Zuhair Ahmed
CS 325-400 Spring 2020
4/5/2020

Homework #1

1) (1 pt) Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in 64nlgn steps. For what values of n does insertion sort run faster than merge sort?
Note: lg n is log "base 2" of n or $\log_2 n$. For most calculators you would use the change of base theorem to numerically calculate lgn.

For all values of n between 2 and 43, Insertion sort runs faster. For all values of n that are 44 and greater, Merge sort runs faster.

n	Insertion Sort - $8n^2$ steps	Merge Sort - 64nlgn steps
64	32,768	24,576
44	15,488	15,374
43	14,792	14,933
32	8,192	10,240
16	2,048	4,096
8	512	1,536
4	128	512
2	32	128

2) (5 pts) For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct and explain.

a. $f(n) = n^{0.25}$; $g(n) = n^{0.5}$
As n approaches infinity, g(n) grows at a faster rate than f(n) and therefore represents that upper bounds for f(n). Hence, $n^{0.25}$ *is* $O(n^{0.5})$

b. $f(n) = n$; $g(n) = \log^2 n$
As n approaches infinity, g(n) grows at a slower rate than f(n) and therefore represents that lower bounds for f(n). Hence, n *is* $\Omega(\log^2 n)$

c. f(n) = log n; g(n) = ln n
As n approaches infinity, g(n) grows at the same rate as f(n) and therefore represents tight bounds for f(n). Hence, log n $is\ \Theta$(ln n)

d. f(n) = $1000n^2$; g(n) = $0.0002n^2 - 1000n$
As n approaches infinity, g(n) grows at the same rate as f(n) and therefore represents tight bounds for f(n). Hence, $1000n^2\ is\ \Theta(0.0002n^2 - 1000n)$

e. f(n) = nlog n; g(n) =$n\sqrt{n}$
As n approaches infinity, g(n) grows at a faster rate as f(n) and therefore represents upper bounds for f(n). Hence, nlog n is O($n\sqrt{n}$)

f. f(n) = e^n; g(n) = 3^n
As n approaches infinity, g(n) grows at a faster rate as f(n) and therefore represents upper bounds for f(n). Hence, $e^n\ is$ O(3^n)

g. f(n) = 2^n; g(n) = 2^{n+1}
As n approaches infinity, g(n) grows at a faster rate as f(n) and therefore represents upper bounds for f(n). Hence, $2^n\ is$ O(2^{n+1})

h. f(n) = 2^n; g(n) =2^{2n}
As n approaches infinity, g(n) grows at a faster rate as f(n) and therefore represents upper bounds for f(n). Hence, $2^n\ is$ O(2^{2n})

i. f(n) = 2^n; g(n) = n!
As n approaches infinity, g(n) grows at a faster rate as f(n) and therefore represents upper bounds for f(n). Hence, $2^n\ is$ O(n!)

j. f(n) = lgn; g(n) = $\sqrt{n}$
As n approaches infinity, g(n) grows at a faster rate as f(n) and therefore represents upper bounds for f(n). Hence, lgn is O($\sqrt{n}$)

3) (4 pts) Let f_1 and f_2 be asymptotically positive non-decreasing functions. Prove or disprove each of the following conjectures. To disprove give a counter example.

a. If $f_1(n) = \Theta(g(n))$ and $f_2(n) = \Theta(g(n))$ then $f_1(n) = \Theta(f_2(n))$.

- Statement is true.
- If $f_1 = \Theta(g(n))$ then this also implies $g = \Theta(f_1(n))$
- If $f_2 = \Theta(g(n))$ then this also implies $g = \Theta(f_2(n))$
- Hence, by transitive property $\Theta(f_1(n)) = \Theta(f_2(n))$ or $f_1(n) = \Theta(f_2(n))$

b. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = \Theta(g_1(n) + g_2(n))$

- Statement is false.
- Counter example is when $f_1(n) = n$, $f_2(n) = n$, $g_1(n) = n!$, $g_2(n) = n^2$
- $f_1(n)$ is indeed $O(g_1(n)$ and $f_2(n)$ is indeed $O(g_2(n))$
- However, $f_1(n) + f_2(n) = \Theta(g_1(n) + g_2(n))$ cannot be true since: $n + n \neq \Theta(n! + n^2)$

