Member 1: **Andrea Laserna** CMSC 142 - 1 Machine Problem 1

Member 2: **Claire Dane Vincoy**

CMSC 142 Machine Problem 1

1. Determine the Big-O complexities of the following snippet of codes. Show the big-O for each block of code (if, for, while, etc.) then evaluate the final big-O from this.

   a)
```
void fun(int x) {                        T(n)
    if (x == 0) return;                  1
    for (int i = 0; i < x; i++)          n+1
        op();                            n
    fun(x-1);                            T(n-1)
}
```

   This can be written as:

   T(n) = T(n-1) + n, where T(0) = 1 is the base case.

   To solve this recurrence relation, we expand it.

   1. T(n) = T(n-1) + n
   2. T(n) = T(n-2) + n-2 + n
   3. T(n) = T(n-3) + n-3 + n-2 + n

   From this, we get a **generalized pattern** of:

   T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + (n-(k-3)) + … + (n-1) + n

   Assume: n - k = 0, therefore, n = k. So,

   T(n) = T(n-n) + (n-(n-1)) + (n-(n-2)) + (n-(n-3)) + … + (n-1) + n

   T(n) = T(0) + (n-n+1) + (n-n+2) + (n-n+3) + … + (n-1) + n

   Simplifying:

   T(n) = T(0) + 1 + 2 + 3 + … + (n-1) + n

   Notice that the pattern follows the sum of natural numbers. Therefore, we have:

   $$T(n) = 1 + n(\frac{n+1}{2})$$

   Since, $T(n) = 1 + n(\frac{n+1}{2})$, **the Big O complexity is O(n^2).**

   b)
```
void fun(int x) {                        T(n)
    if (x == 0) return;                  1
    for (int i = 0; i < x; i++)          n+1
        op();                            n
```

```
    for(int i = 0; i < 4; i++)          4
        fun(x/4);                        T(n/4)
}
```

This can be written as:

$T(n) = 4 \cdot T(\frac{n}{4}) + n$, where T(0) = 1 is the base case.

To solve this recurrence relation, we expand it.

1. $T(n) = 4 \cdot T(\frac{n}{4}) + n$
2. $T(n) = 16 \cdot T(\frac{n}{16}) + 2n$
3. $T(n) = 64 \cdot T(\frac{n}{64}) + 3n$

From this, we get a **generalized pattern** of:

$$T(n) = 4^k T(n/4^k) + kn$$

Assume:

$$\frac{n}{4^k} = T(0)$$
$$\frac{n}{4^k} = 1$$
$$n = 4^k$$
$$log_4 n = log_4 4^k$$
$$log_4 n = k$$

Simplifying:

$$T(n) = n \times 1 + log_4 n \times n$$

Since, $T(n) = n + n\, log_4 n$, **the Big O complexity is O(n log n).**

c) function fun(int x) {

```
m = 0                           1
while m * m < x + 50 do         n
    op()                        n
    m++                         n
end while
return 0                        1
}
```

This can be written as:

$$i^2 < n + 50$$

Taking the squares of both sides:

$$i = \sqrt{n + 50}$$

Solving for i, we get $\sqrt{n + 50}$. So, **the Big O complexity is O**$(\sqrt{n})$.

d) For each of the following pairs of functions f(n) and g(n), determine whether f(n)=O(g(n)), g(n)=O(f(n)), or both. Show your solution/explanation.

a. $f(n) = \frac{n^2 - n}{2}$, $g(n) = 6n$

f(n) ≤ cg(n) for n > n_0, c > 0, n_0 ≥ 1

= $\frac{n^2 - n}{2}$ ≤ c·6n

Taking the dominant term in f(n) gives us $n^2/2$ while the dominant term in g(n) is 6n. Since $n^2/2$ grows asymptotically faster compared to 6(n), there is no constant c that can satisfy this inequality for all sufficiently large n. Therefore, **f(n) ≠ O(g(n)).**

g(n) ≤ cf(n) for n > n_0, c > 0, n_0 ≥ 1

= 6(n) ≤ c·(n^2-n)/2

Since $n^2/2$ grows asymptotically faster than 6(n), there is a constant c that can satisfy this inequality for all sufficiently large n. Therefore, **g(n) = O(f(n)).**

b. $f(n) = n + 2\sqrt{n}$, $g(n) = n^2$

f(n) ≤ cg(n) for n > n_0, c > 0, n_0 ≥ 1

= $n + 2\sqrt{n}$ ≤ c·n^2

Taking the dominant term in g(n) gives us n^2 which grows asymptotically faster compared to $n + 2\sqrt{n}$. We can say that there is a constant c that can satisfy the inequality for all sufficiently large n. Therefore, **f(n) = O(g(n)).**

