**Problem 1**

Consider the following algorithm that uses brute force maximum independent set algorithm on each connected component:

**Algorithm 1** Maximum Independent Set Algorithm

1: Find all connected components in the input graph \( G \)
2: Set \( I^* \) to be an empty set
3: for each connected component \( C \) do
4: Solve the maximum independent set problem on \( C \) and let \( I \) be the solution obtained
5: Set \( I^* \) to be \( I^* \cup I \)
6: end for
7: Return \( I^* \)

There are at most \( n \) connected components and so the for-loop executes at most \( n \) times. As discussed in class, each call to the algorithm for solving the maximum independent set problem runs in time \( O(t^2 \cdot 2^t) \), where \( t \) is the number of vertices in the connected component \( C \). Since \( t = O(\log n) \), this running time is \( O(\log^2 n) \cdot 2^{O(\log n)} \). Since \( 2^{O(\log n)} \) is polynomial in \( n \), the overall running time of Line 3-5 (for loop) is a polynomial in \( n \). Since the for-loop executes \( n \) times, the total running time of Line 2-6 is still polynomial in \( n \). The first line on the other hand, can be easily implemented via graph search algorithms like breadth-first search (BFS). Since BFS runs in polynomial time, we can claim the algorithm runs in polynomial time overall.

**Problem 2**

The strategy with 2 jars to get \( f(n) = \sqrt{n} \) can be described as follows:

Take the first jar and drop it from rung \( \lceil \sqrt{n} \rceil \), and if it survives, try rung \( 2\lceil \sqrt{n} \rceil \). In this manner, we continue to raise the rung level by \( \lceil \sqrt{n} \rceil \) in each iteration that the jar survives. This can continue for a maximum of \( n/\lceil \sqrt{n} \rceil \leq \lceil \sqrt{n} \rceil \) steps.

If the jar breaks in one of the drops at rung \( k\lceil \sqrt{n} \rceil \), we know that the highest safe rung must be between \((k-1)\lceil \sqrt{n} \rceil \) and \( k\lceil \sqrt{n} \rceil \). We can then start dropping the second jar from rung \((k-1)\lceil \sqrt{n} \rceil + 1\) and move up one rung up at a time. Thus the maximum number of drops for the second jar to find the highest safe rung will be \( \lceil \sqrt{n} \rceil \) as well. The overall running time should be \( 2\lceil \sqrt{n} \rceil = O(\sqrt{n}) \).

**Problem 3**

(a) We cannot have both factors of \( n \) greater than \( \sqrt{n} \) because then the resulting number will be greater than \( n \). Thus it is guaranteed that if \( n \) has a factor (besides 1 and \( n \)) then at least one such factor must be at most \( \sqrt{n} \). Thus \( 2 \to \sqrt{n} \) is a sufficient range to examine for factors of \( n \).

(b) The worst case happens when \( n \) is a prime number so that none of the numbers in range 2 to \( \sqrt{n} \) can evenly divide \( n \). The worst case running time in this case should be \( \Theta(\sqrt{n}) \)

(c) Since \( n \) can be arbitrarily large, we pay attention to its size. Its size is determined by its representation and typically we use binary representation and will need \( \lfloor \log_2 n \rfloor \) bits to represent \( n \). If we let \( s \) denote \( \lfloor \log_2 n \rfloor \), we see that \( n = \Theta(2^s) \). Thus the worst case running time for the naive primality testing algorithm would be (based on part (b)), \( \Theta(2^s) = \Theta(2^{s/2}) \).

(d) Implementation is omitted here. 758500183202087890352073067 is not a prime number. It can be factored as:

\[
758500183202087890352073067 = 838041641 \times 941083981 \times 961748927
\]
Problem 4

(a) \( \sum_{i=1}^{n} i = \Theta(n^2) \).

(b) The running time of {\tt mergeSort} is \( \Theta(n \cdot \log n) \).

(c) We can rewrite

\[
(\log_2 n)^{\log_2 n} = (2^{\log_2 \log_2 n})^{\log_2 n} = (2^{\log_2 n \cdot \log_2 n})^{\log_2 n} = n^{\log_2 \log_2 n}.
\]

(e) \( \log_2 128 = 7 \) and so \( (\log_2 128)^n = 7^n \).

(f) The Maximum independent Set problem can be solved by a brute force algorithm (discussed in class) with running time \( \Theta(n^2 \cdot 2^n) \).

(g) According to the sum of geometric series, we have

\[
\sum_{i=0}^{n} i^2 = \Theta(n^3).
\]

In ascending order of growth we have:

\[
(h) 7^n, \quad (g) \sum_{i=0}^{n} \frac{n^2}{2}, \quad (b) \Theta(n \log n), \quad (a) \Theta(n^2), \quad (c) (\log_2 n)^{\log_2 n} = (d) n^{\log_2 (\log_2 n)}, \quad \Theta(n^2 \cdot 2^n), \quad 7^n
\]

Problem 5

Consider the following algorithm: This algorithm searches through the list based on absolute values. In order

\begin{algorithm}
1: Set \( i = 0 \)
2: Set \( j = L.length - 1 \)
3: while \( i < j \) do
4: if \( \text{abs}(L[i]) < \text{abs}(L[j]) \) then
5: \quad Set \( j = j - 1 \)
6: else if \( \text{abs}(L[i]) > \text{abs}(L[j]) \) then
7: \quad Set \( i = i + 1 \)
8: else
9: \quad if \( L[i] + L[j] = 0 \) then
10: \qquad Return True
11: \quad end if
12: end if
13: end while
14: Return False
\end{algorithm}

to have two numbers sum up to zero, they must have the same absolute value. Since the list is sorted, we know that the absolute values will decrease and then increase if 0 is one of the elements. Then by comparing absolute values we will find the first pair with the same absolute value if such pairs exist. The running time of this algorithm is \( O(n) \) since each step one of the indices will move by 1 or the algorithm terminates. The maximum steps the indices can run is \( n \). On the other hand, if all the items in the list are positive or negative, there won’t be such a pair existing and the algorithm will loop through one of the indices and return False.
Problem 6

(a) This algorithm runs in constant time $O(1)$

(b) Even for a positive list, with 90% or more positive numbers, it is still possible that we randomly sample at least 2 negative items from the list. The algorithm will under such circumstances give an output of negative for a positive list and vice versa.

(c) We are assuming random selection, and so

$$P(A) \leq 0.1 \cdot 0.1 \cdot 1 = 0.01.$$  

Similarly, $P(B)$ and $P(C)$ are also bounded above by 0.01. On the other hand, for case D:

$$P(D) = 0.1 \cdot 0.1 \cdot 0.1 = 0.001$$

Given $L$ is positive, at most there is probability of:

$$P(\text{negative}) = P(A) + P(B) + P(C) + P(D) \leq 0.011$$

such that the algorithm will output negative even though $L$ is positive.