Kevin Chen
CS 580U
Professor Jonathan Mwaura
Problem set 1

Problem # 1

a. $f(n)=n^2+2n+3$ $g(n)=n^2$ (Need to find constants c & n_0 such that $0 \le n^2+2n+3 \le cn^2$ for $n \ge 1$)

$n^2+2n+3 \le cn^2$

↓

Subtract $3n^2+2n+3$ from both sides:

$0 \le (c-1)n^2 - 2n - 3$

↓

I'll choose $c=2$, therefore:

$0 \le n^2 - 2n - 3$

↓

Factor right side:

$0 \le (n-3)\cdot(n+1)$ → This inequality holds for $n \ge 3$.

↓

Therefore for $c=2$ and $n_0=3$, we have $0 \le n^2+2n+3 \le 2n^2$ for all $n \ge 3$. Thus $f(n) = O(n^2)$.

For $f(n) \neq O(n)$ proof:

If $f(n) = O(n)$, there should be constants c and n_0 such that $0 \le n^2+2n+3 \le cn$ for all $n \ge n_0$.

Now, looking at the terms using n^2 and n, it's clear that n^2+2n+3 grows faster than cn for any constant c if c and n grow very large. Therefore, $f(n)$ is not in $O(n)$.

Thus: $f(n) = O(n^2)$, but $f(n) \neq O(n)$.

b. $f(n) = n\log n + 100n$ and $g(n) = n\log n$ (need to find c and n_0 such that $0 \le n\log n + 100n \le c \cdot n\log n$ for all $n \ge n_0$.

Assuming $n > 0$, divide inequality by $n\log n$:

$0 \le 1 + 100/\log n \le c$

↓

See the behavior of $100/\log n$: as n becomes larger, the denominator grows slower than numerator, and this expression will approach infinity.

Therefore I'll choose $c = 101$ and $2n_0 = 2$ to satisfy the inequality for $n \ge n_0$. This would imply that $f(n) = O(n\log n)$.

Now, to prove $f(n) \neq O(n)$, if $f(n) = O(n)$, there should be constants 'c' and 'n_0' such that $0 \le n\log n + 100n \le c \cdot n$ for all $n \ge n_0$.

If we analyze the terms involving $n\log n$ and n, however, you can see that $n\log n + 100n$ grows faster than $c \cdot n$ for any constant c when n becomes large. Thus, $f(n)$ is NOT $O(n)$.

We can conclude that $f(n) = O(n\log n)$ and $f(n) \neq O(n)$.

c) Need to find constants $c1$, $c2$, $n1$, $n2$

$0 \le 2n^2+4 \le c1\,(4n^2+2)$ for all $n \ge n1$, to prove $f(n) = O(g(n))$

$0 \le 4n^2+2 \le c2\,(2n^2+4)$ for all $n \ge n2$, to prove $g(n) = O(f(n))$

Prove $f(n) = O(g(n))$:

Find $c1$ and $n1$ that satisfy the condition given $f(n) = 2n^2+4$, and $g(n) = 4n^2+2$.

$0 \leq 2n^2 + 4 \leq c1(4n^2+2)$ for all $n \geq n1$.

$2n^2+4 \leq c1(4n^2+2)$

$2n^2+4 \leq 4c1n^2 + 2c1$

$4 - 2c1 \leq (4c1-2)n^2$

I'll assume $c1 \geq 2$ since $c1$ needs to be a positive constant. We can now choose $c1$ and $n1$ such that $4 - 2c1 \leq 0$ for all $n \geq n1$.

I'll set $c1 = 2$, and $n1 = 1$.

So, $2n^2+4 \leq 2(4n^2+2)$ for all $n \geq 1$. This would prove $f(n) = O(g(n))$.

To prove $g(n) = O(f(n))$, I need to find $c2$ and $n2$ such that $0 \leq 4n^2+2 \leq c2(2n^2+4)$ for all $n \geq n2$.

$4n^2+2 \leq c2(2n^2+4)$

$4n^2+2 \leq 2c2 - 4c2$

$2c2 - 4 \geq 0$ (for simplicity, I assume $c2 \geq 2$)

I'll set $c2 = 2$ and $n2 = 1$.

