Q1

(a)

$f(n)=n^2+2n+3$ and $g(n)=n^2$

We need to find constants c and n_0 such that $0 \le n^2+2n+3 \le cn^2$ for all $n \ge n_0$.

Let's consider the inequality $0 \le n^2+2n+3 \le cn^2$ for $n \ge 1$:

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

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

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

Now, let's choose $c=2$.We get:

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

Factor the right side:

$0 \le (n-3)(n+1)$

This inequality holds for $n \ge 3$.

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

Now, let's prove that $f(n) \ne O(n)$:

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 $n \ge n_0$.

When we look at the terms using n^2 and n, it is clear that n^2+2n+3 grows faster than cn for any constant if c and n becomes very large. Therefore, f(n) is not in O(n).

Hence, $f(n)=O(n^2)$, but $f(n) \ne O(n)$.

(b)

Proof that $f(n)=O(n\log n)$:

As $f(n)=n\log n+100n$ and $g(n)=n\log n$

We need to find c and n_0, such that $0 \le n\log n+100n \le c \cdot n\log n$ for all $n \ge n_0$.

Divide the inequality by $n\log n$ (assuming $n>0$):

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

Now seeing the behavior of 100 / logn, as n becomes large. The denominator grows slower than numerator, hence expression approaches infinity. Therefore, we can choose $c=101$ and $2n_0 = 2$ to satisfy the inequality for $n \ge n_0$. **This implies that $f(n)=O(n\log n)$.**

Proof that $f(n) \ne O(n)$:

If $f(n)=O(n)$, there should be constants 'c' and 'n_0' such that $0 \leq n\log n + 100n \leq c' \cdot n$ for all '$n \geq n0$'.

However, when you analyze the terms involving $n\log n$ and n, it's clear that $n\log n+100n$ grows faster than 'c'$\cdot n$ for any constant 'c' when n becomes large. Therefore, ($f(n)$ is not $O(n)$.

In conclusion, $f(n)=O(n\log n)$, and $f(n) \neq O(n)$.

(c)

To prove that $f(n) = O(g(n))$ and $g(n)=O(f(n))$ for $f(n)=2n^2+4$ and $g(n)=4n^2+2$, we need to find constants $c1,c2,n1,n2$:

1. $0 \leq 2n^2+4 \leq c1(4n^2+2)$ for all $n \geq n1$ (to prove $f(n)=O(g(n))$)
2. $0 \leq 4n^2+2 \leq c2(2n^2+4)$ for all $n \geq n2$ (to prove $g(n)=O(f(n))$)

Proof for $f(n)=O(g(n))$:

Given $f(n)=2n^2+4$ and $g(n)=4n^2+2$, let's find $c1$ and $n1$ that satisfy the condition

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

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

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

$4\text{-}2c1 \leq (4c1-2)n^2$

Let's assume $c1 \geq 2$ ($c1$ needs to be a positive constant).

Now, we can choose $c1$ and $n1$ such that $4-2c1 \leq 0$ for all $n \geq n1$.

Let's set $c1=2$ and $n1=1$.

So, $2n^2+4 \leq 2(4n^2+2)$ for all $n \geq 1$.

This proves that $f(n)=O(g(n))$.

Proof for $g(n)=O(f(n))$:

Let's find $c2$ and $n2$ such that $0 \leq 4n^2+2 \leq c2(2n^2+4)$ for all $n \geq n2$.

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

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

$2c2\text{-}4 \geq 0$(for simplicity, assuming $c2 \geq 2$)

Let's set $c2=2$ and $n2=1$.

So, $4n^2+2 \leq 2(2n^2+4)$ for all $n \geq 1$.

This proves that $g(n)=O(f(n))$.

In conclusion, $f(n)=O(g(n))$ and $g(n)=O(f(n))$ for $f(n)=2n^2+4$ and $g(n)=4n^2+2$.

