1. Prove the following formula using induction for all \( n \geq 0 \). Be clear about what your base case, your induction hypothesis, and inductive step are (label them).

\[
\sum_{i=0}^{n} 2^i = 2^{i+1} - 1
\]

**Solution:**

We prove this using (weak) induction on \( n \).

**Base case:** \( n = 0 \). Then we must show that:

\[
\sum_{i=0}^{0} 2^i = 2^1 - 1
\]

The left hand side is exactly \( 2^0 = 1 \) and therefore is equal to the right hand side.

**Inductive step:** Let \( n \) be a non-negative integer.

**Induction hypothesis:** Suppose that the hypothesis is true for \( n \):

\[
\sum_{i=0}^{n} 2^i = 2^{n+1} - 1
\]

We must show that hypothesis holds for \( n + 1 \):

\[
\sum_{i=0}^{n+1} 2^i = 2^{(n+1)+1} - 1
\]

Considering *only* the left hand side:

\[
\sum_{i=0}^{n+1} 2^i = \left(\sum_{i=0}^{n} 2^i\right) + 2^{n+1}
\]

which by the induction hypothesis is:

\[
2^{n+1} - 1 + 2^{n+1}
\]

and after combining terms:

\[
2 \times 2^{n+1} - 1 = 2^{n+2} - 1 = 2^{(n+1)+1} - 1
\]

as desired.

\[\blacksquare\]

2. Suppose you meet someone that is supposedly very generous. They have unlimited money and would like to offer you some under two conditions: 1) they keep their money in packages of $4 and $7 only for convenience 2) while they will always offer you at least $20, they will consider it very rude if you do not take the full amount they offer and take the money back as a result. Are they really generous i.e. is it always possible to take the amount they offer using only their packages of $4 and $7? Prove your answer. *Hint: Look at the proof with strong induction on slide 27.*
Solution: Yes, they are generous. We prove this using strong induction for any amount of money \( n \geq 20 \) (since they will always offer at least $20).

**Base cases:**

- \( n = 20 \) can be formed by 5 $4 packages.
- \( n = 21 \) can be formed by 3 $7 packages.
- \( n = 22 \) can be formed by 2 $7 packages and 2 $4 packages.
- \( n = 23 \) can be formed by 1 $7 package and 4 $4 packages.

**Inductive step:** Let \( i \geq 20 \) be a non-negative integer.

**Induction hypothesis:** Suppose for all \( 20 \leq k \leq i \) that our hypothesis is true, that is, that we can form the amount of money offered exactly using only $4 and $7 packages.

We must show that after the an offer of \( i + 1 \). Suppose \( 20 \leq i + 1 \leq 23 \). Then, we’ve established by the base case that the hypothesis holds. Otherwise, consider \( j = i + 1 - 4 \). Then, it must be that \( 20 \leq j < i \). By the (strong) induction hypothesis, we must be able to form an offer of \( j \) dollars using some number of $4 and $7 packages. If we add one more $4 package, we will have formed \( i + 1 \) dollars. Therefore, the hypothesis holds for \( i + 1 \), as desired.

3. The pseudocode for BubbleSort is reproduced below:

```plaintext
procedure BubbleSort(a1, a2, . . . , an; a list of distinct real numbers with n ≥ 2)
1. for i := 1 to n − 1
2. for j := 1 to n − i
3. if aj > aj+1 then
4. interchange aj and aj+1

We prove the following loop invariant for BubbleSort:

After the \( k^{th} \) iteration of the outer loop, the last \( k \) elements of the list contain exactly the \( k \) largest elements of the set \{a1, a2, . . . , an\}, in sorted order.

**Solution:** We prove this by induction on \( k \), the number of outer loop iterations. A useful visualization can be seen in the following:

\( k = 0 \): (before we enter the for loop for the first time)

```
\begin{array}{c|c}
\hline
\text{big numbers (unsorted)} & \text{big numbers (unsorted)}
\end{array}
```

\( k - 1 \Rightarrow k \):