So, $4n^2+2 \leq 2(2n^2+4)$ for all $n \geq 1$. This proves $g(n) = O(f(n))$.

To conclude: $f(n) = O(g(n))$ and $g(n) = O(f(n))$ for $f(n) = 2n^2+4$ and $g(n) = 4n^2+2$.

d. To prove this, suppose the following two functions:

1. $f(n) = e^n$
2. $g(n) = n$

Prove: $f(n) \geq O(g(n))$

This statement means c and n_0 constants are positive, which means that $0 \leq e^n \leq cn$ for all $n \geq n_0$. It's proved by showing that e^n grows at a slower rate compared to n, meaning e^n is asymptomatically dominated by n.

Consider $\lim_{n\to\infty} \frac{e^n}{n}$. Apply L'Hopital's rule, for any positive constant C, there exists a large n_0 such that $0 \leq e^n \leq cn$ for all $n \geq n_0$. This proves $f(n) = O(g(n))$.

Disprove: $g(n) = O(f(n))$

This states that constants c and n_0 are positive such that $0 \leq n \leq ce^n$ for all $n \geq n_0$. However this is not true.

Consider $\lim_{n\to\infty} \frac{n}{e^n}$. This approaches 0, meaning that n is dominated by e^n. For any positive constant C, the n value is $n > ce^n$. Thus $g(n) \geq O(f(n))$. This would disprove $g(n) = O(f(n))$.

$f(n) = O(g(n))$ is true, and $g(n) = O(f(n))$ is NOT true.

Problem #2

a) Find positive constants such that $0 \leq 2n+1 \leq c \cdot 2^n$ for all $n \geq n_0$

Analyze inequality:

$0 \leq 2n+1 \leq c \cdot 2^n$

↓

Simplify middle term:

$0 \leq 2n+1 \leq c \cdot 2^{n-1}$

↓

I'll choose $c=1$ and $n_0=1$:

$0 \leq 2n+1 \leq 2 \cdot 2^{n-1}$

↓

This holds for all $n \geq 1$, therefore $2n+1 = O(2^n)$.

b) I will disprove, show that for constants c and n_0, there exists $n \geq n_0$ such that $2^{2n} > c \cdot 2^{n^2}$

Consider this limit: $\lim_{n \to \infty} 2^{2n} / 2^{n^2}$

This equals 0, meaning 2^{2n} is dominated by 2^{n^2}. So $2^{2n} = O(2^{n^2})$.

c) I'll compare the growth rates of the function: $\log(\log^k n)$ grows slower than $\log^k(\log n)$. $\log^k n$ is an iterated function, which is very slow. So $\log(\log^k n)$ is slower in comparison to $\log^k(\log n)$, thus $g(n) = \log^k(\log n)$ is asymptotically larger.

d) Upper bound (O): Find constants c and n_0 such that:

$0 \leq \log_3 n \leq c \cdot \log_2 n$ for all $n \geq n_0$.

Next, analyze inequality:

$0 \leq \log_3 n \leq c \cdot \log_2 n$

$0 \leq \log_3 n / \log_2 n \leq c$

↓

I'll choose $c=1$, $n_0=1$ since $\log_3 n / \log_2 n \geq 0$ for $n \geq 1$

$0 \leq \log_3 n \leq \log_2 n$

↓

This holds for all $n \geq 1$, so $f(n) = O(g(n))$.

Lower bound (Ω): Find constants c and n_0 such that $0 \leq c \cdot \log_2 n \leq \log_3 n$ for all $n \geq n_0$

Analyze inequality:

$0 \leq c \cdot \log_2 n \leq \log_3 n$

$0 \leq c \cdot \log_3 n / \log_3 2 \leq \log_3 n$

↓

I'll choose $c = \frac{1}{\log_3 2}$ and $n_0 = 1$, since $\log_3 n / \log_3 2 \geq 0$ for $n \geq 1$:

$0 \leq \log_2 n / \log_3 2 \leq \log_3 n$

↓

This holds for all $n \geq 1$, so $f(n) = \Omega(g(n))$.

