//with ratio of hits to expected hits (n/10)

        for(int i=0; i<10; i++){
        cout<< i << ": " << hits[i];
        double results = hits[i];
        cout << results / (n/10.0) <<endl;
}
```

The middle line of this compound statement may seem odd, but it is necessary. The results are put in a temporary variable of type double. Because double has a larger range than int, the compiler does not complain of information loss.

```
double results = hits[i];
```

This assignment needs to be done to force floating-point division in the statement that follows. Otherwise-as happens when you divide one integer by another- C++ would perform integer division, throwing fractional results away! An alterna- tive would have been to use static_cast<double>(hits[i]) to cast the data. The rand_0toN1 function is the same function I introduced at the end of Chapter 2, "Decisions, Decisions."

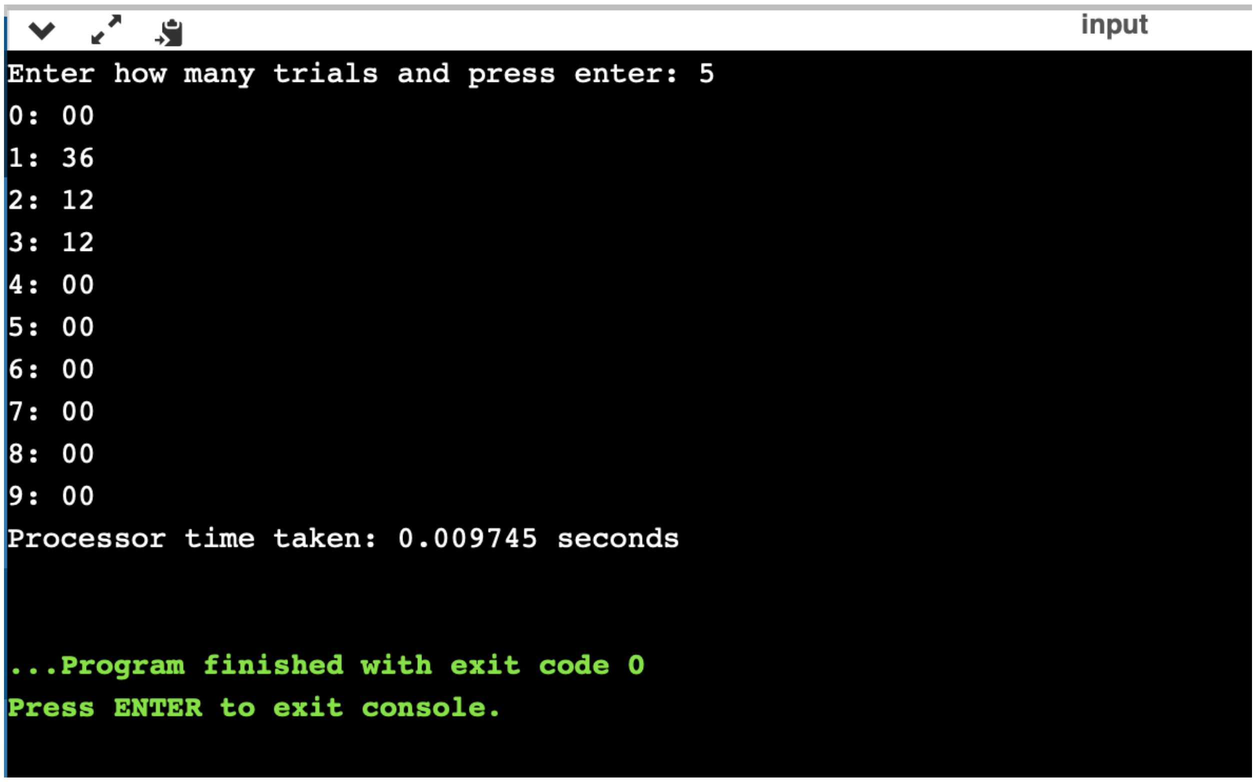
```
//Random 0-to-N1 function.
//Generate a random integer from 0 to N1

int rand_0toN1(int n){
        return rand() % n;
}
```

After that we complete the function clock() and it will show the time of the processor in the program:

Genesis Roman
Group #5

```
  time_req = clock();
    for(int i=0; i<200000; i++)
    {
        a = log(i*i*i*i);
    }
    time_req = clock()- time_req;
    cout << "Processor time taken: "
        << (float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
return 0;
}
```

Output



```
Enter how many trials and press enter: 5
0: 00
1: 36
2: 12
3: 12
4: 00
5: 00
6: 00
7: 00
8: 00
9: 00
Processor time taken: 0.009745 seconds