```
\begin{array}{c|c|c|c}
\hline
\text{small numbers (unsorted)} & \text{big numbers (sorted)} & \text{big numbers (sorted)} & \text{big numbers (sorted)}
\end{array}
```

**Base case:** \( k = 0 \). Before we run the outer loop, the last 0 positions contain the 0 largest elements in sorted order. This is vacuously true because the last 0 positions constitute an empty set and any conclusion can be drawn from an universal conditional over it.

**Inductive step:** Let \( k \) be a positive integer.
**Induction hypothesis:** Suppose that after the \((k - 1)^{th}\) time through the outer loop, positions \(a_{n-k+2}, \ldots, a_n\) contain the \(k-1\) largest elements of the input, in sorted order.

We must show that after the \(k^{th}\) time through the outer loop, positions \(a_{n-k+1}, \ldots, a_n\) contain the \(k\) largest elements of the input, in sorted order. During the \(k^{th}\) iteration of the outer loop, \(i = k\) and the algorithm finds the maximum value among the first \(n - i + 1\) elements of the list and places it in position \(a_{n-i}\). We can see this because the inner loop compares consecutive elements (line 3) and swaps these as necessary so that the larger element of the pair ends up further to the right. Therefore, the maximum value among the first \(n - i + 1\) elements will, once encountered, be repeatedly swapped to the position corresponding to the largest value of \(j + 1\). We see that this is when \(j = n - i\) so that \(j + 1 = n - i + 1\). Thus after the \(k^{th}\) iteration of the outer loop, position \(a_{n-k+1}\) contains the maximum of the first \(n - i + 1 = n - k + 1\) elements, yet, by our induction hypothesis, must additionally be no larger than the final \(k - 1\) elements. Therefore, \(a_{n-k+1}\) contains the \(k^{th}\) largest element of our input, as desired. Applying the induction hypothesis, this now means that the last \(k\) elements contain exactly the \(k\) largest elements of the set \(\{a_1, a_2, \ldots, a_n\}\) in sorted order.

4. The pseudocode for InsertionSort is reproduced below:

```pseudo
class InsertionSort:
1. for \(j := 2\) to \(n\):
2.     \(i := 1\)
3.     while \(a_j > a_i\):
4.         \(i := i + 1\)
5.     \(m := a_j\)
6.     for \(k := 0\) to \(j - i - 1\):
7.         \(a_{j-k} := a_{j-k-1}\)
8.     \(a_i := m\)
```

We prove the following loop invariant for InsertionSort:

After the \(l^{th}\) iteration of the outer loop, the first \(l + 1\) elements of the list are the first \(l + 1\) elements from the input in sorted order.

**Solution:** We prove this by induction on \(l\), the number of outer loop iterations. A useful visualization can be seen in the following:

- \(l = 1\); (before we enter the for loop for the first time)
  
  ![Diagram](image)

- \(l-1 \rightarrow l\)
  
  ![Diagram](image)

- \(l+1 \rightarrow l\)
  
  ![Diagram](image)
**Base case:** \( l = 0. \) When \( l = 0, \) before the outer loop has never executed, the first position of the list contains only one element. A single element list is always sorted.

**Inductive step:** Let \( l \) be a positive integer.

**Induction hypothesis:** Suppose that after the \((l - 1)\)th time through the outer loop, positions \( a_1, \ldots, a_l \) contain the first \( l \) elements of the input in sorted order.

We must show that after the \( l \)th time through the outer loop, positions \( a_1, \ldots, a_{l+1} \) contain the first \( l + 1 \) elements of the input in sorted order. During the \( l \)th iteration of the outer loop, \( j = l + 1 \) the while loop executes until either \( i = j \) or it encounters the first \( i < j \) such that \( a_j \leq a_i \). Following this, we shift the the elements at positions \( i \) through \( j - 1 \) to the right one position. Then, we move the saved value of \( a_j \) to \( a_i \). Therefore, the final position of \( a_j \) after the \( l \)th iteration of the outer loop must mean it is strictly larger than all elements at positions preceeding it and no larger than the element immediately following it. Since shifting elements in a list does not affect the ordering, along with our induction hypothesis, it must also be true that it is no larger than any of the elements following it up until position \( j \). Therefore, the first \( j + 1 = l \) elements of the list now contain the first \( l \) elements of the original input in sorted order, as desired.

\[\blacksquare\]

5.

(a) **Solution:** procedure MinSortThird\((a_1, a_2, \ldots, a_n: \) a list of real numbers with \( n \geq 2)\)

1. for \( k := 1 \) to \( n - 1 \) by 3
2. \( m := a_k \)
3. \( i := k \)
4. for \( j := k + 3 \) to \( n \) by 3
5. if \( a_j < m \) then
6. \( m := a_j \)
7. \( i := j \)
8. \( a_i := a_k \)
9. \( a_k := m \)

(b) **Solution:** This is the same analysis as MinSort but with \( n' = \lfloor n/3 \rfloor \). That is, we have \( O((n/3)^2) = O(n^2) \).

6. Give the number of comparisons that will be performed by each sorting algorithm if the input array of length \( n \) happens to be of the form \([1, 2, \ldots, n-3, n-2, n, n-1]\) (i.e., sorted except for the last two elements).

On the real exam, you would be given pseudocode for the algorithms, though it is a very good idea to be comfortable with how the algorithms work to save time on the exam. For now, you can refer to the textbook for pseudocode.

(a) **MinSort (SelectionSort)**

**Solution:** \( \frac{n(n-1)}{2} \)

MinSort does the same amount of work on all inputs. It does \( n - 1 \) comparisons to find the minimum of the list before swapping it to the front. Then it does \( n - 2 \) comparisons on the next pass to find the minimum of the remaining elements, and so on, until it eventually does 1 comparison to find the minimum of the last remaining two elements. The total number of comparisons is \( 1 + 2 + \cdots + (n - 1) \).

(b) **BubbleSort**

**Solution:** \( \frac{n(n-1)}{2} \)

BubbleSort does the same amount of work on all inputs. It does \( n - 1 \) comparisons on the first pass, then \( n - 2 \) comparisons on the next pass, and so on. The total number of comparisons is \( 1 + 2 + \cdots + (n - 1) \).
(c) InsertionSort

Solution: \(\frac{n^2+n-2}{2}\)

Using the lecture’s implementation, on this input, InsertionSort will do one comparison when \(j = 1\). It will do two comparisons when \(j = 2\) (comparing \(a_2\) with \(a_1\) and then with itself). It will then do three comparisons when \(j = 3\), and so on, up to \(n - 1\) comparisons when \(j = n - 1\). This is because each new number \(a_j\) being inserted into the sorted list is greater than all the numbers that have been inserted before it, and InsertionSort compares from the beginning of the list each time. Finally, when \(j = n\), InsertionSort will do \(n - 1\) comparisons since \(a_n\) is bigger than all but one of the elements that have been inserted before it. It takes \(n - 1\) comparisons (comparing \(a_n\) with all the other elements except itself) in order to determine to place \(a_n\) in the second to last position. The total number of comparisons is \(1 + 2 + 3 + \cdots + (n-1) + (n-1)\). We can simplify this sum to \(\frac{n^2+n-2}{2}\). Note that this is just one less comparison than the worst case of InsertionSort, which would be if all the list elements were in sorted order. The analysis for the worst case of InsertionSort is done in Example 6 on page 222 of Rosen. On this input for Q2’s implementation, it will perform one comparison for the first \(n - 2\) iterations because of the loop condition breaks in each of the first iterations. Only for the last iteration it performs 2 comparisons, so the total comparisons are \(n\) times.