g(n) ≤ cf(n) for n > n_0, c > 0, n_0 ≥ 1

= n^2 ≤ c·$(n + 2\sqrt{n})$

Since n^2 grows asymptotically faster than $n + 2\sqrt{n}$, there is no constant c that can make this inequality hold for all sufficiently large n. Therefore, **g(n) ≠ O(f(n)).**

c. $f(n) = 4nlogn + n$, $g(n) = \frac{n^2-n}{2}$

$f(n) \leq cg(n)$ for $n > n_0$, $c > 0$, $n_0 \geq 1$

$= 4nlogn \leq c\cdot\frac{n^2-n}{2}$

Taking the dominant term in f(n) gives us 4nlogn while g(n) gives us $n^2/2$. Since $n^2/2$ grows asymptotically faster compared to $4nlogn$, there exists a constant c that can satisfy this inequality for all sufficiently large n. Therefore, **f(n) = O(g(n)).**

$g(n) \leq cf(n)$ for $n > n_0$, $c > 0$, $n_0 \geq 1$

$= \frac{n^2-n}{2} \leq 4nlogn$

The dominant term in g(n) is $n^2/2$, which grows asymptotically faster than $4nlogn$. Therefore, there is no constant c that can make this inequality hold for all sufficiently large n. Therefore, **g(n) ≠ O(f(n)).**

2. (Member 1) Sort the array A = [64, 57, 13, 70, 85, 39, 22, 48] using
   a. Insertion Sort