...Program finished with exit code 0
Press ENTER to exit console.
```

Genesis Roman
Group #5

C++ program for implementation of Bubble sort

The bubble sort algorithm is normally implemented with data in an array. It starts at the end of the data and swaps adjacent items if the later item is smaller than the earlier item. It then moves one item up in the data and repeats the operation. So the smaller items "bubble up" towards the front of the data while the larger items fall towards the bottom of the data. The algorithm is finished when it can pass through the data from end to first without swapping any values.

As stated, a bubble sort could be implemented with data in double linked list, or with a single linked list by reversing the algorithm to push larger items down the data rather than bubbling the smaller items up through the data. Here is an example including the code to test the sort function.

When bubble sort is invoked from main() the contents are rearranged into sorted order. In this program I am comparing two adjacent nodes, actual nodes are swapped instead of just swapping the data, and then the program will Print the sorted list as it shows in the output.

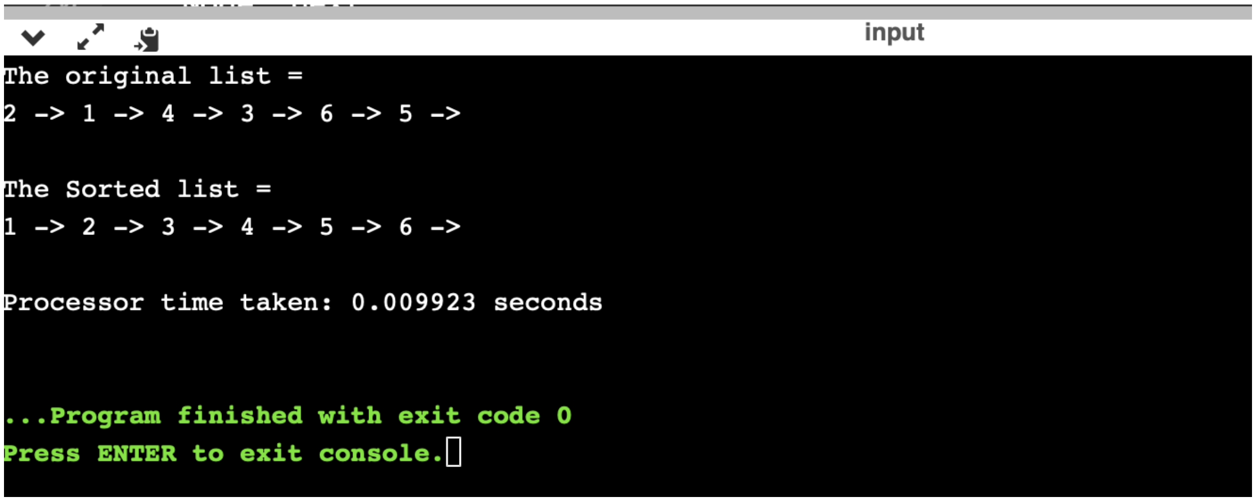
```
#include<iostream>
#include<vector>
#include<string>
#include<bits/stdc++.h>

using namespace std;

struct Node
{
        int data;
        Node *next;

        Node(int x)
        {
                data = x;
                next = NULL;
        }
};

void print_list(Node *head)             //constructor
{
        Node *start = head;

        while(start)
        {
                cout<<start->data<<" -> ";
                start = start->next;
        }
```

Genesis Roman
Group #5

```
	cout<<"\n\n";
}

void my_swap (Node *node_1, Node *node_2)
{
	int temp = node_1->data;
	node_1->data = node_2 -> data;
	node_2 -> data = temp;
}

void bubble_sort(Node *head){
	int swapped;

	Node *lPtr;			// left pointer will always point to the start of the list
	Node *rPrt = NULL;		// right pointer will always point to the end of the list
	do
	{
		swapped = 0;
		lPtr = head;
		while(lPtr->next != rPrt)
		{
			if (lPtr->data > lPtr->next->data)
			{
				my_swap(lPtr, lPtr->next);
	   swapped = 1;
			}
			lPtr = lPtr->next;
		}

//as the largest element is at the end of the list, assign that to rPtr as there is no need to
//check already sorted list

		rPrt = lPtr;
	}while(swapped);
}

int main()
{
   float a;
   clock_t time_req;
	Node *head = new Node(2);
	head -> next = new Node(1);
	head -> next -> next = new Node(4);
	head -> next -> next -> next = new Node(3);
	head -> next -> next -> next  -> next = new Node(6);
	head -> next -> next -> next  -> next -> next = new Node(5);
```

Genesis Roman
Group #5

```
            cout<<"The original list = "<<endl;
            print_list(head);

            bubble_sort(head);

            cout<<"The Sorted list = "<<endl;
            print_list(head);

  time_req = clock();
    for(int i=0; i<200000; i++)
    {
       a = log(i*i*i*i);
    }
    time_req = clock()- time_req;
    cout << "Processor time taken: "
       << (float)time_req/CLOCKS_PER_SEC << " seconds" << endl;


            return 0;
}
```



```
The original list =
2 -> 1 -> 4 -> 3 -> 6 -> 5 ->

The Sorted list =
1 -> 2 -> 3 -> 4 -> 5 -> 6 ->

Processor time taken: 0.009923 seconds


...Program finished with exit code 0
Press ENTER to exit console.
```

Comparison:

As the output of both of the program shows we can conclude that the Array takes less time than the bubble sort instead that is a simple method to use and for the program to generate it fastest.