Question 1

(a) Algorithm Description
Key idea: Two Sorted lists can be merged in linear time
We need to merge k sorted lists, hence we will take a List of k sorted lists as input and then perform divide and conquer on this new List.

Divide Step: Divide List[L1,L2...Lk] into two sublists List[L1...k/2] and List[Lk/2+1...Lk]

Conquer Step: Merge List[L1...k/2] recursively
Merge List[Lk/2+1...Lk] recursively

Combine Step: Merge the two resultant lists in linear time

(b) Consider the following algorithm.

Algorithm 1: Merge k sorted lists using divide & Conquer.

Input: A List[L1,L2...Lk] containing k sorted lists with a total of n elements combined
Output: Final Merged Sorted List with n elements

```
/* Here's my algorithm.  */
```

```


Merge_Sorted_Lists(List[L1,L2...Lk], i, j){
    if i == j
    return List[i]

    mid = [(i + j)/2]

    List1 <- Merge_Sorted_Lists(List[L1,L2...Lk], i, mid)
    List2 <- Merge_Sorted_Lists(List[L1,L2...Lk], mid+1, j)

    rerturn Merge(List1, List1.size, List2, List2.size)

}

Merge(L1, m, L2, n) {
    create List L3 with size m+n
    Copy L1 to L3 (first m elemnents of L3)
    end1 <- m
    end2 <- n
    endresult <- m+n

    while(end1 >= 1 and end2 >=1 ) do
        if(L2[end2] > L3[end1]) then
                L3[endresult] = L2[end2]
                endresult--
                end2--
        else
                L3[endresult] = L3[end1];
                endresult--
                end1--
        end
    end

    while(end2>=1)do
        L3[endresult] = L2[end2];
        endresult--
        end2--
    end

    return L3

}

```

(c) Explain the algorithm (Correctness).
The basic logic as mentioned is that we recursively divide the lists in two halves, until we reach the base condition (1 element). The base condition states that if one element then return the list as it is. When we go bottom up, we keep on merging sorted lists (combine step) until our first call where we merge the two sorted half lists into our final merge List of n elements.
Example: L1 [3, 12, 19, 25, 36], L2[34, 89], L3[17, 26, 87], L4[28], L5[2, 10, 21, 29, 55, 59, 61]
We keep on dividing till we reach the base condition
Initially list is broken down into: [L1 L2 L3] and [L4 L5]
[L1 L2 L3] breaks down into [L1 L2] and [L3]
[L4 L5] breaks down into [L4] and [L5] (base condition reached)
[L1 L2] breaks down into [L1] and [L2] (base condition reached)

Now we keep on merging by applying our merge sorted lists logic until we reach the root. L1 and L2 merge to give the result [3, 12, 19, 25, 34, 36, 89] Similarly, we keep on merging our divided steps until we reach the root and get our final Sorted List
L : 2,3,10,12,17,19,21,25,26,28,29,34,36,55,59,61,87,89.

(d) Running time analysis.
Recurrence: $T(n) = 2*T(k/2) + n$
Assume the tree recursion method, We keep on dividing until at the leaves we reach at k nodes (k lists) and we process n at each level. Since height of tree is Log(k), we have n logk times. Final solution will be $k + n(\log(k))$.

$T(n) = O(n \log(k))$

Question 2

(a) Inversions in array 4,2,9,1,7
{4,2}
{4,1}
{2,1}
{9,1}
{9,7}

(b) What array with elements from the set {1, 2, . . . , n} has the most inversions? How many inversions does it have?
Soln: An array in descending order made from the above elements would have the highest inversions. A[n, n-1,...., 2, 1]
Here the 1st element n, would have n-1 inversions
2nd element n-1 would have n-2 inversions
.
.
.
second to last element 2 would have 1 inversion
last element 1 would have 0 inversions.
Total inversions =

$$\sum_{n=1}^{n-1} n$$

Since sum of n numbers = n * (n + 1)/ 2
sum of n - 1 numbers = (n-1) * (n + 1 -1) / 2
Total inversions = n * (n-1) / 2

(c) Algorithm Description
Here we need to find the count of inversions in an Array. We can simply use the merge sort algo. Basically, for each array element after we merge, count all elements more than it to its left and add the count to the output.
Since the subarray is already sorted, it is obvious that elements to the left of an element will also form an inversion, hence we can directly add to the count.
Divide Step: Divide A[1,2...n] into two sublists A[1...n/2] and A[n/2+1...n]

Conquer Step: Sort A[1...n/2] recursively
Sort A[n/2+1...n] recursively and also calculate inversions for subarrays

Combine Step: Merge the two resultant lists in linear time and add the inversions for the subtree
Consider the following algorithm.

