1. (a) *Algorithm Description*

We proceed with a "Divide and Conquer" strategy and divide the problem of merging k sorted lists as follows:

- Merge $\lfloor k/2 \rfloor$ sorted lists $L_1, \dots, L_{\lfloor k/2 \rfloor}$ into one single sorted list A_1.
- Merge $\lceil k/2 \rceil$ sorted lists $L_{\lfloor k/2 \rfloor+1}, \dots, L_k$ into one single sorted list A_2.

Then we conquer the two aforementioned problems with recursion and combine their respective solutions A_1 and A_2 into A_3 using the canonical Merge subroutine discussed in class.

(b) *Algorithm Pseudocode*

Algorithm 1 Merge-Lists($L_1, \dots, L_k$)

```
1: if $k = 1$ then                                          // base case
2:     return $L_1$
3: end if
4: $A_1 \leftarrow$ Merge-Lists($L_1, \dots, L_{\lfloor k/2 \rfloor}$)     // conquer left subproblem with recursion
5: $A_2 \leftarrow$ Merge-Lists($L_{\lfloor k/2 \rfloor+1}, \dots, L_k$)   // conquer right subproblem with recursion
6: $A_3 \leftarrow$ Merge($A_1, A_2$)                        // combine solutions with Merge subroutine
7: return $A_3$
```

(c) *Algorithm Correctness*

To show the correctness of Algorithm 1, we will use (strong) induction on k. Let k be an arbitrary integer greater or equal to 1. Let $L_1, \dots, L_k$ be k arbitrary lists (with the assumption that all numbers in the lists are distinct). We wish to show that Merge-Lists, on input $L_1, \dots, L_k$, merges them into one single sorted list.

- *Base Case*: for the base case we have $k = 1$. In this case, L_1 is already sorted and Step 2 of Algorithm 1 simply returns the single list L_1.
- *Inductive Hypothesis*: assume Algorithm 1 correctly merges m sorted lists, for every $m < k$, into one single sorted list.
- *Inductive Step*: from the inductive hypothesis, it follows that A_1 and A_2 are sorted lists because they merge $\lfloor k/2 \rfloor < k$ and $\lceil k/2 \rceil < k$ sorted lists, respectively. Since the Merge subroutine correctly merges two sorted lists into a single sorted list, A_1 and A_2 are correctly combined into a single sorted list. We conclude then that Algorithm 1 correctly merges the k sorted lists into a single sorted list. □

(d) *Algorithm Analysis*

Let $T(n, k)$ be the total running time of merging k sorted lists with a total of n elements. We have $T(n, k) = \mathcal{O}(n)$ for $k = 1$; note L_1 in Step 2 of Algorithm 1 will contain exactly n elements requiring potential copies so it is prudent to assume it takes time proportional to n at the base case. For the recursion, A_1 is a list of size n_1 and A_2 is a list of size n_2 where $n_1 + n_2 = n$. So merging them takes $\mathcal{O}(n)$ time. Thus, the recurrence becomes:

$$T(n,k) = \begin{cases} n & \text{if } k = 1 \\ T(n_1, \lfloor k/2 \rfloor) + T(n_2, \lceil k/2 \rceil) + n & \text{otherwise} \end{cases}$$

To solve this recurrence, we proceed with the recursion tree approach. For simplicity assume k is a power of 2. Then the recursion tree is a complete binary tree with k nodes at the leaves and depth

$\log k$. The work at the root node is n. At the next level is $n_1 + n_2$ which is n. Each level incurs an $\mathcal{O}(n)$ cost and there are $\log k$ levels hence the total work is $\mathcal{O}(n \log k)$. $\square$

2. (a) $(4, 2), (4, 1), (2, 1), (9, 1), (9, 7)$.