7. Some more properties of MinSort:

(a) While we often talk about the number of comparisons an algorithm makes, it is useful to know the number of assignments (\(\leftarrow\) or :=) made. What is the worst-case number of assignments MinSort will make? State and justify the general form of an input leading to this.

(b) Can we improve the running time in certain cases for MinSort? Explain how and calculate the worst-case time this might add to the algorithm.

(c) As stated in lecture, lower (and upper bounds) are usually not required to be “tight” e.g. for any non-trivial algorithm a lower bound is 1. Perform the lower bound analysis on Slide 91 using \(n/3\) and come up with a new lower bound for MinSort. Run your lower bound on the input \([5, 3, 8, 1, 4, 9]\) and show your calculations.

Solution:

1. Lines 2-3 and 8-9 always make 4 assignments total. Considering the inner loop, we only make an assignment (two, in fact) when we find an element smaller than the current minimum. Therefore, an input that is sorted in reverse will cause inner loop to the conditional on line 5 true \(n - 2(i - 1) - 1\) (or \(n - 2(k - 1) - 1\) from lecture slides) times. Therefore, there would be \(4(n-1) + 2\sum_{i=1}^{n-1} (n - 2(i - 1) - 1)\) assignments.

2. Note that MinSort (without modification) still runs in \(\Omega(n^2)\) even on a sorted list. We could add an additional step before the start of each outer loop to check if the list is already sorted using (at most) \(n\) comparisons. Therefore, on a sorted list, we would exit early. However, on an input that is “almost” sorted e.g. all but the last element are in the correct order, we would incur an additional time cost of \(n - 1\) for each outer loop iteration.

3. Suppose we perform a similar calculation by considering only the first \(\frac{n}{3}\) iterations of the outer loop. Then, the inner loop considers always at least the last \(\frac{2n}{3}\) elements of the list. A similar analysis shows we run the inner loop at least \(\frac{n}{3} \times \frac{2n}{3} = \frac{2n^2}{9}\) over the course of the algorithm. This is slightly looser than the one with \(\frac{n}{2}\).

Example: Since in the given example \(n = 6\), we only consider the first \(6/3 = 2\) times through the loop.

For the first iteration, we find that the minimum element in the last 5 elements of the list to be 1. Therefore, our list becomes \([1, 3, 8, 5, 4, 9]\).

On the second iteration, we find the minimum element in the last 4 elements of the list to be 3. Thus, our list does not change in this iteration.
8. Technically, both algorithms solve the sorting problem. On each iteration, SortA selects the smallest element in the list (lines 5 through 8, “if \( a_j < \text{item} \)”) and positions it in the sorted list (lines 9 and 10). So on the \( t_{th} \) iteration of sort A, we are finding the \( t_{th} \) smallest element and moving it to position \( t \). Therefore, if we do this \( n - 1 \) times in total (line 1, “for \( i := 1 \) to \( n - 1 \)”), all \( n \) elements will be in sorted order.

On the other hand, on each iteration, SortB selects the largest element in the list (lines 5 through 8, “if \( a_j > \text{item} \)”) and positions it in the sorted list (lines 9 and 10). On the \( t_{th} \) iteration of sortB, we are finding the \( t_{th} \) largest element and moving it to the \( t_{th} \) to last position, so if we do this \( n - 1 \) times in total (line 1, “for \( k := 1 \) to \( n - 1 \)”), all \( n \) elements will be in sorted order.

Both algorithms are variants of Selection Sort as they pick one element at a time (smallest for SortA, largest for SortB) and places them in such a way that after the \( t \)-th iteration, \( t \) of the elements are in the correct place (the smallest \( t \) elements for SortA, and the largest \( t \) elements for SortB). Since they apply analogous strategies, SortA might be called MinSort, while SortB might be called MaxSort.

9. Suppose we start with the list of real numbers 2, 7, 5, 6, 2, 4 and run one of these algorithms to sort it. We stop the program after 3 iterations of the outer loop, and we see that the list now looks like 2, 5, 6, 7, 2, 4.

(a) Which algorithm(s) could have been used to sort the list?

**Solution:** InsertionSortA or InsertionSortC.

InsertionSortA inserts elements from the beginning, so the first part of the list is always sorted, and each iteration is searching linearly through the sorted part from beginning to end until it finds where to insert element \( a_j \).

InsertionSortB inserts elements from the end, so the last part of the list is always sorted, and each element is searching linearly through the sorted part from end to beginning until it finds where to insert element \( a_j \).