(d)

To prove this, let's consider the following example. Suppose we have the following two functions:

f(n) = e^n

g(n) = n

Proof: f(n) = O(g(n))

The statement f(n) = O(g(n)), means c and n_0 constants that are positive which means, that 0 <= e^n <= cn for all n >= n_0. It is proved by showing that e^n grows at a slower rate compared to n, which means e^n is asymptotically dominated by n.

Consider the limit: lim$n \to \infty$ e^n / n

By applying L'Hôpital's Rule; for any positive constant c, there exists a large n_0 such that 0 <= e^n <= cn for all n >= n_0.

This proves f(n) = O(g(n)).

Disproof: g(n) = O(f(n))

The statement g(n) = O(f(n)) says that the constants c and n_0 are positive such that 0 <= n <= ce^n for all n >= n_0. But, this is not true.

Consider the limit: lim$n \to \infty$ n / e^n

This limit approaches zero, which means that n is dominated by e^n. For any positive constant c, the value of n is n > ce^n. Hence, g(n) >= O(f(n)).

This disproves g(n) = O(f(n)).

In summary, f(n) = O(g(n)) is true, but g(n) = O(f(n)) is not true.

Q2

(a)

To prove this, find positive constants c and n0 such that

$0 \leq 2n+1 \leq c \cdot 2n$ for all $n \geq n0$.

Analyzing the inequality:

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

Simplify the middle term:

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

Now, we choose c=1 and n0=1:

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

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

(b)

To disprove this, show that for constants c and $n0$, there exists an $n \geq n0$ such that $2^{2n} > c \cdot 2^{n^2}$.

Let's consider the limit:

$\lim_{n \to \infty} 2^{2n} / 2^{n^2}$

This limit is equal to zero, which means 2^{2n} is dominated by 2^{n^2}.

Therefore, $2^{2n} = O(2^{n^2})$.

(c)

To determine which function is asymptotically larger, compare the growth rates of the function. As a result, $\log(\log^k n)$ grows slower than $\log^k(\log n)$.

The notation $\log^k n$ is an iterated logarithm function, which is really slow.Hence, $\log(\log^k n)$ is slower in comparison to $\log^k(\log n)$, and it's found that $g(n)=\log^k(\log n)$ is asymptotically larger.

(d)

Upper Bound (O):

To prove $f(n)=O(g(n))$, find constants c and $n0$ such that

$0 \leq \log_3 n \leq c \cdot \log_2 n$ for all $n \geq n0$.

Let's analyze the inequality:

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

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

Choose $c=1$ and $n0=1$ (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):

To prove $f(n)=\Omega(g(n))$, find constants c and $n0$ such that

$0 \leq c \cdot \log_2 n \leq \log_3 n$ for all $n \geq n0$.

Analyzing the inequality:

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

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

Choose $c = 1 / \log_3 2$ and $n0=1$ (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, it can be found that $f(n)=\Theta(g(n))$.

In summary, the relationship between $f(n)=\log_3 n$ and $g(n)=\log_2 n$ is given by:

$f(n)=\Theta(g(n))$

Q3

(a)

```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;

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

    return 0;
}
```

(b)

```
Array: 3 5 2 8 6
Threshold Value: 0.5
Number of critical events: 6
```

(c)