(b) The array with entries from the set $\{1, 2, \dots, n\}$ with the most inversions will have their entries in "backward," reversed sorted order: $\{n, n-1, \dots, 1\}$. This is because every single pair of array indices is inverted, so we get $n-1$ inversions with the first entry, $n-2$ inversions with the second entry, $n-3$ inversions with the third entry, and so on. In general, for a fixed index i we will have $n-i$ inversions. So the total number of inversions over all possible choices of i will be:

$$\sum_{i=1}^{n} (n-i) = (n-1) + (n-2) + \cdots + 1 + 0 = \frac{n(n-1)}{2}$$

(c) 1. *Algorithm Description*

We proceed with a "Divide and Conquer" strategy motivated by Merge-Sort. Specifically, for pairs of array indices (i, j) with $i < j$, we distinguish three inversion cases: (1) a left-inversion if $i, j \leq \lfloor n/2 \rfloor$, (2) a right-inversion if $i, j \geq \lceil n/2 \rceil$, and (3) a cross-inversion if $i \leq \lfloor n/2 \rfloor$ and $j \geq \lceil n/2 \rceil$. Similar to Merge-Sort, we start by splitting the array into two (nearly equal) halves. We then recurse on the left and count all the inversions that are restricted to the first half of the array. These are precisely the left-inversions. Then we invoke a second recursive call on the right to count all the right-inversions. Finally, we delegate the residual work left over (i.e. counting the number of cross-inversions) to a modified version of the Merge subroutine (see Algorithm 2 for details). Since every inversion is either left or right or cross and cannot be any more than one of these three, we can simply report the sum of the results of the two recursive calls plus the output from the aforementioned, modified Merge subroutine.

2. *Algorithm Pseudocode*

Algorithm 2 Reports the total number of inversions in $A[p \cdots q]$

```
 1: function COUNT-INVERSIONS($A, p, q$)
 2:     if $p \geq q$ then
 3:         return 0
 4:     end if
 5:     $m \leftarrow \lfloor (p+q)/2 \rfloor$
 6:     $l \leftarrow$ COUNT-INVERSIONS($A, p, m$)                    // count left-inversions
 7:     $r \leftarrow$ COUNT-INVERSIONS($A, m+1, q$)                  // count right-inversions
 8:     $c \leftarrow$ CROSS-INVERSIONS($A, p, m, q$)                 // count cross-inversions
 9:     return $l + c + r$
10: end function
11:
12: function CROSS-INVERSIONS($A, p, m, q$)                          // modified MERGE subroutine
13:     $n_1 \leftarrow m - p + 1$
14:     $n_2 \leftarrow q - m$
15:     $L[1 \cdots n_1] \leftarrow A[p \cdots m]$
16:     $R[1 \cdots n_2] \leftarrow A[m+1 \cdots q]$
17:     $i \leftarrow 1$
18:     $j \leftarrow 1$
19:     $c \leftarrow 0$                                             // counts # of cross-inversions
20:     for $k \leftarrow p$ to $q$ do
21:         if $n_1 < i$ then            // if we exhaust $L$ then copy unexhausted items of $R$ into $A$
22:             $A[k] \leftarrow R[j]$
23:             $j \leftarrow j + 1$
24:         else if $n_2 < j$ then       // if we exhaust $R$ then copy unexhausted items of $L$ into $A$
25:             $A[k] \leftarrow L[i]$
26:             $i \leftarrow i + 1$
27:         else if $L[i] \leq R[j]$ then                            // this does not cause an inversion
28:             $A[k] \leftarrow L[i]$
29:             $i \leftarrow i + 1$
30:         else                                                     // but this does!
31:             $A[k] \leftarrow R[j]$
32:             $j \leftarrow j + 1$
33:             $c \leftarrow c + (n_1 - i + 1)$     // inversion pairs $(L[i], R[j]), (L[i+1], R[j]), \ldots, (L[n_1], R[j])$
34:         end if
35:     end for
36:     return $c$
37: end function
```

3. *Algorithm Correctness*

   We use (strong) induction on n, the number of elements in A, to demonstrate the correctness of Algorithm 2. Recall the number of elements in $\{A[p], A[p+1], \ldots, A[q]\}$ is $q-p+1$, and henceforth, we will denote this by $|A|$.