Tight bound (Θ): Since both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ hold, we can deduce $f(n) = \Theta(g(n))$

Therefore the relationship between $f(n) = \log_3 n$ and $g(n) = \log_2 n$ is given by $f(n) = \Theta(g(n))$

Question 3:

a. (Submitted on canvas)

```cpp
#include<iostream>
using namespace std;

int countCriticalEvents(int array[], int size, double t) {
    int criticalEvents = 0;
    for (int i = 0; i < size; ++i) {
        for (int j = i + 1; j < size; ++j) {
            if (array[i] > t * array[j]) {
                criticalEvents++;
            }
        }
    }
    return criticalEvents;
}

int main() {
    // Example array and threshold value
    int inputArray[] = { 3, 5, 2, 8, 6 };
    int size = sizeof(inputArray) / sizeof(inputArray[0]);
    double thresholdValue = 0.5;

    cout << "\nArray: ";
    for (int i = 0; i < size; i++)
        cout << inputArray[i] << " ";

    cout << "\nThreshold Value: " << thresholdValue;

    // Count and output the number of critical events
    int result = countCriticalEvents(inputArray, size, thresholdValue);
    std::cout << "\nNumber of critical events: " << result << std::endl;

    return 0;
}
```

```
/tmp/c9j2dFf4aW.o
Array: 3 5 2 8 6
Threshold Value: 0.5
Number of critical events: 6
```

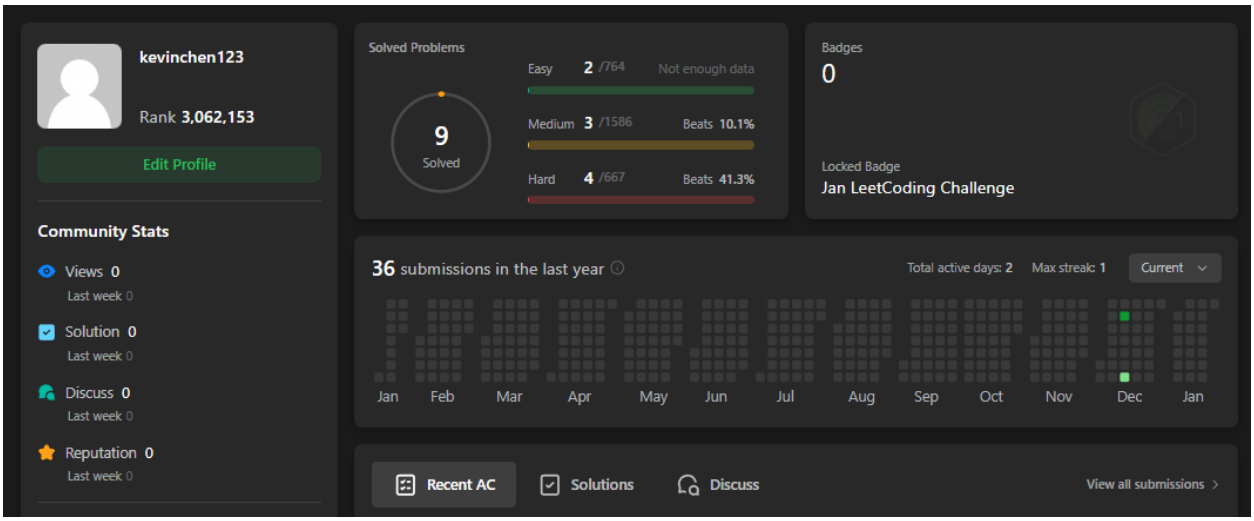
b.

```cpp
#include<iostream>
using namespace std;

int countCriticalEvents(int array[], int size, double t) {
    int criticalEvents = 0;
    for (int i = 0; i < size; ++i) {
        for (int j = i + 1; j < size; ++j) {
            if (array[i] > t * array[j]) {
                criticalEvents++;
            }
        }
    }
    return criticalEvents;
}

int main() {
    // Example array and threshold value
    int inputArray[] = { 5, 4, 1, 3, 1, 7, 9 };
    int size = sizeof(inputArray) / sizeof(inputArray[0]);
    double thresholdValue = 2.7;

    cout << "\nArray: ";
    for (int i = 0; i < size; i++)
        cout << inputArray[i] << " ";

    cout << "\nThreshold Value: " << thresholdValue;

    // Count and output the number of critical events
    int result = countCriticalEvents(inputArray, size, thresholdValue);
    std::cout << "\nNumber of critical events: " << result << std::endl;

    return 0;
}
```