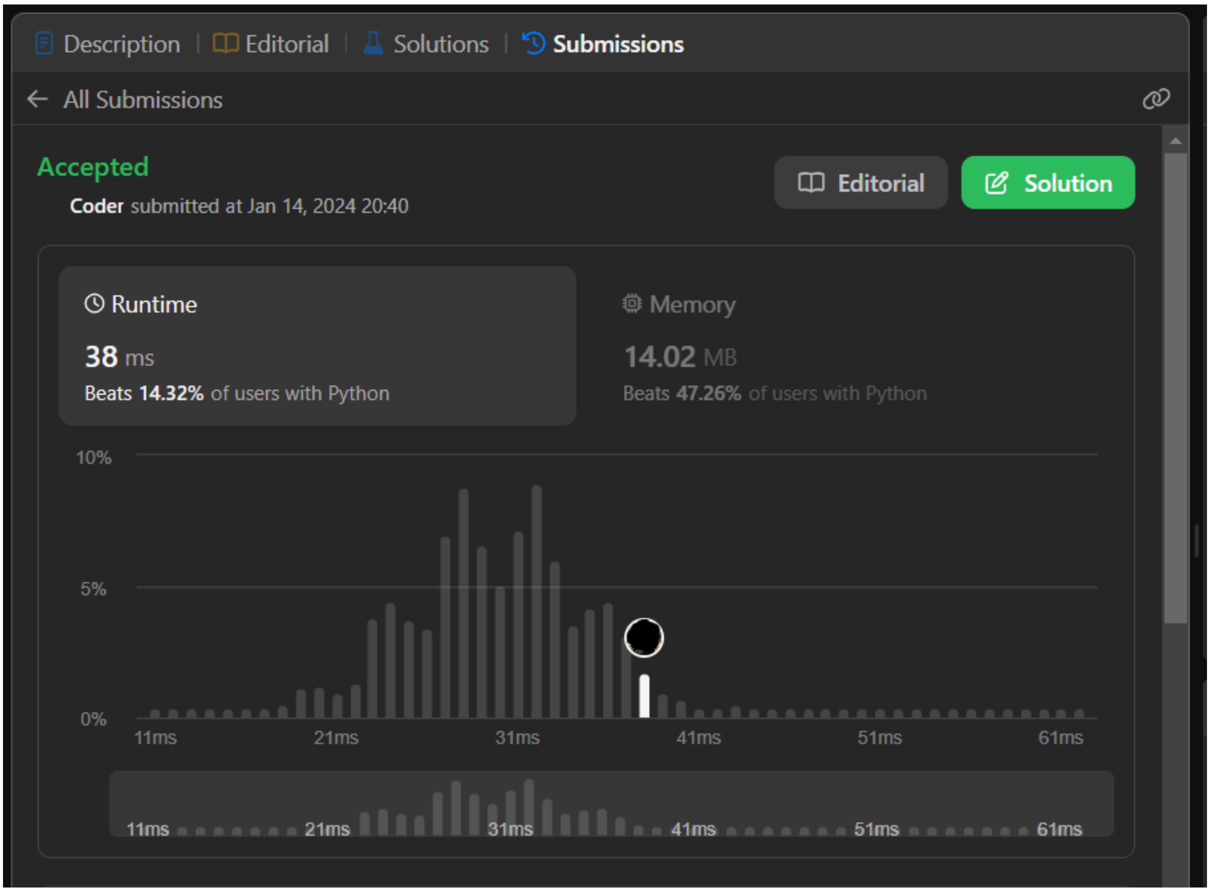
There are 2 loops running in the function of critical events. Outer loop is running n times, where n is the

```
Array: 5 4 1 3 1 7 9
Threshold Value: 2.7
Number of critical events: 5
```

size of array and inner loop is also running n times. Hence, the total time complexity of the algorithm is $O(n^2)$.

Q4

(a)



(b)

The operations in the first line are constant operations. The while loop runs as long as *low* variable is less than or equal to *high* variable. In while loop, binary search algorithm is applied, which has O(logn) complexity. Due to this, the time complexity of the function is O(logn).

I have used binary search approach to make the algorithm in logn complexity.

(c)

```
class Solution(object):
    def searchInsert(self, nums, target):
        low, high = 0, len(nums) - 1

        while low <= high:
            mid = low + (high - low) // 2

            if nums[mid] == target:
                return mid
            elif nums[mid] < target:
                low = mid + 1
            else:
```

```
            high = mid - 1
    return low
```

Q5

(a)

We will use array of { 4, 3, 2, 1, 5 } for demonstration of 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, array is sorted.

(b)

Formula for Comparisons

- 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 comparison is required.

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

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

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 comparison is required.

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

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

(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 eachother. 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 eachother, 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:

$$^{1}/_{4}\,(n-1)(n-1)$$

$$= \,^{1}/_{4}\,(n-1)^2$$

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