Problem 1
[10 pts] Given a collection of n nuts and a collection of n bolts, arranged in an increasing order of size, give an $O(n)$ time algorithm to check if there is a nut and a bolt that have the same size. The sizes of the nuts and bolts are stored in the sorted arrays *NUTS[1..n]* and *BOLTS[1..n]*, respectively. Your algorithm can stop as soon as it finds a single match (i.e, you do not need to report all matches).

Solution: Since we have two sorted array, we want to keep track both of them with two iterators, i for *NUTS[1..n]* and j for *BOLTS[1..n]*.

```
i, j ← 1;
while i < n and j < n do
    if NUTS[i] = BOLTS[j] then
        return true; /* Once we find two identical number in both array we stop looping
         */
    else if NUTS[i] > BOLTS[j] then
        j ← j + 1; /* We want smaller number array increase their index and compare
         again with greater one                                                     */
    else
        i ← i + 1;
    end
end
```

We go through index in both array only once, so the worse case is $O(2n) \approx O(n)$.
Noticed that if $NUTS[1..n] \cap BOLTS[1..n] = \emptyset$, then their range is not overlap and there is no reason to keep searching, thus we can add initial statement.

```
i, j ← 1;
while i <n and j <n do
    if NUTS[n] < BOLTS[1] or NUTS[1] > BOLTS[n] then
        return;
    end
    if NUTS[i] == BOLTS[j] then
        return true;
    else if NUTS[i] > BOLTS[j] then
        j ← j + 1;
    else
        i ← i + 1;
    end
end
```

Problem 2
[15 pts] Let $A[1..n]$ be an array of distinct positive integers, and let t be a positive integer.

1. **[5 pts]** Assuming that A is sorted, show that in $O(n)$ time it can be decided if A contains two distinct elements x and y such that $x + y = t$.

2. **[10 pts]** Use part (a) to show that the following problem, referred to as the 3-Sum problem, can be solved in $O(n^2)$ time:
(3-SUM) Given an array $A[1..n]$ of distinct positive integers that is not (necessarily) sorted, and a positive integer t, determine whether or not there are three distinct elements x, y, z in A such that $x + y + z = t$.

Solution:

1. One solution is to set two x and y be the minimum (A[1]) and maximum (A[n]) of array $A[1..n]$, and then assume that the value of t is sum of x and y, if the sum is greater than t we decrease the index of the y, and similarly if the sum is less than t we increase the index of the x.

```
i ← 1  j ← n;
while i < j do
    if A[i] + A[j] = t then
        return true;
    else if A[i] + A[j] > t then
        j ← j − 1;
    else
        i ← i + 1;
    end
end
```

The worst case is value of i reach j which takes $n - 1$ times, thus time complexity is $O(n)$.

2. By adding extra variable z, meaning we want to go through part (a) process for all elements in $A[1..n]$ such that $x + y + z = t$ and $x \neq y \neq z$.

```
for k ← 1 to n do
    i ← 1  j ← n;
    while i < j and i ≠ k and j ≠ k do
        if A[i] + A[j] + A[k] = t then
            return true;
        else if A[i] + A[j] + A[k]> t then
            j ← j − 1;
        else
            i ← i + 1;
        end
    end
end
```

By adding a for loop for part (a), our time complexity is $O(n^2)$ in this case.

Problem 3
[10 pts] Let $A[1..n]$ be an array of positive integers (A is not sorted). Pinocchio claims that there exists an $O(n)$-time algorithm that decides if there are two integers in A whose sum is 1000. Is Pinocchio right, or will his nose grow? If you say Pinocchio is right, explain how it can be done in $O(n)$ time; otherwise, argue why it is impossible.

Solution: Yes, by using hash table to search specific value, it only takes $O(n)$ time complexity. Assume $A[1..n]$ store in hash table, then we could implement a two sum algorithm as following.

```
for i ← 1 to n do
    x ← 1000 − A[i];
    if x exist in A[1..i] then
        return true;
    end
end
```

As mentioned before, we can use hash table search method (A.contains(x) in some languages) to find whether x exist in array, thus it takes $O(n)$ time complexity.

Problem 4
[10 pts] Let $A[1..n]$ be an array of points in the plane, where A[i] contains the coordinates (x_i, y_i) of a point p_i, for $i = 1, ..., n$. Give an $O(nlgn)$ time algorithm that determines whether any two points in A are identical (that is, have the same x and y coordinates).

Solution: Assume by using heapsort (time complexity of $O(nlogn)$) to sort $A[1..2]$ as $A_{sorted}[1..n]$. Then we compare two adjacent points to find if there are two identical coordinates.

$i \leftarrow 1$;
while $i < n$ - *1* **do**
  if $A_{sorted}[i].x = A_{sorted}[i+1].x$ *and* $A_{sorted}[i].y = A_{sorted}[i+1].y$ **then**
    return true;
  else
    $i \leftarrow i + 1$;
  end
end

The total time complexity of cost is $O(nlogn) + O(n) \approx O(nlogn)$.

Problem 5
[15 pts] Show how to determine in $O(n^2lgn)$ time whether any three points in a set of n points are collinear.

Solution: Assume the initial value in $A[1..n]$ and we first want to compute slope of all possibilities.

```
A_slope; /* Initialize an empty array A_slope for storing slopes */
i ← 1;
while i < n do
    j ← i + 1;
    while j < n do
        slope ← (A[i].y − A[j].y)/(A[i].x − A[j].x);
        A_slope.add(slope);
        j ← j + 1;
    end
    i ← i + 1;
end
```

Next step by using nested loop for checking two identical slope in A_{slope}, iff this happens, three points of set are collinear. Totally the time complexity is $O(n^2) + O(n^2) \approx O(n^2) < O(n^2lgn)$.