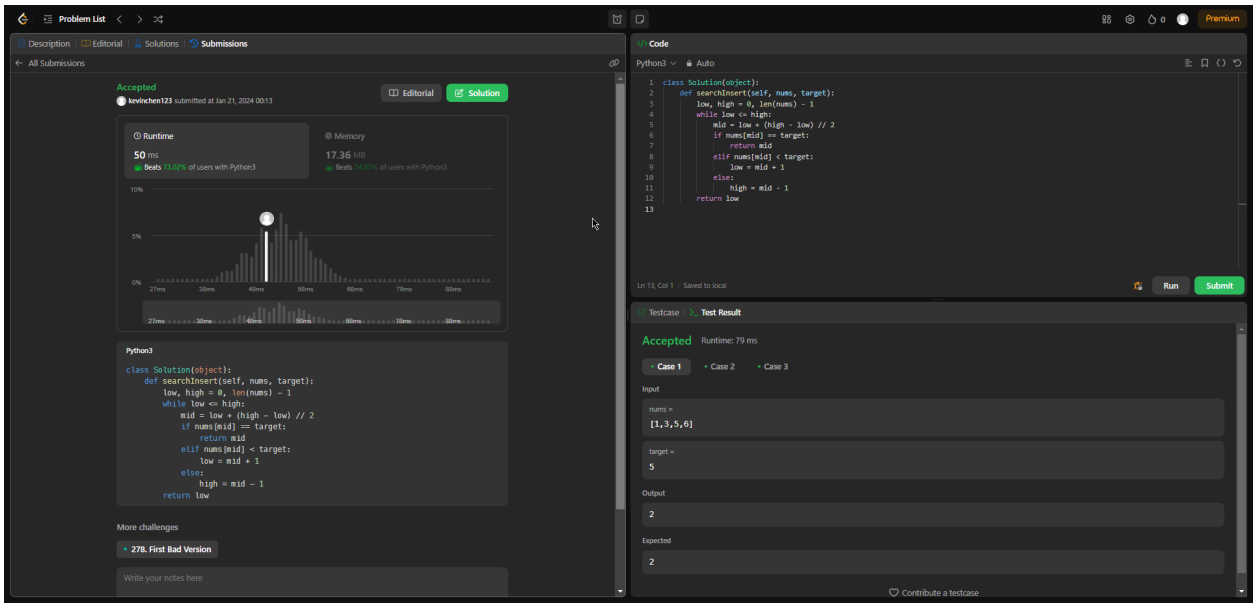
```
/tmp/c9j2dFf4aW.o
Array: 5 4 1 3 1 7 9
Threshold Value: 2.7
Number of critical events: 5
```

c. There are 2 loops running in the function of critical events. Outer loop is running n times, where n is the size of array and inner loop is also running n times. Hence, the total time complexity of the algorithm is $O(n^2)$.

Question 4:

a. 

b.

1.1. 

1.2. For my solution, the operations in the first line are constant-time operations. It works with the while loop employing a binary search strategy, which halves the search space on each iteration. This results in a logarithmic time complexity of O(logn). With each step, half of the remaining elements are removed by adjusting the `low` and/or `high` index based on each comparison between `target` and the middle element of the active search interval. This is an efficient method to narrow down the search range, which is, again the

logarithmic style of the algorithm's complexity. For proof of correctness, this is ensured by my loop invariants maintained throughout the binary search, which are:

1. If `target` value exists in the array, it will always be between the indices `low` and `high`.
2. The `low` index will always point to the first element greater than or equal to `target`, or if `target` is not found, to the insertion point when `high` becomes less than `low`.

