Problem 1

$T(n) = T(n/2) + n; \quad T(n) = 1$

To use the master formula,

$a = 1$

$b = 2$

$k = 1$

Hence, $a = 1 < b^k = 2^1$. Therefore, $T(n) = \Theta(n^k)$

$Tn = \Theta(n^1) = \Theta(n)$

Problem 2

~~Algorithm isPrime~~(n):

Input: An integer number

Output: Checks if the number is prime and return 1 if so, 0 if not

```
if n<2 || n%2=0 then
    return 0
else if n<4 then
    return 1
i<-3
while i*i < n do
    if n%i = 0
        return 0
    i = i+2
return 1
```

Therefore total running time in terms of value is $O(n^{1/2})$ ∴ in terms of size

$T(b) = O(f(2^b))$ when $f(n) = O(n^{1/2})$

$T(b) = O((2^b)^{1/2}) = O(2^{b/2})$

4.
A. To prove the correctness of the algorithm,

i. Does it have a base case and do the non-base calls lead to the base case?
The algorithm has a base case which will handle lists of size 0 or 1. When such a list is encountered, it will return the input. The non-base calls also lead to base cases because every time we are dividing the list into two smaller parts which will lead to a list of size 1 or 0.

ii. Is the base case output correct?
We expect the base case to return the given list itself which the algorithm does. Hence, the base case returns the expected and correct output.

iii. Given the algorithm works for all lists of size less than n, let's prove if it does hold true for size of n. After a successful partition into L1 (left) and L2 (right), we know that recSort works (returns valid result for length < n). Hence, recSort on both L1 and L2 return valid and sorted results. The merge algorithm will merge L1 and L2 in the correct order. Hence, L will have an ordered list. Therefore, the algorithm works for length of n.

Since all the above criteria are satisfied, the algorithm is correct.

B.

Algorithm recSort(S)	Operations
Input: list L with n elements	
Output: list L sorted	
if L.size() > 1 then	2
(L 1 , L 2) ← partition(L, n/2)	O(n) + 1
recSort(L1)	T(n/2)
recSort(L2)	T(n/2)
L ← merge(L1 , L2)	O(n) + 1
return L	1

The total running time is given by T(n) = 2T(n/2) + O(n) + c where c is constant for n > 1 and 3 otherwise. Since O(n) dominates c, we can omit c.

T(n) = 2T(n/2) + O(n) for n > 1
T(n) = 3 otherwise (n <= 1)

To use the master formula, a = 2, b = 2, k = 1. Since $a = b^k$, T(n) = O(nlogn).

C. The algorithm is implemented using Java. Tests have been done to compare the running time with LibrarySort which is slower than RecSort. RecSort is more than 40% faster than LibrarySort on repeated tests using the tool provided.