Question 1 (1 point): In the class we talked about different possible ways of speeding up insertion sort. One proposal was to use binary search.

* Write a pseudo-code for binary search which takes as input a sorted array A of size n and a query element q. Your procedure should return the position of q in A, if q occurs in A. Otherwise it should return -1.

Answer:
There are two different implementations:

Recursive Solution
1. `Binary_Search(A[1..n],q,low,high):
   2. if(high < low):
      3. return -1
   4. mid = (high + low)/2
   5. if(A[mid] == q):
      6. return q
   7. elif(A[mid] > q):
      8. return Binary_Search(A[1..n],q,low,mid-1)
   9. else:
      10. return Binary_Search(A[1..n],q,mid+1,high)

Iterative Solution
1. `Binary_Search(A[1..n],q):
   2. low = 1
   3. high = n
   4. while (low<high):
      5. mid = (high + low)/2
      6. if(A[mid] == q):
      7. return q
      8. elif(A[mid] > q):
         9. high=mid-1
      10. else:
         11. low=mid+1
      12. return -1
• Given a sorted array of size n, and an arbitrary query q, what is the maximum number of comparisons that your binary search procedure performs? Prove your result formally.

Answer: There is a maximum number of recursive steps we can call. This is due to search space being halved at each iteration. Let’s assume \( n = 2^k \) for some \( k \in \mathbb{N} \). At each iteration of Binary_Search, our function either:
- returns a value (line 6), or
- recursively calls Binary_Search with half of the search space (line 8 and 10)
Since we are looking for maximum number of comparisons (ie. worst case) we will assume we keep halving the array.

After first recursive call (or while iteration), our space will be of size \( \frac{n}{2} = 2^{k-1} \).
Likewise, after \( m^{th} \) step, our space will be of size \( \frac{n}{2^m} = 2^{k-m} \).
The worst case will be where our search space is completely exhausted, that happens when we cannot further divide our array into two. This is when we are left with an array of size 1. Assuming this happens at the \( m^{th} \) step, we need to solve for \( 2^{k-m} = 1 \). This equation is satisfied at \( m = k \). Since we had assumed \( n = 2^k \), we know that \( k = \log_2 n \), hence the maximum number of iterations will be \( m = \log_2 n \).
Question 2 (2 point): Your friend claims to have invented a new sorting algorithm. The basic idea is as follows: Find the smallest number in A[1...n] and exchange it with A[1]. Then find the smallest number in A[2...n] and exchange it with A[2] and so on. She needs your help to formalize this idea, and to prove its correctness.

• Write a pseudo-code for your friend’s algorithm.

   Answer:
   1. Sort(A):
   2.     for i = 0 to length(A)-1:
   3.         smallest_index = i
   4.         for j = i+1 to length(A)-1:
   5.             if A[j] < A[smallest_index]:
   6.                 smallest_index = j
   7.             temp = A[i]

• State the loop invariant that the algorithm maintains.

   Answer:
   At the start of each iteration of the for loop of lines 1-9, the subarray A[0..i-1] consists of the i-1 smallest elements within the original array A, but in sorted order.

• Show using an induction argument that the algorithm indeed maintains the loop invariant.

   Answer:
   Initialization:
   i = 0 which implies that A[0..i-1] = A[-1], which is an empty array, thus it is sorted.
   Maintenance:
   The nested for loop ensures that the smallest element is found, within the remaining array A[i+1...len(A)], by comparing all elements to the current smallest element, starting at A[i], then updating that element if a smaller one is found.
   if A[j] < A[smallest_index], then smallest index is assigned j
   The smallest element is swapped with A[i] (within lines 7-9) and since A[0..i-1] was already sorted and contained elements smaller than any element within A[i..len(A)], A[0..i] is now sorted.
   Termination:
   i = n+1 which implies that A[0..n] = A is sorted.
• What is the time complexity of the algorithm in $\Theta$ notation?

