1. Output
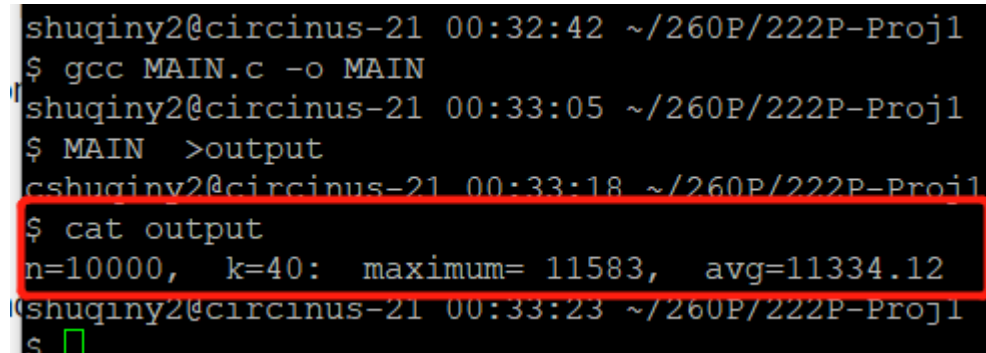
Compile the program on openLab using the following commands:

```
gcc MAIN.c -o MAIN
MAIN > output
```

The results are shown below:

2. Algorithm and Optimization

Array Best[k] always store the indices of the first k largest elements in descending value order. Compare the incoming new element with Best's tail. If the new element is larger, it has to be placed in Best array (until something larger to replace it or never). Here is the pseudo code.

```
int doalg(int n, int k, int Best[]) {
    int pointer = 0;
    Best[pointer] = 1;
    for (i = 2 to n) {
        if (COMPARE(i, Best[pointer]) == 1) {
            int p = binarySearch(Best, 0, pointer - 1, i);
            shift Best right by one slot
            Best[p] = i;
            pointer++;
        }
        else  // i < Best[pointer]
            Best[++pointer] = i;
        if (pointer >= k) pointer = k - 1;  // keep pointer < k
    }
    return 1;
}
```

- Main algorithm:

  Binary Search -> The first k element is sorted, so binary search can be used to find the position to place incoming new element.

  Insertion Sort -> Once the position is found, shift the Best array to right by a slot to make a room for the new element.

- Optimization:

  At first, I use sequential search and the output is:
  ```
  n=10000, k=40: maximum= 16609, avg=15388.86.
  ```
  It's not ideal because it takes O(N) comparison. Binary search only takes O(log N) and the result is decreased to:
  ```
  n=10000, k=40: maximum= 11598, avg=11342.81.
  ```

  It can been seen from the code that Best array is defined as `int Best[`**`MAXK`**`]` with **`MAXK`** `=` `100`, but we don't have to maintain the whole array in order. With that in mind, we only need to shift the first K elements. I implement this by using an index pointer, when pointer >= k, pointer = k – 1. By doing this, the avg comparison is improved dramatically.

  When a new element comes, it needs to be compared with `Best`'s last element. If it is smaller, simply append it to the tail; otherwise, call binary search to find a position. Since we already know the comparison between new element and `Best`'s tail, the tail doesn't have to be involved again while doing binary search. In this way, we spare a few `COMPARE` and reduce the avg by 8.

3. Analysis

- Theoretical expected WC:

Theoretically, the worst case is when the input array is an ascending ordered array. Every incoming element should be compared with the tail of `Best`, do binary search and insert. So the number of comparisons is:

$$N + \sum_{i=1}^{k} log_2(i) + (N - k) * log_2(K)$$

Given N = 10000 and k = 40, the total comparisons is 63006.

- Theoretical AVG:

Theoretically, the average number of comparisons is:

$$AVG = \sum E(x) = \sum xP(x)$$

x means the number of comparisons. Let's assume a is an incoming element, it's at index i in element array. The possibility that a doesn't need to do binary search is the same as the possibility that there are at least k elements larger than a. Therefore, the number of comparisons of one single element a is:

$$1 + P_{(a\ is\ larger\ than\ k\ elements)} * log_2 k$$

Then total comparison is:

$$N + \sum_{i=1}^{n} P(element[i]\ is\ larger\ than\ k\ elements) * log_2 k$$

Given N = 10000 and k = 40, the total comparisons is 10,967.

- Analysis for observed WC and AVG and explanation of inconsistencies:

WC   The probability of getting an ascending ordered array is: 1 / n!

Given n = 10000, 1/10000! is most 0. Even though we run the program 1000 times, it's still very unlikely to meet this situation. So I generate an array = [1,2, …, 10000] to test this out. The result can be seen below.

n= 100, k=10: maximum= 385, avg= 385.00	
n= 100, k=20: maximum= 469, avg= 469.00	
n= 100, k=40: maximum= 537, avg= 537.00	
n= 1000, k=10: maximum= 3985, avg= 3985.00	
n= 1000, k=20: maximum= 4969, avg= 4969.00	
n= 1000, k=40: maximum= 5937, avg= 5937.00	
n=10000, k=10: maximum= 39985, avg=39985.00	
n=10000, k=20: maximum= 49969, avg=49969.00	
n=10000, k=40: maximum= 59937, avg=59937.00	

when n=10000,　k=40, we get maximum= 59937, which is only 4.9% less than theoretical value.

AVG

Using different (n,k), we can get the following chart.

n= 100, k=10: maximum= 233, avg= 192.26
n= 100, k=20: maximum= 345, avg= 291.25
n= 100, k=40: maximum= 509, avg= 442.36
n= 1000, k=10: maximum= 1243, avg= 1168.46
n= 1000, k=20: maximum= 1498, avg= 1392.73
n= 1000, k=40: maximum= 1980, avg= 1840.20
n=10000, k=10: maximum= 10340, avg=10249.42
n=10000, k=20: maximum= 10741, avg=10594.92
n=10000, k=40: maximum= 11606, avg=11334.21

when n=10000,　k=40, we get avg = 11334, which is only 3.2% more than theoretical value.