4) (10 pts) Merge Sort and Insertion Sort Programs
Implement merge sort and insertion sort to sort an array/vector of integers. Name your programs "mergesort" and "insertsort". Both programs should read inputs from a file called "data.txt" where the first value of each line is the number of integers that need to be sorted, followed by the integers.
Example values for data.txt:
4 19 2 5 11
8 1 2 3 4 5 6 1 2
The output will be written to files called "merge.out" and "insert.out".
For the above example the output would be:
2 5 11 19
1 1 2 2 3 4 5 6
To receive full credit all code must be commented.
Submit a copy of your insertsort and mergesort in a ZIP file to TEACH. We will test execution with an input file named data.txt.

```
/*******************************
 * Author: Zuhair Ahmed
 * Date Created: 4/5/2020
 * Filename: insertSort.cpp
 * Overview: This program sorts unsorted list of numbers via Insertion Sort
 * Input: "data.txt" file with first element being length followed by
 *        unsorted numbers
 * Output: Sorted list of numbers back to "insert.out" file
 * ******************************/

#include <iostream>
#include <fstream>
#include <cstdlib>

void print(int* array, int size);
void insertionSort(int* array, int size);

int main()
{
```

```
  int n; // number of unsorted elements from data.txt file
  const int MAX = 9999;
  int arr[MAX];

  std::fstream myFile;
  myFile.open ("data.txt");
  if (myFile.fail()) //check to ensure file is avalible to open
  {
    std::cout << "Unable to open file!" << std::endl;
    exit(1); // terminate with error
  }
  else
  {
     myFile >> n; //input length of unsorted list into n
     for (int i = 0; i < n; i++) //input remaining elements into array
         myFile >> arr[i];
  }

  std::cout << "Array before Sorting: ";
  print(arr, n);
  insertionSort(arr, n);

  std::cout << "Array after Sorting: ";
  print(arr, n);

  myFile.close();
  return 0;
}

void print(int *array, int size)
{
  for(int i = 0; i < size; i++) //loop around array and output each element
    std::cout << array[i] << " ";

  std::cout << std::endl;
}

void insertionSort(int *array, int size)
{
  int key, j;
  for(int i = 1; i < size; i++)
  {
    key = array[i]; //set key equal to first element which is now sort half of array
```

```
    j = i;
    while((j > 0) && (array[j-1] > key)) //while element is is > than key
    {
      array[j] = array[j-1]; //swap
      j--;
    }

    array[j] = key; //advance key to next element right of sorted array
  }
}

/*******************************
 * Author: Zuhair Ahmed
 * Date Created: 4/5/2020
 * Filename: mergeSort.cpp
 * Overview: This program sorts unsorted list of numbers via Merge Sort
 * Input: "data.txt" file with first element being length followed by
 *       unsorted numbers
 * Output: Sorted list of numbers back to "insert.out" file
 * ******************************/

#include <iostream>
#include <fstream>
#include <algorithm>
void print(int* array, int size);
void mergeSort(int* array, int start, int end);
void mergeHelper(int array[], int Firstindex, int mid, int Lastindex);

int main()
{
  int n; // number of unsorted elements from data.txt file
  const int MAX = 9999;
  int array[MAX];

  std::fstream myFile;
  myFile.open ("data.txt");
  if (myFile.fail()) //check to ensure file is avalible to open
  {
    std::cout << "Unable to open file!" << std::endl;
    exit(1); // terminate with error
  }
  else
  {
```

```
      myFile >> n; //input length of unsorted list into n
      for (int i = 0; i < n; i++) //input remaining elements into array
          myFile >> array[i];
  }

  std::cout << "Array before Sorting: ";
  print(array, n);
  int start = 0;
  int end = n-1;
  mergeSort(array, start, end);

  std::cout << "Array after Sorting: ";
  print(array, n);

  myFile.close();
  return 0;
}

void print(int *array, int size)
{
  for(int i = 0; i < size; i++) //loop around array and output each element
    std::cout << array[i] << " ";

  std::cout << std::endl;
}

void mergeSort(int* array, int start, int end)
{
   if (start < end)
   {
      int mid = start + (end - start)/2; //find middle index

      mergeSort(array, start, mid); //recursively sort first half
      mergeSort(array, mid+1, end); //recursively sort second half

      mergeHelper(array, start, mid, end); //recursively merge two halves together
   }
}

void mergeHelper(int array[], int Firstindex, int mid, int Lastindex)
{
   //temp variables
   int z;
```

```
int y;
int x;

//define two halves of merged array
int sub1 = mid - Firstindex + 1;
int sub2 =  Lastindex - mid;

int Second[sub2]; //temp array2
int First[sub1];  //temp array1

//push elements into temp array1
for (x = 0; x < sub1; x++)
   First[x] = array[Firstindex + x];
//push elements into temp array2
for (y = 0; y < sub2; y++)
   Second[y] = array[mid + 1+ y];

z = Firstindex;
y = 0;
x = 0;

//while temp variable is less then temp array swap elements
while (x < sub1 && y < sub2)
{
   if (First[x] <= Second[y])
   {
      array[z] = First[x];
      x++;
   }
   else
   {
      array[z] = Second[y];
      y++;
   }
   z++;
}
while (x < sub1)
{
   array[z] = First[x];
   z++;
   x++;
}
while (y < sub2)
```

```
   {
      array[z] = Second[y];
      z++;
      y++;
   }
}
```