Iteration 1

57	64	13	70	85	39	22	48

Iteration 2

13	57	64	70	85	39	22	48

Iteration 3

13	57	64	70	85	39	22	48

Iteration 4

13	57	64	70	85	39	22	48

Iteration 5

Member 1: **Andrea Laserna** CMSC 142 - 1 Machine Problem 1

Member 2: **Claire Dane Vincoy**

13	39	57	64	70	85	22	48

Iteration 6

13	22	39	57	64	70	85	48

Iteration 7

13	22	39	48	57	64	70	85

b. Bubble Sort

Iteration 1

64	57	13	70	85	39	22	48

Iteration 1

57	64	13	70	85	39	22	48

Iteration 2

57	13	64	70	85	39	22	48

Iteration 3

57	13	64	70	85	39	22	48

Iteration 4

57	13	64	70	85	39	22	48

Iteration 5

57	13	64	70	39	85	22	48

Iteration 6

57	13	64	70	39	22	85	48

Iteration 7

57	13	64	70	39	22	48	85

Iteration 2

Iteration 1

13	57	64	70	39	22	48	85

Iteration 2

13	57	64	70	39	22	48	85

Iteration 3

13	57	64	70	39	22	48	85

Iteration 4

13	57	64	39	70	22	48	85

Iteration 5

13	57	64	39	22	70	48	85

Iteration 6

13	57	64	39	22	48	70	85

Iteration 7

13	57	64	39	22	48	70	85

Iteration 3

Iteration 1

13	57	64	39	22	48	70	85

Iteration 2

13	57	64	39	22	48	70	85

Iteration 3

13	57	39	64	22	48	70	85

Iteration 4

13	57	39	22	64	48	70	85

Member 1: **Andrea Laserna** CMSC 142 - 1 Machine Problem 1

Member 2: **Claire Dane Vincoy**

Iteration 5

13	57	39	22	48	64	70	85

Iteration 6

13	57	39	22	48	64	70	85

Iteration 7

13	57	39	22	48	64	70	85

Iteration 4

Iteration 1

13	57	39	22	48	64	70	85

Iteration 2

13	39	57	22	48	64	70	85

Iteration 3

13	39	22	57	48	64	70	85

Iteration 4

13	39	22	48	57	64	70	85

Iteration 5

13	39	22	48	57	64	70	85

Iteration 6

13	39	22	48	57	64	70	85

Iteration 7

13	39	22	48	57	64	70	85

Iteration 5

Iteration 1

13	39	22	48	57	64	70	85

Iteration 2

13	22	39	48	57	64	70	85

Iteration 3

13	22	39	48	57	64	70	85

Iteration 4

13	22	39	48	57	64	70	85

Iteration 5

13	22	39	48	57	64	70	85

Iteration 6

13	22	39	48	57	64	70	85

13	22	39	48	57	64	70	85

c. Selection Sort

Iteration 1

64	57	13	70	85	39	22	48
64	57	13	70	85	39	22	48
13	57	64	70	85	39	22	48

Iteration 2

13	57	64	70	85	39	22	48
13	22	64	70	85	39	57	48

Iteration 3

13	22	64	70	85	39	57	48
13	22	39	70	85	64	57	48

Iteration 4

13	22	39	70	85	64	57	48
13	22	39	48	85	64	57	70

Iteration 5

13	22	39	48	85	64	57	70
13	22	39	48	57	64	85	70

Iteration 6

13	22	39	48	57	64	85	70
13	22	39	48	57	64	85	70

Iteration 7

13	22	39	48	57	64	85	70
13	22	39	48	57	64	70	85

d. Merge Sort

64	57	13	70	85	39	22	48

64	57	13	70
85	39	22	48

64	57

13	70

64

57

13

70

Merge

57	64

13	70

13	57	64	70

Member 1: **Andrea Laserna**

Member 2: **Claire Dane Vincoy**

Merge Sort

85	39	22	48

85	39	22	48

Merge

39	85	22	48

22	39	48	85

13	22	39	48	57	64	70	85

e. Quick Sort

64	57	13	70	85	39	22	48
64	57	13	70	85	39	22	48
64	57	13	70	85	39	22	48
13	57	64	70	85	39	22	48
13	57	64	70	85	39	22	48
13	57	64	70	85	39	22	48
13	57	64	70	85	39	22	48
13	39	64	70	85	57	22	48
13	39	22	70	85	57	64	48
13	39	22	48	85	57	64	70

Partition

Member 1: **Andrea Laserna** CMSC 142 - 1 Machine Problem 1

Member 2: **Claire Dane Vincoy**

13	39	22

85	57	64	70

Quicksort

13	39	22

13	39	22

13	39	22

13	22	39

Partition

13	39

13	22	39	48

Quicksort

85	57	64	70

85	57	64	70

57	85	64	70

57	85	64	70

57	64	85	70

57	64	85	70

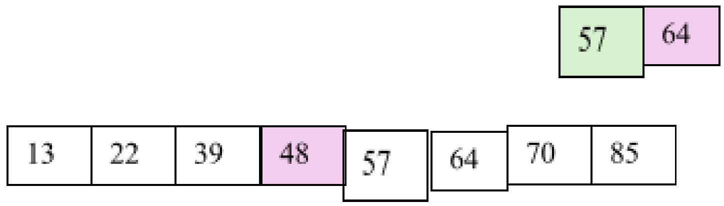
57	64	70	85

Partition

57	64

85v

57	64



3. (Member 2) Sort the array A = [57, 22, 70, 13, 85, 48, 64, 39] using
   a. Insertion Sort
   b. Bubble Sort
   c. Selection Sort
   d. Merge Sort
   e. Quick Sort

4. Summarize the complexities of sorting and searching algorithms in one table:

Algorithm	Best-Case	Scenario / Input Structure for the Best Case	Worst-case	Scenario / Input Structure for the Worst Case
Insertion Sort	O(n)	Sorted Array	O(n^2)	Reverse Sorted Array
Bubble Sort	O(n)	Sorted Array	O(n^2)	Reverse Sorted Array
Selection Sort	O(n^2)	Sorted Array	O(n^2)	Reverse Sorted Array
Merge Sort	O(nlogn)	Sorted Array	O (nlogn)	Reverse Sorted Array
Quick Sort	O(nlogn)	When the chose pivot is the middle or near the middle element	O(n^2)	Pivot is the largest or smallest element for every partition
Heap Sort	O(nlogn)	Building a heap and subsequent removals	O(nlogn)	Maintaining heap during element extraction (index adjustments at each level)
Binary Search	O(1)	Target element is in the central index	O(log n)	When the element is in the first position
Linear Search	O(1)	First element matches target element	O(n)	Target element not in the list or found at the end of the list.

5. The Bubble Sort algorithm is shown below.

```
bubble_sort(array A):
do:
        swapped = False
        for current = N to 1:
                prev = current – 1
                if A[prev] > A[current]:
                        swap A[prev] ↔ A[current]
                        swapped = True
while swapped = True
```

Show the possible modifications in order to optimize the number of timesteps. Also, explain how the changes results to optimization. Note: do not replace the whole code with a new one.

```
bubble_sort(array A):
        i = 1
        do:
                swapped = False
                for current = N to i:
                        prev = current – 1
                        if A[prev] > A[current]:
                                swap A[prev] ↔ A[current]
                                swapped = True
                i++
        while swapped = True
```

A variable i was added to indicate the top of the array in which as the values bubble up, we can ignore those sorted values and only iterate through the remaining values. This is incremented by one after each iteration since after x iterations of the bubble sort, the top x values are sorted. This optimizes the code more because we do not have to perform bubble sort to already sorted values all over again.