Problem 1.

In Insertion sort algorithm we assume that the first element is already sorted and move from left to right checking if the right element is smaller than the left element going from element 2 to element n. If its not true we will have to make an operation that will insert the smaller element to the left and move all the subsequent elements to the right. However in a best case scenario we would have a list (or array) that is already sorted. So all we have to do is would be to compare each element once with its left element and will have to go thru the list once therefore the Run time will be only dependent on the number of items in the list linearly.

Problem 2.

If f(n) is Ω(g(n)) it implies that for any n >no and constants C1 and C2 0 <=C1(g(n)) <= f(n) <= C2(g(n))

Now if g(n) is Ω(h(n)) it implies for any n >no and constants D1 and D2 0 <=D1(h(n)) <= g(n) <= D2(h(n))

Since f(n) is bound by g(n) i.e f(n) lies in a range set by C1g(n) and C2g(n) and g(n) is bound by h(n) i.e all possible values of g(n) will lie between D1h(n) and D2h(n) we can say f(n) is bound by h(n)

Hence f(n) = Ω(h(n))

Problem 3.

True

Problem 4.

False

Problem 5.

For insertion sort at the max we need to make n(n-1)/2 comparisons starting comparing second element to the first and compare all the elements until N Summation of J varying from 2 to n. and for exchange also we will have at the max exchanges of n(n-1)/2.

Problem 6.

A binary tree is a representation of list such that it has equal number of left and right child i.e it has 2^D elements in other words the height of the tree is D = lg(n), a binary tree where the nodes are less than lg(n) is called near binary tree.

Problem 7.

Heap sort gets it done in near log n as merge sort algorithm but it does not need as much memory and since it does not need to store the small small arrays to divide the problem heap sort does the sorting in place and hence requires as little memory as used by Insertion sort.

Problem 8.

Height of n element heap is lg(n) +1 (from leaves to the root), if root is not counted then lg(n)

Problem 9.

Its neither a min or a max heap

Problem 10.

```
Var string = givenstring
Var NewString
Length = length of string
Index = 1
Def  reverseString(string, NewString, index){
If index > Length     //comment – we need to run this only till we complete the whole lenght
        Break
Else
        NewString[index] = string[Length+1-index] // comment store string in reverse order
        Index = index +1
        reverseString(string,NewString,index)
}
```

Problem 11.

3 inversions

1. 2 with 5
2. 2 with 3
3. 4 with 5

Problem 12.

N(n-1)/2

Problem 13.

N(n-1)/2

Problem 14.

Assumption there is only one maximum element

Element – name of Array

Length = length of array

Index = Round(length/2)

```
Def CheckMax(index) {
If  Element[index-1] < Element [index] and Element[index] > Element[index+1]
        Return Element[index]
Else if Element[index]< Element[index+1]
        Index = index + Round(length-index)/2
Else
        Index = index - Round(length-index)
CheckMax(index)
}
```