5) (10 pts) Merge Sort vs Insertion Sort Running time analysis
a) Modify code- Now that you have verified that your code runs correctly using the data.txt input file, you can modify the code to collect running time data. Instead of reading arrays from the file data.txt and sorting, you will now generate arrays of size n containing random integer values from 0 to 10,000 to sort. Use the system clock to record the running times of each algorithm for n = 5,000, 10,000, 15,000, 20,000, …. You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data. Output the array size n and time to the terminal. Name these new programs insertTime and mergeTime.
Submit a copy of the timing programs to TEACH in the Zip file from problem 5, also include a "text" copy of the modified timing code in the written HW submitted in Canvas.

```
/*******************************
 * Author: Zuhair Ahmed
 * Date Created: 4/5/2020
 * Filename: insertTime.cpp
 * Overview: This program sorts unsorted list of numbers via Insertion Sort
 * Input: Random integers from 0 to 10,000
 * Output: Array size n and run time to terminal
 * ****************************/

#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <ctime>

void print(int* array, int size);
void insertionSort(int* array, int size);

int main()
{
  const int MAX = 10000;
  int arrLength = 100000;
```

```
  int arr[arrLength];

  srand(time(NULL)); //seed random number

  //fill array with random values
  for (int i = 0; i < arrLength; i++)
    arr[i] = rand() % MAX;

 //  std::cout << "Array before Sorting: ";
//   print(arr, arrLength);
 //  std::cout << std::endl;

  std::clock_t start;
  start = std::clock();
  insertionSort(arr, arrLength); //sort array!

  std::cout << "Array after Sorting \n";
//   print(arr, arrLength);
  double duration = (std::clock() - start) / (double) CLOCKS_PER_SEC;

  std::cout << "Array Length is: " << arrLength << std::endl;
  std::cout << "Sorting took me via Insertion Sort: " << duration << " seconds";
  std::cout << std::endl;

  return 0;
}

void print(int *array, int size)
{
  for(int i = 0; i < size; i++) //loop around array and output each element
    std::cout << array[i] << " ";

  std::cout << std::endl;
}

void insertionSort(int *array, int size)
{
  int key, j;
  for(int i = 1; i < size; i++)
  {
    key = array[i]; //set key equal to first element which is now sort half of array
    j = i;
    while((j > 0) && (array[j-1] > key)) //while element is is > than key
```

```
    {
      array[j] = array[j-1]; //swap
      j--;
    }

    array[j] = key; //advance key to next element right of sorted array
  }
}

/*******************************
 * Author: Zuhair Ahmed
 * Date Created: 4/5/2020
 * Filename: insertTime.cpp
 * Overview: This program sorts unsorted list of numbers via Merge Sort
 * Input: Random integers from 0 to 10,000
 * Output: Array size n and run time to terminal
 * ******************************/

#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <ctime>

void print(int* array, int size);
void mergeSort(int* array, int start, int end);
void mergeHelper(int array[], int Firstindex, int mid, int Lastindex);

int main()
{
  const int MAX = 10000;
  int arrLength = 100000;
  int arr[arrLength];

  srand(time(NULL)); //seed random number

  //fill array with random values
  for (int i = 0; i < arrLength; i++)
    arr[i] = rand() % MAX;

 //  std::cout << "Array before Sorting: ";
 //  print(arr, arrLength);
 //  std::cout << std::endl;
```

```
  std::clock_t start;
  start = std::clock();
  int beg = 0;
  int end = arrLength-1;
  mergeSort(arr, beg, end); //sort array!

  std::cout << "Array after Sorting \n";
//   print(arr, arrLength);
  double duration = (std::clock() - start) / (double) CLOCKS_PER_SEC;

  std::cout << "Array Length is: " << arrLength << std::endl;
  std::cout << "Sorting took me via Merge Sort: " << duration << " seconds";
  std::cout << std::endl;

  return 0;
}

void print(int *array, int size)
{
  for(int i = 0; i < size; i++) //loop around array and output each element
    std::cout << array[i] << " ";

  std::cout << std::endl;
}

void mergeSort(int* array, int start, int end)
{
   if (start < end)
   {
      int mid = start + (end - start)/2; //find middle index

      mergeSort(array, start, mid); //recursively sort first half
      mergeSort(array, mid+1, end); //recursively sort second half

      mergeHelper(array, start, mid, end); //recursively merge two halves together
   }
}

void mergeHelper(int array[], int Firstindex, int mid, int Lastindex)
{
   //temp variables
   int z;
   int y;
```

```
int x;

//define two halves of merged array
int sub1 = mid - Firstindex + 1;
int sub2 =  Lastindex - mid;

int Second[sub2]; //temp array2
int First[sub1];  //temp array1

//push elements into temp array1
for (x = 0; x < sub1; x++)
   First[x] = array[Firstindex + x];
//push elements into temp array2
for (y = 0; y < sub2; y++)
   Second[y] = array[mid + 1+ y];

z = Firstindex;
y = 0;
x = 0;

//while temp variable is less then temp array swap elements
while (x < sub1 && y < sub2)
{
   if (First[x] <= Second[y])
   {
      array[z] = First[x];
      x++;
   }
   else
   {
      array[z] = Second[y];
      y++;
   }
   z++;
}
while (x < sub1)
{
   array[z] = First[x];
   z++;
   x++;
}
while (y < sub2)
{
```

