Brian Bauman
CSC 421 Assignment 1

1.

```
n = size of BOLTS (or NUTS)
i = 0
j = 0

while i < n and j < n do
    if BOLTS[i] == NUTS[j] then
        return true
    else if BOLTS[i] < NUTS[j] then
        i++
    else
        j++
end while
return false
```

This algorithm relies on the arrays being presorted. If a given nut is smaller than the first bolt, move to the next nut and recompare. Continue doing this until it matches (and exit) or it is too big (in which case you choose the next bolt). In any case other than finding a match, either i or j is incremented. Because neither can be incremented further than n, the worst case scenario would be incrementing them both to n. As such, this algorithm makes 2n comparisons in the worst case. Comparisons are the main operation being performed here and can be used to estiate the complexity. It should be noted that while there are two comparison checks listed here within the while loop, the comparison can be performed once and the returned value can be used to determine which value to implement.

2.

a)

```
n = size of A
x = 0
y = n - 1

while x < y do
    sum = A[x] + A[y]
    if sum == t
        return true
    else if sum < t
        x++
    else
        y—
end while
return false
```

This algorithm sums A[x] and A[y], increasing or decreasing an index depending on how the result compares to t. Each iteration of the while loop makes a single comparison (the driving cost here). Because we must stop when x and y cross, this algorithm makes no more than a single pass through the array (coming from both ends at once), meaning that the while loop executes at most n times.

b)

```
n = size of A

for z = 0 to n - 1 do

      x = 0
      y = n - 1
      r = t - A[z]

      while x < y and  do

            if x == z then x++
            if y == z then y—

            sum = A[x] + A[y]
            if sum == r
                  return true
            else if sum < r
                  x++
            else
                  y—
      end while

end for
return false
```

Iterating through A, we can determine the value, r, that is the difference between the current number (A[z]) and our target sum, t. Because we now know that we must find two additional numbers that sum to r, we can reuse the code from 2.(a) to find this answer in O(n) time (see explanation above). Because we are running an algorithm that takes O(n) time for each of the n values in A, it is clear that this algorithm takes O(n^2) time. Note that we avoid allowing either of the other two values, x and y, to be the value at index z.

3.

This is impossible, and Pinnochio's nose will grow. Because the array is not sorted, there is no way to know which value in the array could not be added with a value you are currently inspecting in order to obtain a sum of 1000. For example, for each value in the array, you can determine the value you are looking for (by subtracting the current value from 1000). To find this value in the rest of an unsorted array would require linear time (O(n)) as discussed in class. Because you must answer this question n times (the length of the array), it is clear that this problem is more complex than O(n). Any attempts to presort the array would require O(nlogn) time and would still not qualify.

4.

```
sort(A)

for i = 1 to n - 1 do
     if (A[i] == A[i - 1]
          return true
end for
return false
```

This algorithm has two separate steps. The first sorts the points in the array in order of ascending x-coordinate, breaking ties in order of ascending y-coordinate. As given at the top of this assignment, sorts can be done in O(nlgn) time. Next, traverse the array once through, comparing each element to its predecessor. This results in n - 1 compares being performed. The algorithm then can be said to take O(nlgn + n) time, which is asymptotically equivalent to O(nlgn) time, as nlgn is the leading term.

5.

```
A = set of points
n = size of A
for i = 0 to n do

     sort(A) based on polar angle w.r.t. A[i]

     for j = 1 to n - 1 do
          if j != i and polarAngle(A[j]) == polarAngle(A[j - 1])
               return true
     end for
end for
return false
```

For a set of n points, iterate through each point in the array. For each point, treat it as the origin and find the polar angle of all other points with respect to this point. Sort the array in order of ascending polar angle. As given at the top of this assignment, this sort can be done in O(nlgn) time. As in the last question, look for two values in this array that have the same polar angle w.r.t. the point you are treating as the origin (exluding this point itself, of course). This pass through the array performs a number of polar angle calculations and comparisons on the order of n, the length of the array. As in the last question, this inner processing takes O(nlgn) time as the sorting — O(nlgn) — is the dominating term. Because you must perform this process by treating each of the n points in the array, the final running time is O(n * nlgn), or O(n^2*lgn).