   - *Base Case*: when $|A| \leq 1$, notice the number of inversions in A is 0. If A contains at most one element then $(p \geq q)$ and Step 3 of Algorithm 2 gets executed—reporting 0 as its output. It follows then that Algorithm 2 executes correctly as intended and produces the desired, expected output when $|A| \leq 1$.

   - *Induction Hypothesis*: assume that Algorithm 2 reports the correct number of inversions when $|A| \leq k$, for every $k = 2, 3, \ldots, n-1$, and in the process sorts A from left to right.

   - *Induction Step*: now consider when $|A| = n$. Step 5 of Algorithm 2 divides A into two subarrays of nearly equal sizes. Let L represent the subarray $A[p \cdots m]$ and R represent the subarray $A[m+1 \cdots q]$. Clearly $|L| < n$ and $|R| < n$. Thus, applying the inductive hypothesis, we are guaranteed that the number of inversions produced in Steps 6 and 7 are correct, i.e. l stores the correct number of inversions in L and r stores the correct number of inversions in R. Moreover, based on the induction hypothesis, L and R are now sorted so there are no longer any inversions in L nor in R, only inversions between L and R. So let's proceed to analyze the Cross-Inversions subroutine, which we delegated earlier to handle precisely the counting of the cross-inversions between L and R. Recall that if an element R_j from the list R is selected ahead of m items from list L, then R_j is less than the remaining m elements from list L, corresponding to m inversions. Additionally, selecting R_j to be placed as the next element in the new combined, sorted list eliminates all m inversions. So no new inversions are created in the merging step of the Cross-Inversions subroutine. Consequently, no inversions are counted twice, since removing an invertion merges L and R into one combined list, and inversions can only exist between lists. Therefore, Cross-Inversions, along with the two recursive calls, correctly account for all inversions in A. □

4. *Algorithm Analysis*

   Let $T(n)$ represent the total running time of Algorithm 2 on an input of size n. At the base case, we have $T(n) = 1$ for $n = 1$. For the recursion, we have two subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, and since we are piggy-backing on the Merge subroutine, we have a combine step of $\mathcal{O}(\lfloor n/2 \rfloor + \lceil n/2 \rceil) = \mathcal{O}(n)$. Thus, the recurrence becomes: $T(1) = 1$, $T(n) = 2T(n/2) + n$. With $a = 2$, $b = 2$, and $f(n) = n$, this recursion solves to $\Theta(n \log n)$, according to case 2 of the Master Theorem discussed in class.

3. (a) $T(n) = 2T(n/2) + n^3$. Here $a = 2$, $b = 2$, $f(n) = n^3$, and $n^{\log_b a} = n^{\log_2 2} = n$. For $\varepsilon = 2$, we have $f(n) = \Omega(n^{\log_2 2+\varepsilon})$. So case 3 of the Master Theorem applies if we can show that $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n. This would mean $(1/4)n^3 \leq cn^3$. Setting $c = 1/3$ would cause this condition to be satisfied. Hence $T(n) = \Theta(n^3)$.

(b) $T(n) = 4T(n/2) + n\sqrt{n}$. Here $a = 4$, $b = 2$, $f(n) = n\sqrt{n}$, and $n^{\log_b a} = n^{\log_2 4} = n^2$. Since $f(n) = \mathcal{O}(n^{\log_2 4-\varepsilon})$ for $\varepsilon = 0.5$, case 1 of the Master Theorem applies, and the solution is $T(n) = \Theta(n^2)$.

(c) $T(n) = 2T(n/2) + n\log n$. To solve this recurrence, we can simply expand the recurrence as follows (ommitting straighforward simplifications):