So, the loop invariant confirms if `target` is within the array, it will definitely be found. In any other case, `low` indicates the correct insertion position. So the algorithm ensures the correct result in O(logn) time for any input.

1.3. (Submitted on canvas)

Question 5:

a.

array of { 4, 3, 2, 1, 5 } to demonstrate bubble sort:

Iteration # 1

- Compare 4 and 3, swap {3, 4, 2, 1, 5}
- Compare 4 and 2, swap {3, 2, 4, 1, 5}
- Compare 4 and 1, swap {3, 2, 1, 4, 5}
- Compare 1 and 5, no swap {3, 2, 1, 4, 5}

After the first iteration, the largest number (5) is at the end.

Iteration # 2

- Compare 3 and 2, swap {2, 3, 1, 4, 5}
- Compare 1 and 3, swap {2, 1, 3, 4, 5}
- Compare 4 and 3, no swap {2, 1, 3, 4, 5}

After the first iteration, the largest number (4, 5) is at the end.

Iteration # 3

- Compare 1 and 2, swap {1, 2, 3, 4, 5}
- Compare 3 and 2, no swap {1, 2, 3, 4, 5}

After the first iteration, the largest number (3, 4, 5) is at the end.

Iteration # 4

- Compare 1 and 2, no swap {1, 2, 3, 4, 5}

After the first iteration, the largest number (2, 3, 4, 5) is at the end.

Iteration # 5
After the first iteration, the largest number (1, 2, 3, 4, 5) is at the end.

Hence, the array is sorted.

b.

Formula for Comparisons
- For n elements, n-1 comparisons are required.
- For n-1 elements, n-2comparisons are required.
  …
  …
  …
- For 2 elements, 1 comparison is required.
- For 1 element, 0 comparisons are required.

Hence, we can conclude that total number of comparisons can be found by sum of n-1 numbers.

$$C(n) = \frac{n * (n-1)}{2}$$

Formula for Swaps
Given that the array is reversed, as this is worse-case scenario; and the formula for it can be calculated same as C(n).
- For n elements, n-1 comparisons are required.
- For n-1 elements, n-2 comparisons are required.
  …
  …
  …
- For 2 elements, 1 comparison is required.
- For 1 element, 0 comparisons are required.

Hence, we can conclude that total number of comparisons can be found by sum of n-1 numbers.

$$S(n) = \frac{n * (n-1)}{2}$$

c.

Initialization
First, we need to determine whether the invariant is true before the first iteration. With i starting from 0, at the beginning of the 0th iteration, the largest 0 elements of the original list are in the last 0 positions of the list. This is true as it is not promising anything.

Maintenance
The largest i elements of the original list occupy the last i positions in the list and are sorted relative to each other. The inner loop executes from index 1 to len(li)- i - 1 (stopping before the

last i elements). Look at the first two elements and swap them if they are out of order. The larger of them comes second. Again, with the second and third element, leave the maximum of the first three elements in the third position. This continues until the maximum of the first len(i) - i elements is in the len(li) - ith position, or the len(li) - i - 1 index. The largest i elements occupying the last i positions in sorted order is found, but now we also have the largest of the len(li) - i in position len(li) - i - 1, i.e. just before last i elements. It is not bigger than any of the last i elements, and at least as big as the other len(li) - i - 1 elements. So, the last i+1 elements are sorted with respect to each other, and are the largest i+1 elements.

Termination
The outer loop executed len(li) times, but it only did any work len(li) - 1 times. Hence, last len(li) - 1 elements are the largest len(li) - 1 elements and are sorted with respect to each other. That leaves one element left, which is at most the smallest element, and is in position 0. So, the entire list is sorted with respect to itself.

d.

Considering a random permutation, the probability of an element greater than other is ½. Similarly, the probability of swapping is also ½. The probability of swapping I and i+1 elements during a single pass becomes ½ * ½ = ¼.

If an array has n elements, the probability of swapping each pair in an array will be:

¼ (n-1)

Bubble sort makes n-1 passes to fully sort the array, hence the probability of swapping by n-1 number of passes can be given by:

Hence, this is the formula to find the average number of swaps in bubble sort.