Q1)

In the question, it is stated that as a hint we can use binary-search due to its time complexity logn and we know that in order to use binary search , we must have a sorted list to achieve this goal.

In order to sort our list S , it is preferred to use Merge-Sort due to its time complexity big-theta nlogn. Merge-sort wont effect our goal due to its complexity nlogn.

After sorting the list , we fill search if there are two element whose sum equals to X. Procedure is

```
SUM-SEARCH( S , X )
        Merge-Sort(S, 0 , S.Length -1 )
        For k = 0 upto S.length-1
                Position = Binary-Search( S , X – S[k] )
                If position ≠ NULL and position ≠index
                        Return true
        Return false
```

As a example , lets have set S = { 2 , 3 , 1 , 4 } . We will first apply merge-sort and it will become S = { 1 , 2 , 3 , 4 } . If we determine X as 5 , in first iteration it will check for binary-search(S , 5 – S[0]) which is binary-search(S , 4) and it will return 3. In if statement , program sees that position is a number and it is not equal to current index and as a result it returns true which means there are two elements whose sum is equal to X.

Q2)

Our base case is when n=2 and in that case T(n)= nlgn becomes T(2)=2lg2 = 2 which is true.

Our inductive step is assuming that there is an integer which is greater than 1 and n = 2^k and in that case $T\left(2^k\right) = 2^k lg2^k$ and we must prove that $T\left(2^{k+1}\right) = 2^{k+1}lg2^{k+1}$ holds.

$$T\left(2^{k+1}\right) = 2T\left(2^{k+1}/2\right) + 2^{k+1}$$

$$T\left(2^{k+1}\right) = 2T\left(2^k\right) + 2^k * 2$$

$$T\left(2^{k+1}\right) = 2 * 2^k * lg2^k + 2^k * 2$$

$$T\left(2^{k+1}\right) = 2 * 2^k(lg2^k + 1)$$

$$T\left(2^{k+1}\right) = 2 * 2^k(lg2^{k+1})$$

$$T\left(2^{k+1}\right) = 2^{k+1}lg2^{k+1}$$

As we can see , we already proven the base and inductive case , we arrive that T(n) = nlgn holds for the numbers which are the powers of 2.

Q3)

A)

As we can see from the pseudecode , there is only one for loop which goes from n to zero and in that case there are n-iteration so we can say that the time complexity is $\theta(n)$.

B)

To do Naive polynomial evaluation, we must keep the coefficients in an array to use while iterating between polynomial fragmentations. We will find by one by x, x^2, x^3 to x^n and we will multiply them with their corresponding coefficient.

So the pseudocode is:

```
Naïve-Polynomial(CoeffArr , x )
 y = 0
for k =1 to CoeffArr.Length
        tempX = 1
        for h =1 to k-1
                tempX = texmpX * x
        y = y + tempX*CoeffArr[k]
```

If we look at the pseudocode that we can see there is nested for loop and as we know nested for loop means $\theta(n^2)$ time complexity.

The runtime we get from naïve polynomial evaluation is worse than horner's approach because horner approach was using linear runtime which is $\theta(n)$.

C)

Initialization:

At the initialization step , we must use the main sum formula in order to calculate the result of starting. So:

y = $\sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$

y $= \sum_{k=0}^{n-(n+1)} a_{k+n+1} x^k$

y $= \sum_{k=0}^{-1} a_{k+n+1} x^0$

y = 0

we found y as 0 so we proved the initialization step.

Maintenance:

If we use the loop invariant that comes from Line 2-3 and if we manipulate the sum formula we arrive :

Y` = a_i + x * y

Y` = a_i + x * $\sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$

Y` = $\sum_{k=-1}^{n-(i+1)} a_{k+i+1} x^{k+1}$

Y` = $\sum_{k`=0}^{n-(i+1)} a_{k`} i^{k`}$

Y` = $\sum_{k`=0}^{n-(i`+1)} a_{k`+i`+1} x^{k`}$

As it stated here , we can summarize that the loop invariant holds here.

Termination:

The last step is the when loop ends and in that case , i should be equal to -1 and if we put this number to sum formula , we arrive that:

Y = $\sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$
Y = $\sum_{k=0}^{n-(-1+1)} a_{k+-1+1} x^k$
Y = $\sum_{k=0}^{n} a_k x^k$

As we can see above, we arrive the sum equation which was asked in the question and when program terminates , the given sum equation holds.

D)

Horner's rule properly evaluates the polynomial as intended when it finishes. This indicates that the algorithm works correctly.