Eric Watson

ECE 331 Homework 8

This assignment required us to compare the execution times of Counting Sort and Merge Sort algorithms while sorting arrays of different sizes. I kept the input arrays (to be sorted) identical for each algorithm, in order to make a valid comparison. These input arrays were populated with random integers ranging from 0 to 1000, to keep the count array for Counting Sort a reasonable size.

Execution Times for various array sizes (in microseconds)

Array Size	**10**	**100**	**500**	**750**	**1000**
Counting Sort	12	17	23	24	27
Merge Sort	13	32	142	216	280

Looking at the results of my trials one can see that the Counting Sort algorithm performed faster for each array size that was tested. Based on this outcome, and upon further analysis, I would choose to use Counting Sort in situations where I am sorting integers with a limited range. For this assignment we kept this range limited (0 to 1000) and therefore Counting Sort performed very well, in linear time. However, for a wide range of values, counting sort can take up a significant amount of memory. Accordingly, I would use Merge Sort when I need a stable, $\theta(n \lg n)$ sort, in the case that the values are spread across a wide range. Merge Sort would be able to handle the wide-range sorting without taking as much memory.

The terminal output below demonstrates the full functionality of my program. Here I am testing an array size of 10, so that I can print out the randomly generated input array, and the resulting sorted arrays, without creating a lengthy output. I first print the random input array values. Next I print out the value of k, the largest generated value, which determines the size of Counting Sorts array C (for debugging purposes). Then I print out the execution time of Counting Sort, and the sorted values to ensure that the algorithm worked correctly. In a similar manner, I also print out the execution time of Merge Sort, and its sorted array values.

ARRAY SIZE: 10

```
Erics-MacBook-Pro-2:Code Watson$ gcc HW08.c
Erics-MacBook-Pro-2:Code Watson$ ./a.out
Input Array Size: 10
The first 10 randomly generated values are:
835
159
73
374
408
291
186
427
259
900
 k = 900
```

```
Counting Sort Execution Time: 0.000012 seconds
The first 10 Counting-Sorted values are:
73
159
186
259
291
374
408
427
835
900
Merge Sort Execution Time: 0.000013 seconds
The first 10 Merge-Sorted values are:
73
159
186
259
291
374
408
427
835
900
```

TESTING

ARRAY SIZE 100

```
Erics-MacBook-Pro-2:Code Watson$ ./a.out
Input Array Size: 100
 k = 992
Counting Sort Execution Time: 0.000017 seconds
The first 10 Counting-Sorted values are:
8
9
20
33
40
42
43
75
83
85
Merge Sort Execution Time: 0.000032 seconds
The first 10 Merge-Sorted values are:
8
9
20
33
40
42
43
75
83
85
```

ARRAY SIZE 500:

```
Erics-MacBook-Pro-2:Code Watson$ ./a.out
Input Array Size: 500
 k = 997
Counting Sort Execution Time: 0.000023 seconds
The first 10 Counting-Sorted values are:
2
2
2
5
9
10
10
12
12
13
Merge Sort Execution Time: 0.000142 seconds
The first 10 Merge-Sorted values are:
2
2
2
5
9
10
10
12
12
13
```

ARRAY SIZE 750:

```
Erics-MacBook-Pro-2:Code Watson$ ./a.out
Input Array Size: 750
 k = 999
Counting Sort Execution Time: 0.000024 seconds
The first 10 Counting-Sorted values are:
2
3
6
8
10
11
12
20
21
21
Merge Sort Execution Time: 0.000216 seconds
The first 10 Merge-Sorted values are:
2
3
6
8
10
11
12
20
21
21
```

ARRAY SIZE 1000:

```
Erics-MacBook-Pro-2:Code Watson$ ./a.out
Input Array Size: 1000
 k = 999
Counting Sort Execution Time: 0.000027 seconds
The first 10 Counting-Sorted values are:
0
1
2
2
2
3
3
4
5
6
Merge Sort Execution Time: 0.000280 seconds
The first 10 Merge-Sorted values are:
0
1
2
2
2
3
3
4
5
6
```