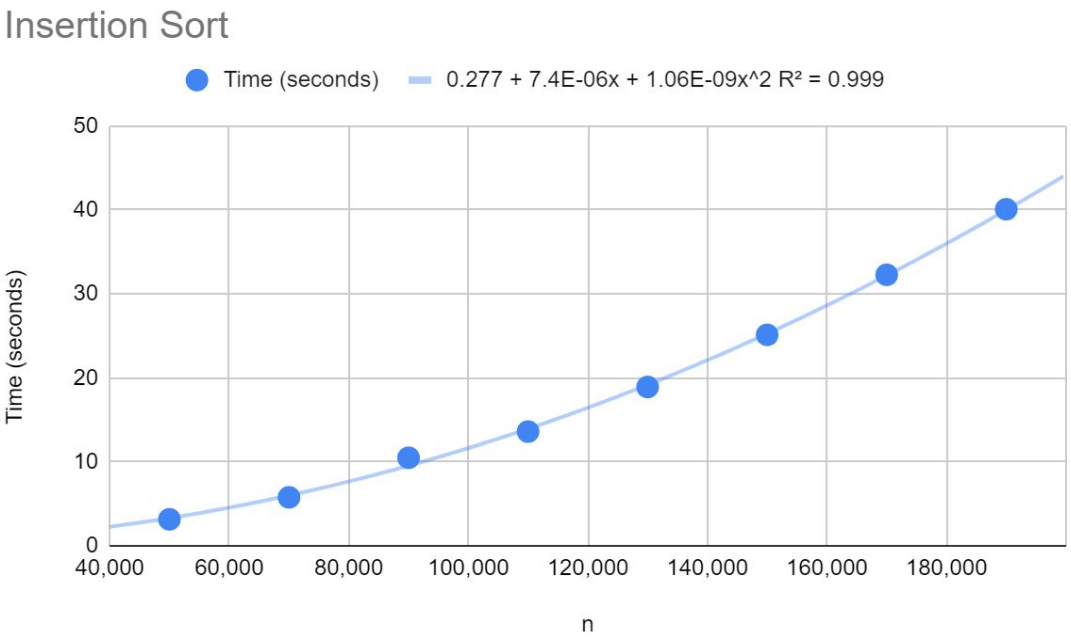
```
      array[z] = Second[y];
      z++;
      y++;
   }
}
```