InsertionSortC is like InsertionSortA in that it maintains that the first part of the list is sorted, but it determines where to place \( a_j \) using a binary search.

Since the given output after three iterations has the first part of the list sorted, we know it could have come from InsertionSortA or InsertionSortB, but not InsertionSortB, because that sorts from the back and the list 2, 5, 6, 7, 2, 4 is unsorted towards the end.

(b) For each algorithm given in part (a), how many comparisons between list elements were performed by the algorithm at the time we stopped the program? Justify all your answers by referring specifically to the pseudocode.

**Solution:** At the end of three iterations, InsertionSortA makes 7 comparisons between the list elements (line 3) while InsertionSortC makes 5 comparisons between list elements (line 6).

**InsertionSortA:**
- 2 comparisons in first iteration (\( a_1 \) to \( a_2 \), \( a_2 \) to \( a_3 \))
- 2 comparisons in second iteration (\( a_1 \) to \( a_3 \), \( a_2 \) to \( a_3 \))
- 3 comparisons in third iteration (\( a_1 \) to \( a_4 \), \( a_2 \) to \( a_4 \), \( a_3 \) to \( a_4 \))

**InsertionSortC:**
- 1 comparison in first iteration (\( a_1 \) to \( a_2 \))
- 2 comparisons in second iteration (\( a_2 \) to \( a_3 \), \( a_1 \) to \( a_3 \))
- 2 comparisons in third iteration (\( a_2 \) to \( a_4 \), \( a_3 \) to \( a_4 \))
10. When finding the maximum and minimum values in a list, as with sorting a list, we measure the cost as the number of comparisons between list elements.

(a) **Solution:** You must do \( n - 1 \) comparisons, since you have to compare each element except the first to the current maximum. Since there are \( n - 1 \) elements besides the first, doing any less than \( n - 1 \) comparisons would not be enough to compare each element to the current maximum.

(b) **Solution:** Similarly, you must do \( n - 1 \) comparisons.

11.

(a) **Solution:** After the \( t \)-th iteration of the while loop, we know \( x \) is not equal to any of the last \( t \) entries of list \( a \).

(b) **Solution:** The base case is when \( t = 0 \), before the loop ever iterates. Trivially \( x \) is not in the last 0 entries of the list \( a \), because \( x \) is not in the empty set.

For the induction step, let \( t \) be a positive integer. Suppose that we have gone through the loop \( t - 1 \) times, and that \( x \) is not equal to any of the last \( t - 1 \) entries of \( a \). On the \( t \)-th iteration of the loop, the value of \( i \) is \( n - t + 1 \). The loop condition on the \( t \)-th iteration says that \( n - t + 1 \geq 1 \) and \( x \neq a_{n-t+1} \).

If we get through the \( t \)-th iteration of the loop, we must have met the loop condition. So after the \( t \)-th iteration of the loop, we can be sure that \( a_{n-t+1} \neq x \). Since we know from the inductive hypothesis that \( x \) is not equal to any of the last \( t - 1 \) entries of \( a \), which are \( a_{n-t+2}, \ldots, a_n \) and we have just checked that \( x \) is not equal to the \( a_{n-t+1} \), we conclude that \( x \) is not equal to any of the last \( t \) entries of \( a \), which are \( a_{n-t+1}, a_{n-t+2}, \ldots, a_n \). This proves the inductive step, and so we conclude that the loop invariant is true for all \( t \geq 0 \) as desired.

(c) **Solution:** If we go through the while loop \( t \) times in total throughout our algorithm, the value of \( i \) at the end of the while loop will be \( i = n - t \), since \( i \) starts at \( n \) and is decremented \( t \) times. If at the end of the algorithm, \( i = 0 \), where \( n \) is the size of \( a \), the loop invariant says exactly that \( x \) is not in \( \{a_1, \ldots, a_n\} \), which means we correctly return 0 in line 4. If \( i \geq 1 \), that means we broke out of the while loop because \( a_i = x \), not because \( i < 1 \). Thus, we have found an occurrence of the target \( x \) at position \( i \), and we correctly return \( i \) in line 4.
12.

(a) State a loop invariant that can be used to show the algorithm `AverageInRange` is correct.

**Solution:** After \( t \) times through the for loop, \( \text{sum} \) is the total salary of those people among \( a_1, \ldots, a_t \) who make between \( L \) and \( H \), and \( N \) is the number of such people.

(b) Prove your loop invariant from part (a).

**Solution:** For the base case, let \( t = 0 \). Before the loop starts, \( \text{sum} \) and \( N \) are both 0. This is correct because there are no people among \( a_1, \ldots, a_t \), as it is the empty set, so \( N \) should be 0, and we said that when there are no employees in the range, their average salary is 0.

Now suppose that after \( t \) times through the for loop, \( \text{sum} \) is the total salary of those people among \( a_1, \ldots, a_t \) who make between \( L \) and \( H \), and \( N \) is the number of such people.

On the \( t + 1 \)st time through the loop, \( i = t + 1 \).

Case 1:
If \( a_{t+1} \) falls in the range from \( L \) to \( H \), then we increment \( \text{sum} \) by \( a_{t+1} \) and \( N \) by 1. Thus, if the value of \( \text{sum} \) before this iteration was the total salary of those people among \( a_1, \ldots, a_t \) who make between \( L \) and \( H \), now it is the total salary of those people among \( a_1, \ldots, a_{t+1} \) who make between \( L \) and \( H \). Similarly, if \( N \) before this iteration counted the total number of people among \( a_1, \ldots, a_t \) with salaries in the range of \( L \) to \( H \) and we increased it by one, now \( N \) counts the number of such people among \( a_1, \ldots, a_{t+1} \).

Case 2:
If \( a_{t+1} \) does not fall in the range of \( L \) to \( H \), the algorithm does nothing. In this case, the total salary of those people among \( a_1, \ldots, a_t \) who make between \( L \) and \( H \) is the same as the total salary of those people among \( a_1, \ldots, a_{t+1} \) who make between \( L \) and \( H \). Similarly, the number of such people among \( a_1, \ldots, a_t \) is the same as the number of such people among \( a_1, \ldots, a_{t+1} \).

(c) Conclude from the loop invariant that the algorithm `AverageInRange` is correct.

**Solution:** Letting \( t = n \) in the loop invariant says that after the algorithm terminates, \( \text{sum} \) is the total salary of those people among \( a_1, \ldots, a_n \) who make between \( L \) and \( H \), and \( N \) is the number of such people. The algorithm returns \((\text{sum}/N, N)\), which is the correct average salary and number of people.

(d) Describe the running time of this algorithm in \( \Theta \) notation, assuming that comparisons and arithmetic operations take constant time. Justify your answer.

**Solution:** This is a linear time algorithm, \( \Theta(n) \). The operations inside the for loop take constant time, and the for loop runs \( n \) times. All other work outside the loop is also constant time.
13.

(a) If \( n = 5 \), give an example of two \( n \)-digit numbers that would be a best-case input to the addition algorithm, in the sense that they would cause the fewest single-digit additions possible.

**Solution:** One example is 11111 plus 22222.

(b) In the best case, how many single-digit additions does this algorithm make when adding two \( n \)-digit numbers?

**Solution:** In each column, there is at least one addition, and the best case is when there is only this one addition (no carrying). Therefore the total number of single-digit additions could be as low as the number of columns, or \( n \).

(c) In the best case, when adding two \( n \)-digit numbers, describe the number of single-digit additions in \( \Theta \) notation.

**Solution:** \( n \) is \( \Theta(n) \), so the best case is linear in the number of digits.

(d) If \( n = 5 \), give an example of two \( n \)-digit numbers that would be a worst-case input to the addition algorithm, in the sense that they would cause the most single-digit additions possible.

**Solution:** One example is 99999 plus 88888.

(e) In the worst case, how many single-digit additions does this algorithm make when adding two \( n \)-digit numbers?

**Solution:** In the rightmost column, there are two numbers to add, so that is one single-digit addition. In all of the other \( n - 1 \) columns, there could be 3 numbers to add (from carrying), so that is two single-digit additions each. Therefore the total number of single-digit additions could be as high as \( 1 + 2(n - 1) = 2n - 1 \).

(f) In the worst case, when adding two \( n \)-digit numbers, describe the number of single-digit additions in \( \Theta \) notation.

**Solution:** \( 2n - 1 \) is \( \Theta(n) \), so the worst case is linear in the number of digits.
14.

(a) **Solution:** True. As $2n^2 + 3n \in \Theta(n^2)$, it must be true that $2n^2 + 3n \in O(n^2)$.

(b) **Solution:** True. When $n \to \infty$, the limit of $\frac{n \log n}{n^2}$ is 0.

(c) **Solution:** False. $24n^4 + 120n^2 \in \Theta(n^4)$.

(d) **Solution:** False. $17n^3 + 18n^2 + 5 \in \Theta(n^3)$. $n^4$ is not a lower bound.

(e) **Solution:** False. $17n^3 + 18n^2 + 5 \in \Theta(n^3)$. $n^3$ is a tight lower bound.

(f) **Solution:** True. $17n^3 + 18n^2 + 5 \in \Theta(n^3)$. $n^2$ is a lower bound that is not tight.

(g) **Solution:** True. As $n \to \infty$, the limit of $\frac{\sqrt{n^3}}{n^2}$ is 0, which says $\sqrt{n^3} \in O(n^2)$.

(h) **Solution:** True. As $n \to \infty$, $\frac{\log(n)}{\log(\log(n))} \to \infty$. Also, it’s also possible to show $\log n < n$ for $n \geq 1$.

(i) **Solution:** True. $\log((n!)^2) = 2 \log(n!) \in \Theta(n \log n)$, so $n^2$ is a good upper bound.

(j) **Solution:** False. $1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \in \Theta(n^3)$.

(k) **Solution:** True. $\log(n^2) = 2 \log(n) \in \Theta(\log(n))$ and $\log(10^{10} \cdot n^{10}) = 10^{10} \cdot 10 \log(n)$, therefore $n^2$ is the most significant and it should be $\Theta(n^2)$.

(l) **Solution:** True. $\sqrt{2^n} = (\sqrt{2})^n \approx 1.414^n$.

(m) **Solution:** False. $n^3$ is a tight upper bound.

(n) **Solution:** True.