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n\log n \\
&= 4T\left(\frac{n}{4}\right) + n\left(\log n + \log\frac{n}{2}\right) \\
&= 8T\left(\frac{n}{8}\right) + n\left(\log n + \log\frac{n}{2} + \log\frac{n}{4}\right) \\
&= \qquad\qquad \vdots \\
&= 2^kT\left(\frac{n}{2^k}\right) + n\sum_{i=0}^{k-1}\log\frac{n}{2^i} \\
&= 2^kT\left(\frac{n}{2^k}\right) + n\left[\sum_{i=0}^{k-1}\log n - \sum_{i=0}^{k-1}\log 2^i\right] \\
&= 2^kT\left(\frac{n}{2^k}\right) + n\left[\sum_{i=0}^{k-1}\log n - \sum_{i=0}^{k-1} i\right] \\
&= 2^kT\left(\frac{n}{2^k}\right) + n\left[k\log n - \frac{k(k-1)}{2}\right] \\
&= 2^kT\left(\frac{n}{2^k}\right) + nk\left(\log n - \frac{k}{2} + \frac{1}{2}\right)
\end{aligned}$$

The expanding stops once $n/2^k$ is equal to 1, which means $k = \log n$. When $k = \log n$, we have:

$$\begin{aligned}
T(n) &= 2^{\log n}\, T\left(\frac{n}{2^{\log n}}\right) + n\log n\left(\log n - \frac{1}{2}\log n + \frac{1}{2}\right) \\
&= nT(1) + n\log n\left(\frac{1}{2}\log n + \frac{1}{2}\right) \\
&= n + \frac{1}{2}n\log^2 n + \frac{1}{2}n\log n \\
&= \mathcal{O}(n\log^2 n)
\end{aligned}$$

Therefore, we conclude $T(n) = \mathcal{O}(n\log^2 n)$.

(d) $T(n) = T(3n/4) + n$. We guess the solution is $T(n) = \mathcal{O}(n)$. According to the definition of big-$\mathcal{O}$ notation, we want to find a constant $c > 0$ and an integer $n_0 \geq 1$ such that $T(n) \leq cn$ for all sufficiently large $n \geq n_0$. We proceed with (strong) induction on n and assume $T(n) \leq cn$ is true for all positive numbers less than n. Then $T(3n/4) \leq (3/4)cn$ and $T(n) \leq (3/4)cn + n$. Hence, in order to prove $T(n) \leq cn$, it is sufficient to prove that $(3/4)cn + n \leq cn$, or equivalently, $(3/4)c + 1 \leq c$. It is easy to see that any $c \geq 4$ suffices. So for any choices of $c \geq 4$ and $n_0 = 1$, $T(n) \leq cn$ holds for all sufficiently large $n \geq n_0$. Therefore, $T(n) = \mathcal{O}(n)$.

4. (a) *Algorithm Description*

We proceed with a "Divide and Conquer" strategy and split the daily stock prices into two (nearly equal) halves. In doing so, we distinguish three possible cases for the optimal buy and sell times: (1) optimal buy and sell times lie exclusively in the first half of the split; (2) optimal buy and sell times lie exclusively in the second half of the split; and (3) optimal buy time lies in the first half of the split and optimal sell time lies in the second half of the split.

Similar to the inversion problem in 2, we can obtain the corresponding values of (1) and (2) by simply recursing on the left and right halves of the array. However, for (3) we should strategize buying on the lowest possible day on the left and selling on the highest possible day on the right. Our tentative solution then becomes as follows: if the array has size 0 or 1, the maximum profit is 0; otherwise, we split the array in half and compute the maximum profit on the left, and call it L_{profit}. We compute the maximum profit on the right, and call it R_{profit}. We find the minimum on the left, and call it L_{min}. We find the maximum on the right, call it R_{max}. We report the maximum of L_{profit}, R_{profit}, and $R_{max} - L_{min}$ as our final output.

In analyzing the time complexity of our solution, it is evident that our recurrence has a base case that incurs an $\mathcal{O}(1)$ cost. However, for its recursive step, we have two recursive calls on a subproblem half as large as the original input followed by an additional linear cost incurred by the task of finding the maximum and minium values across the left and right. Thus, the recurrence becomes: $T(1) = 1$, $T(n) = 2T(n/2) + n$, which solves to $\Theta(n \log n)$—which is not quite what we want.