Eric Watson

ECE 331 Homework 8 Source Code (HW08.c)

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

void print_first_ten(int A[]){
  int i;
  for (i =0; i < 10; i++){
    printf("%d\n", A[i]);
  }
}
void merge (int B[], int p, int q, int r){
  int i, j, k;
  int n1 = q - p + 1; //length of subarray A[p..q]
  int n2 = r - q;                  //length of subarray A[q+1..r]
// let L[1..n1+1] and R[1..n2+1] be new arrays
  int L[n1+1], R[n2+1];

  for (i=0; i< n1; i++){
    L[i] = B[p+i-1];
  }

  for (j=0; j< n2; j++){
    R[j] = B[q+j];
  }
//      Issue sentinel cards
  L[n1] = 9999999;
  R[n2] = 9999999;

  i = 0;
  j = 0;

  for (k = (p-1); k < r; k++){
    if (L[i] <= R[j]){
      B[k] = L[i];
      i = i + 1;
    }
    else{
      B[k] = R[j];
      j=j+1;
    }
  }
}
void merge_sort(int B[], int p, int r){

  if (p < r){
    int q = floor((p+r)/2);
    merge_sort(B, p, q);
    merge_sort(B, q+1, r);
    merge (B,p,q,r);
  }
}

void counting_sort(int A[], int B[], int k, int size){
  int i,j;
  int C[k+1];                  // C[k+1] = [0...k]
// let C[0 to k] be new array
// c contains the number of each value
  for (i=0; i <= k; i++){
    C[i]=0;
  }
  for (j=0; j < size; j++){
    C[A[j]] = C[A[j]] + 1;
// C[i] now contains the number of elements equal to i.
  }
  for (i = 1; i <= k; i++){   // running sum
    C[i] = C[i] + C[i-1];
// C[i] now contains the number of elements less than or equal to i.
```

```
  }
  for (j = size-1; j >= 0; --j){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]] - 1;
  }
}

int main(){
  clock_t begin_count, end_count, begin_merge, end_merge;
  double count_time;
  double merge_time;
  int size,i;
  int k =0;

  time_t seconds;
  time(&seconds);
  srand((unsigned int) seconds);

  printf("Input Array Size: ");
  scanf("%d", &size);
  int A[size];                      // Counting Sort
  int B[size];                      // Counting Sort Output Array
  int M[size];                      // Merge Sort

  for (i=0; i< size; i++){          // Populate A with random integers
    A[i] = rand() % 1000;
  }

//DEBUGGING - Make sure the integers are randomly generated
//  printf("The first 10 randomly generated values are:\n");
//  print_first_ten(A);

  for (i=0; i < size; i++){         // Copy the sorted values of A into M
    M[i] = A[i];
  }
// Determine the largest value k
  for (i=0; i < size; i++){
    if (A[i] > k){
      k = A[i];
    }
  }

  printf(" k = %d\n", k);

  begin_count = clock();            // start clock
  counting_sort(A, B, k, size);    // Counting Sort the values of A, then place into B
  end_count = clock();              // end clock
  count_time = (double)(end_count-begin_count)/ CLOCKS_PER_SEC;
  printf("Counting Sort Execution Time: %f seconds\n", count_time);
  printf("The first 10 Counting-Sorted values are:\n");
// Separate printing method due to the change in indices
// Print the firt 10 sorted values
    for (i = 1; i <= 10; i++){
        printf("%d\n", B[i]);
}

  begin_merge = clock();          //start clock
  merge_sort(M,1,size);           // merge sort the values of B
  end_merge = clock();            // end clock
  merge_time = (double)(end_merge-begin_merge)/ CLOCKS_PER_SEC;

  printf("Merge Sort Execution Time: %f seconds\n", merge_time);

//DEBUGGING - Make sure the values are sorted properly
  printf("The first 10 Merge-Sorted values are:\n");
  print_first_ten(M);

  return(0);
}
```