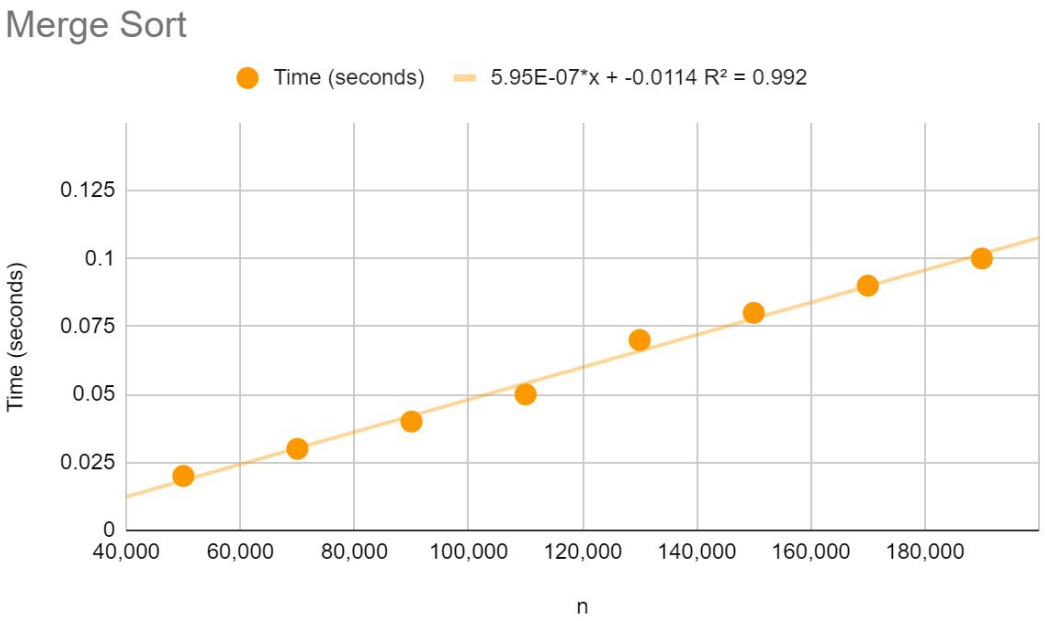
b) Collect running times - Collect your timing data on the engineering server. You will need at least seven values of t (time) greater than 0. If there is variability in the times between runs of the same algorithm you may want to take the average time of several runs for each value of n. Create a table of running times for each algorithm.