**Answer:**

$\Theta(n^2)$

Ignoring lines that take constant time (lines 3-9), the *for* loop of lines 1-9 will always have to run for $n$ number of times, while the nested *for* loop will also always need to run for $n$ number of time maximum (as the first iteration in the main loop will cause the nested loop to run for $n$ time). Thus regardless of the input, there will be $n^2$ number of iterations through the for loop.

\[
f(n) = n^2
\]

\[
g(n) = n^2
\]

\[
0 \leq C_1 \cdot n^2 \leq n^2 \leq C_2 \cdot n^2
\]

\[
C_1 = \frac{1}{2}
\]

\[
C_2 = 2
\]

\[
n_0 \geq 1
\]

\[
f(n) = \Theta(n^2)
\]
**Question 3 (2 points):** This problem will help you get some practice with induction proofs. In the class, we used induction to prove the loop invariant of insertion sort. Here, we want to use induction to prove the following statement: The number of subsets of \{1, 2, ..., n\} having an odd number of elements is $2^n$.

**Answer:**

Let's look at the base case where $n = 1$. $S = \{1\}$ and $P(S) = \{\emptyset, \{1\}\}$.

Where $S$ is our set, and $P(S)$ is its power set (set with all subsets).

Number of elements (cardinality) in $P(S)$ is $|P(S)| = 2^n$ given $|S| = n$.

Number of odd-numbered subsets: $Odd = 2^{n-1} = 2^{1-1} = 2^0 = 1$ holds for base.

Also realize number of even-numbered subsets $Even = All - Odd = 2^n - 2^{n-1} = 2^{n-1} = 1$ also holds.

Now, let’s look at the inductive step. Assume this is true for a set with $n - 1$ elements. Let’s try to arrive at the conclusion it also holds for set with $n$ many elements. If $S_{n-1}$ (subscript to show number of elements) has $2^{n-1}$ many subsets, we are assuming it has $2^{n-2}$ many odd-numbered (and even-numbered) subsets.

We realize every subset of $S_{n-1}$ is also a subset of $S_n$. This is because the only difference between two sets is the element $n$. In other words, we did not remove any elements from $S_{n-1}$, hence any subset should stay as a subset in this transition. We also realize that $S_n$ has new subsets (that $S_{n-1}$ did not have). And these are simply all subsets of $S_{n-1}$ with an additional $n$.

(An example to make it easier to see: Assume n=3. $P(S_2) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$. And, $P(S_3) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\} \cup \{\{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

We kept 4 subsets that existed in previous step. Then we also added 3 to all of them.)

For subsets that already existed in $S_{n-1}$: $Odd_{S_{n-1}} = 2^{n-2}$ and $Even_{S_{n-1}} = 2^{n-2}$.

For subsets that are newly created in $S_n$: $NewOdd_{S_n} = Even_{S_{n-1}}$ and $NewEven_{S_n} = Odd_{S_{n-1}}$. This holds because we are simply “adding one element to each subset”. Hence, odd subsets become even and vica versa.

And, $Odd_{S_n} = Odd_{S_{n-1}} + NewOdd_{S_n} = 2^{n-2} + 2^{n-2} = 2^{n-1}$. 

5 of 6
**Question 4 (1 points)** Let \( f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_k n^k \) be a degree-\( k \) polynomial where every \( a_i > 0 \). Show that \( f(n) \in \Theta(n^k) \).

**Answer:**

\[
f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_k n^k
\]

\[
g(n) = n^k
\]

\[
f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_k n^k = \sum_{i=0}^{k} a_i n^i
\]

\[
0 \leq C_1 \cdot n^k \leq \sum_{i=0}^{k} a_i n^i \leq C_2 \cdot n^k
\]

\[
0 \leq C_1 \leq \sum_{i=0}^{k} \frac{a_i}{n^k - i} \leq C_2
\]

\[
\lim_{n \to \infty} \sum_{i=0}^{k} \frac{a_i}{n^{k-i}} = a_k
\]

\[
0 \leq C_1 \leq a_k \leq C_2
\]

\[
C_1 = a_k - 1
\]

\[
C_2 = a_k + 1
\]

\[
n_0 \geq 2
\]