Algorithm 2: Count the number of inversions in an Array A[1...n]

Input: An array A[1...n] with n distinct numbers
Output: Count of the number of inversion pairs. If $i < j$ and $A[i] > A[j]$, then the pair (A[i], A[j]) is called an inversion of A.

```
/* Here's my algorithm. */
```

```
Get_Inversion_Count(A[1,2...n], i, j){
    if i == j
    return 0

    mid <- [(i + j)/2]
    count <- 0

    count += Get_Inversion_Count(A[1,2...n], i, mid)
    count += Get_Inversion_Count(a[1,2...n], mid+1, j)

    count += Count_Inversions_Merge(A[1,2,...n], i, mid, j)

    return count

}

Count_Inversions_Merge(A[1,2,...n], i, mid, j){
    Create an array B[1...j-i+1]
    a = i, b = mid+1, c=1
    count = 0
    while a<= mid and b<=j do
        if A[a] <= A[b] then
            B[c] = A[a]
            a++, c++
        else
            B[c] = A[b]
            b++, c++
            count += mid - i + 1
        end
    end

    if a <= mid
        copy A[a..mid] to B
    if b <= j
        copy A[b..j] to B

    Copy B back to A[i...j]
    return count
}
```

Explanation with Example(Correctness):
Consider the List $\{4, 2, 9, 1, 7\}$
Like a typical divide and conquer algo, we first divide the elements recursively until we reach a base case (here, n = 1). So the array above first get divided into $\{4, 2, 9\}$ and $\{1, 7\}$.
We keep on dividing, until we reach single elements. We then do a merge sort while building up.
Consider 4 and 2, we sort it as 2 and 4, count all values to left of 4, here count = 1. For 1 and 7 , count = 0.
Now merging {2,4} and {9}, Here since 9 is bigger than both 2 and 4, no inversion count increments.
Now merging {2,4,9} and {1,7}, 1 moves at front and all the elements 2 4 and 9 automatically gets counted as the inversions, count = 3.
Similarly, one more inversion gets counted when 7 is moved ahead of 9.
We get a sorted list along with total inversions = 5

Running time analysis. The time complexity is the same as that of merge sort(done in class).
Recurrence relation = T (n) = 2 T (n/2) + n
On solving this by any of the methods. We get our time analysis of O (n log(n))

Question 3

(a) T(n) = 2T(n/2) + n^3
Soln: (Masters theorm)
Acc. to masters theorm:
Considering 3 constants $a >= 1$, $b > 1$, $n' >= 1$
and
T(n) = 1 $\quad n <= n0$
T(n) = a*T(n/b) + f(n) $\quad n > n'$

We compare f(n) vs $n^{log_b a}$
If the comparison satisfies one of the three cases, we get our time complexity.
Here, a = 2, b = 2, f(n) = n^3
n^3 vs $n^{log_2 2}$
n^3 vs $n^{1+number}$

As per Case 3
if 1) f(n) = $\Omega(n^{log_b a+\epsilon})$ for some constant $\epsilon > 0$
2) if there exists a $C < 1$, $n0 \geq 1$, such that
a.f(n/b) $\leq C.f(n)$ for all $n \geq n0$
then T(n) = θ f(n)
In our case,
$n^3 \geq n^{log_2 2+\epsilon}$
$= n^3 \geq n^{1+\epsilon}$ where $\epsilon = 2$ which is > 0 .. condition 1 is satisfied
and
$2 * (n/2)^3 \leq C.n^3$
$= n^3/4 \leq C.n^3$
For C = 1/4 , even the 2nd condition is satisfied
Hence, acc to the theorm,
T(n) = θ f(n)
T(n) = $\theta(n^3)$

(b) T(n) = 4T(n/2) + $n\sqrt{n}$
Soln: (Masters theorm)
Acc. to masters theorm:
Considering 3 constants $a >= 1$, $b > 1$, $n' >= 1$
and
T(n) = 1 $\quad n <= n0$
T(n) = a*T(n/b) + f(n) $\quad n > n'$

We compare f(n) vs $n^{log_b a}$
If the comparison satisfies one of the three cases, we get our time complexity.
Here, a = 4, b = 2, f(n) = $n\sqrt{n}$
$n\sqrt{n}$ vs $n^{log_2 4}$
$n\sqrt{n}$ vs $n^{2-number}$

As per Case 1
if 1) f(n) = O($n^{log_b a-\epsilon}$) for some constant $\epsilon > 0$
then T(n) = θ $n^{log_b a}$
In our case,
$n\sqrt{n} \geq n^{log_2 4-\epsilon}$
$= n\sqrt{n} \geq n^{2-\epsilon}$ where $\epsilon = 0.5$ which is > 0 .. condition is satisfied

Hence, acc to the theorm,
T(n) = θ $n^{log_b a}$
T(n) = θ $n^{log_2 4}$
T(n) = $\theta(n^2)$