Insertion Sort	
n	**Time (seconds)**
50,000	3.15
70,000	5.77
90,000	10.48
110,000	13.6
130,000	18.91
150,000	25.12
170,000	32.27
190,000	40.08

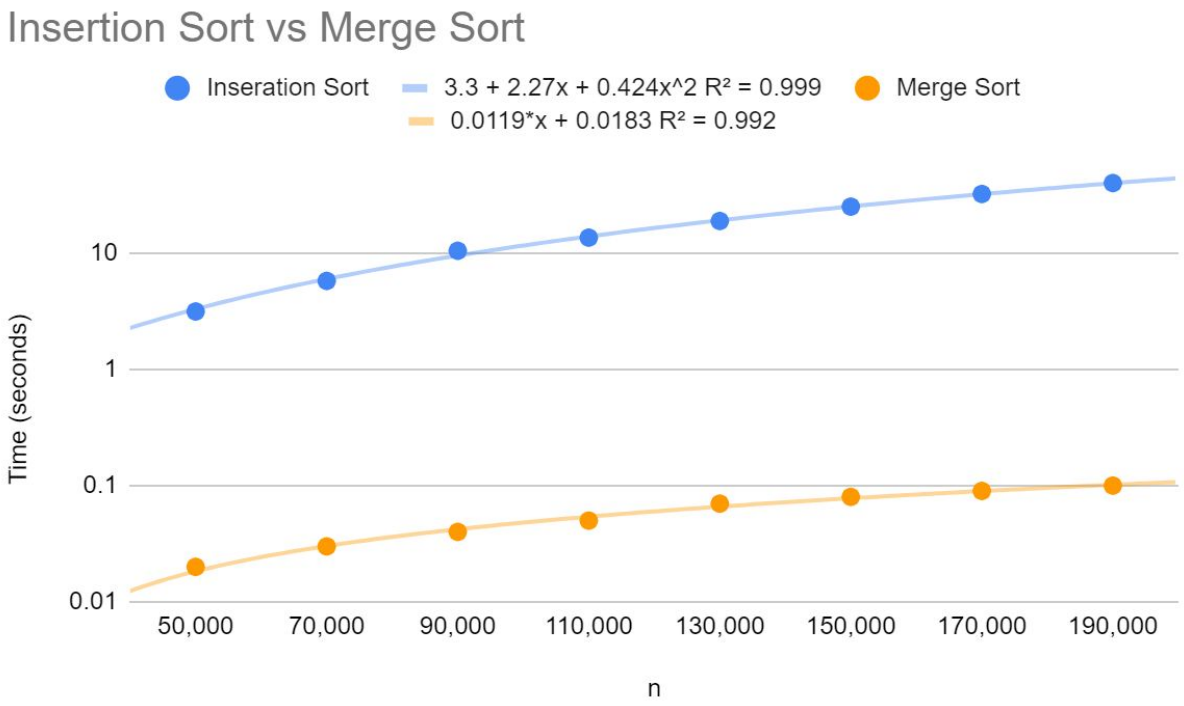
Merge Sort	
n	**Time (seconds)**
50,000	0.02
70,000	0.03
90,000	0.04
110,000	0.05
130,000	0.07
150,000	0.08
170,000	0.09
190,000	0.10

c) Plot data and fit a curve - For each algorithm plot the running time data you collected on an individual graph with n on the x-axis and time on the y-axis. You may use Excel, Matlab, or any other software. What type of curve best fits each data set? Give the equation of the curves that best "fits" the data and draw that curves on the graphs.





d) Combine - Plot the data from both algorithms together on a combined graph. If the scales are different you may want to use a log-log plot.



e) Comparison - Compare your experimental running times to the theoretical running times of the algorithms? Remember, the experimental running times were the "average case" since the input arrays contained random integers.

From a theoretical perspective, for "average case" as well "worst case" Insertion Sort has a time complexity of $O(n^2)$ vs Merge sort which has O(n*logn). Given these were random inputs we would expect these empirical results to closely mimic "average case" results which they do appear to do. Insertion sort is growing an exponential rate as indicated by curve of best fit is exponential vs Merge sort which is growing in linear time as indicated by line of best fit. Both have very high R^2 values so these curves appear to a good approximation of trendline of scatter plot. Hence we can conclude, as values get to be much larger the Merge sort algorithm will outperform Insertion sort algorithm. Note all tests were conducted on Flip/OSU engineering servers.

Submit your code to TEACH in a zip file with the code from Problems 4 & 5.