Yash Divecha
(Section 1)

Design and Analysis of Algorithm (DAA)
Project 1

ydivech1@binghamton.edu

"I have done this assignment completely on my own. I have not copied it, nor have I given my solution to anyone else. I understand that if I am involved in plagiarism or cheating I will have to sign an official form that I have cheated and that this form will be stored in my official university record. I also understand that I will receive a grade of 0 for the involved assignment for my first offense and that I will receive a grade of "F" for the course for any additional offense."

Name: Yash Divecha Sign: ____________________

****

Solution 1: Relevant files:

- "yash_divecha_proj1.c" – Source Code.
- README file - Explains how to compile and execute the source file.
- Makefile – Compiles the source code into an object file and cleans the object files as well by make clean.

Solution 2: Project1.pdf

1. **Insertion Sort:** By using Method 2 approach, Barometer operation is comparison (<), hence as per the source code.

```
for(i=1;i<n;i++){
        for(j=i;j>0;j--){
                if(a[j] < a[j-1]){            //checks if the previous number is greater
                        swap(a[j], a[j-1])
                }
        }
}
```

$\sum_{i=1}^{n-1}\sum_{j=1}^{i}(1) = \sum_{i=1}^{n-1} i$ = 1+2+3+…+(n-1) = (n-1)*n/2

Hence instruction count of insertion sort is Θ (n^2)

2. **Counting Sort:** Barometer operator is c[j] = c[j]-1, hence as per the source code.

Even if, for loop goes till "s" the barometer instruction runs till "n", due to while loop condition c[j] > 0. where s = length of counting array and n = length of original array.

```
1. for(j=0;j<=99;j++){              //here "s" is 99
        2. while(c[j] > 0){         //due to this condition it loops till n (length of a[])
        3.      a[q] = j;           //generating final array depending on the count array.
        4.      c[j] = c[j]-1;      //Barometer operation.
        5.      q=q+1;
        }
}
```

$\sum_{j=1}^{s} Count\ 2(4) = \sum_{i=1}^{s} c[j]$ = n.

Hence instruction count of counting sort is Θ (n).

---

3. Merge Sort:

As we can see from the source code that, dividing **[Divide]** takes two recursive calls with each taking n/2 time complexity and merge **[Conquer]** takes linear time with n time complexity as it has to merge two subarrays. (left and right)

```
if(lowIndex<highIndex){                                            ............. constant time
      midIndex=(lowIndex+highIndex)/2;                             ............. constant time
      mergeSort(a,n,lowIndex,midIndex);                            ............. T(n/2)
      mergeSort(a,n,midIndex+1,highIndex);                         ............. T(n/2)
      merge(a,n,lowIndex,midIndex,midIndex+1,highIndex);           ............. T(n)
}
```

Merge-Sort recursive calls are running for n/2 times (left subarray) and n/2 times (right subarray).
Merge – Loops in the merge function are running for total length of left subarray (n/2) times + total length of right subarray (n/2) times = n/2 + n/2 = n times.

By using the method 1 of Instruction count, we can formulate the recursive equations by counting Division cost Θ(1) + merge cost Θ(n) and recursive calls cost 2*T(n/2) .

Recursive equations.

T(n) = 1 when n =1

= T(n/2) + T(n/2) + n when n>1

Hence solving the recursive equations:

T(n) = 2T(n/2) + n

= 2(2 T(n/4) + (n/2)) +n recursive call.

= 4 T(n/4) + n+n = 4 T(n/4) + 2n

= 8 T(n/8) + 3n recursive call.

= 16 T(n/16) + 4n recursive call.

.

.

.

$= 2^k\, T\left(\frac{n}{2^k}\right) + kn$ (i)

Assume n =2^k, then k=$\log_2 n$, substituting in (i) we get.

= n T(1) + $(\log_2 n)$n

For Time complexity we consider the highest order. Hence
Merge Sort = Θ($(\mathrm{n}\log_2 n)$