(c) T(n) = 2T(n/2) + $n * logn$
Solving by expanding recurrences,
Now T(n/2) = 2T(n/4) + $(n/2) * log(n/2)$
Hence T(n) = 2(2T(n/4) + $(n/2) * log(n/2)$) + $n * logn$
T(n) = 4T(n/4) + $n * (log(n) - log(2))$ + $n * logn$
T(n) = 4T(n/4) + $n * log(n) - n$ + $n * logn$
T(n) = 4T(n/4) + $2n * log(n) - n$
Similarly,
T(n/4) = 2T(n/8) + $(n/4) * log(n/4)$
Solving similarly as above,
T(n) = 8T(n/8) + $3n * log(n) - 3n$
kth step:
T(n) = $2^k * T(n/2^k) + k * n * log(n) - n * k * (k-1)/2$
In base case, i.e $T(1), n/2^k = 1$
Therefore $n = 2^k$ and $k = logn$... equation 1
Substitute respective values in General eqn
T(n) = $n * T(1) + log(n) * n * log(n) - n * log(n) * (log(n) - 1)/2$
T(n) = $n * T(1) + log^2(n) * n - (n * log^2(n))/2 - (n * log(n))/2$
T(n) = $n + n * log^2(n) - n * log(n)$ on removing constants
Hence,

T(n) = O $(n * log^2(n))$

(d) T(n) = T(3*n/4) + n
Soln: (Masters theorm)
Acc. to masters theorm:
Considering 3 constants $a >= 1$, $b > 1$, $n' >= 1$
and
T(n) = 1 $n <= n0$
T(n) = a*T(n/b) + f(n) $n > n'$

We compare f(n) vs $n^{log_b a}$
If the comparison satisfies one of the three cases, we get our time complexity.

Here, a = 1, b = 4/3, f(n) = n
n vs $n^{log_{4/3}1}$
Since Log 1 = 0
n vs $n^{0+number}$

As per Case 3
if 1) f(n) = $\Omega(n^{log_b a+\epsilon})$ for some constant $\epsilon > 0$
2) if there exists a $C < 1$, $n0 \geq 1$, such that
a.f(n/b) $\leq C.f(n)$ for all $n \geq n0$
then T(n) = θ f(n)
In our case,
$n \geq n^{log_{4/3}1+\epsilon}$
$= n \geq n^{0+\epsilon}$ where $\epsilon = 1$ which is > 0 .. condition 1 is satisfied
and
$1 * (3 * n/4) \leq C.n$
$= (3 * n/4) \leq C.n$
For C = 3/4 , even the 2nd condition is satisfied
Hence, acc to the theorm,
T(n) = θ f(n)
T(n) = $\theta(n)$

Question 4

(a) Algorithm description
We can use the typical divide and conquer methodology here to get the profit in the left and right half recursively. The only problem is The Buy and Sell can be in two different halves, resulting in crossing between our two recursions, which we need to handle in our merge stage. The problem constraints us to solve in O(n) time, hence we need to solve the cross problem in constant time as compared to linear time in our previous approaches.
Divide Step: Divide A[1,2...n] into two sublists A[1...n/2] and A[n/2+1...n]
Conquer Step: Calculate max profit, buy and sell day for A[1...n/2] recursively
Calculate max profit, buy and sell day for A[n/2+1...n] recursively
Combine Step: Calculate Profit for cross-array indices and compare with left and right sub-array. Merge these solutions recursively and get the final profit, buy day and sell day.

(b) Consider the following algorithm.

Algorithm 3: Calculate the buy and sell days to maximize profit

Input: An array A[1..n], each indice indicates a day and the value at that indice is the rate of stock on the day

Output: Day to Buy, day to sell to maximize profit

`/* Here's my algorithm. */`

```

MaxStockProfit(A[1...n], i, j) {
    if i == j
    return (0, A[i], A[j], i, j)

    mid <- [(i + j)/2]

    profit_l, min_l, max_l, buy_l, sell_l <- MaxStockProfit(A[1...n], i, mid)

    profit_r, min_r, max_r, buy_r, sell_r <- MaxStockProfit(A[1...n], mid +1, j)
    new_profit <- max_r - min_l

    buy_day <- buy_l
    sell_day <- sell_l
    max_profit <- profit_l

    if(new_profit > profit_l and new_profit > profit_r) then
        max_profit <- new_profit
            buy_day = buy_l
            sell_day = sell_r
    else
        if (profit_r >= profit_l) then
            max_profit <- profit_r
            buy_day = buy_r
            sell_day = sell_r
        end
    end

    return max_profit, min(min_l, min_r), max(max_l, max_r), buy_day, sell_day
}

callingFunction(A[1...n]){
    profit, min, max, buy_day, sell_day <- MaxStockProfit(A[1...n], 1, n)
    print Buy on $buy_day$
    print Sell on $sell_day$
    print Profit is $profit$
}
```

(c) Correctness:
So we have 3 possible cases when we merge,
1. The buy and sell for max profit happen in the left subarray
2. The buy and sell for max profit happen in the right subarray
3. The buy is in the left subarray and the sell is in the right subarray

In our algo, we solve for all of these 3 cases. We assume max profit occurs in the left subarray, we then calculate the new profit by subtracting max in right sub array with min in left subarray. After comparing new profit, left and right profit, we make a decision for the final profit at merge.
Doing this recursively, gives us the final profit, buy day and sell day.

(d) Running time analysis.
Recurrence relation that we get here is:
T(n) = 2*T(n/2) + O(1)
As we solved this recurrence in class, we know that the final solution for this is: 2n - 1

Hence, T(n) = O(n)