The key idea to improve this is to delegate the linear task of finding the maximum and minimum values to the recursive calls. This solution works just as before, but now it never incurs an $\mathcal{O}(n)$ cost at each step to find the minimum and maximum values. In fact, it only incurs $\mathcal{O}(1)$ work at each level. So we get a new recurrence of the form: $T(1) = 1$, $T(n) = 2T(n/2) + 1$, which solves to $\mathcal{O}(n)$, as desired. The pseudocode for this is given below.[1]

(b) *Algorithm Pseudocode*

Algorithm 3 Maximum Single Sell Profit

```
function MAX-SINGLE-SELL-PROFIT($A, p, q$)          // maximum single sell profit from $A[p \cdots q]$
    if $p = q$ then
        return $(0, A[p], A[q])$
    end if
    $m \leftarrow \lfloor (p+q)/2 \rfloor$
    $(L_{profit}, L_{min}, L_{max}) \leftarrow$ MAX-SINGLE-SELL-PROFIT($A, p, m$)
    $(R_{profit}, R_{min}, R_{max}) \leftarrow$ MAX-SINGLE-SELL-PROFIT($A, m+1, q$)
    $x \leftarrow$ MAX($L_{profit}, R_{profit}, R_{max} - L_{min}$)
    $y \leftarrow$ MIN($L_{min}, R_{min}$)
    $z \leftarrow$ MAX($L_{max}, R_{max}$)
    return $(x, y, z)$
end function

function QUERY-MAX-PROFIT-INDICES($A, x$)          // classic two-sum problem
    $i \leftarrow 1$
    $j \leftarrow 1$
    $\mathcal{H} \leftarrow \{\}$          // an amortized constant time lookup table
    for $k \leftarrow 1$ to $n$ do
        $y \leftarrow A[k] - x$
        if $y$ in $\mathcal{H}$ then
            $i \leftarrow \mathcal{H}[y]$
            $j \leftarrow k$
            return $(i, j)$          // buy on day $i$, sell on day $j$
        end if
        $\mathcal{H}[A[k]] \leftarrow k$
    end for
    return $(i, j)$          // here $i = j = 1$ so no way to make money
end function
```

[1]Note that this solution returns the maximum profit, not the indices/days it occurs. To get the indices/days, we resort to the classic "two-sum" problem, which can conviniently be solved in $\mathcal{O}(n)$ time with auxiliary linear space. Consult Algorithm 3 for details.

(c) *Algorithm Correctness*

We prove the correctness of Algorithm 3 by using (strong) induction on n—the number of permissible days to buy and sell stocks. Note the number of days in $\{A[p], A[p+1], \dots, A[q]\}$ is $q - p + 1$ and we denote by $|A|$.

- *Base Case*: when $|A| = 1$ the maximum profit is 0 and $q - p + 1 = 1$. So $p = q$ and Step 3 of Algorithm 3 gets executed, reporting 0 in its first entry. It follows then Algorithm 3 works correctly as intended and produces the desired, expected output when $|A| = 1$.
- *Induction Hypothesis*: assume that Algorithm 3 reports the correct maximum profit along with the minimum and maximum values on the left and right when $|A| \leq k$, for every $k = 2, \dots, n-1$.
- *Induction Step*: then it follows from the induction hypothesis that Step 6 and 7 of Algorithm 3 report the correct maximum profit on the left and right (L_{profit}, R_{profit}), as well as the corresponding minimum and maximum values within them ($L_{min}, L_{max}, R_{min}$, and R_{max}). This is true because on the left $|A| = \lfloor n/2 \rfloor < n$ and on the right $|A| = \lceil n/2 \rceil < n$, and the induction hypothesis along with the base case guarantees the correct output for all problem sizes strictly less than n. Now when $|A| = n$, the location of the maximum profit does not deviate from our initial assumption, i.e. it is either found on the left (in which case L_{max} is the correct output), or on the right (in which case R_{max} is the correct output) or in between, in which case the difference $R_{max} - L_{min}$ is the correct output. But Step 8 of Algorithm 3 computes precisely the maximum among these values, which means for a problem of size n, we have accounted for all possible locations of the maximum profit, and picked the maximum one. So, if there is one, Algorithm 3 always outputs the correct maximum profit. □