Homework 2

1a. Give a linear time algorithm to complete the merge step of a three-way sort.

Given three sorted subarrays A, B, and C, create a fourth array D large enough to hold all elements in A, B, and C. Create three pointers a, b, c, at the first elements of A, B, and C, respectively.

Next, compare a and b, and then the smaller of a or b with c. The result will be the smallest element of {a, b, c}, append this value to D and increment the pointer for the value added to the next element in the list. Repeat this process until all of the elements have been added to D.

This algorithm requires two comparisons and an increment for each element sorted. Therefore the approximate running time for n elements is 3n, or O(n).

1b. Model the running time of the three-way merge sort using a recurrence.

Divide step: O(1) Conquer step: 3 * T(n / 3) Merge step: O(n)
Recurrence: O(1) + 3 * T(n / 3) + O(n) => **3 * T(n /3) + n + 1**

1c. Solve the recurrence relation to obtain the running time of 3-way Merge Sort.

$3 * T(n/3) + n + 1$
$= 3 * 3 * T(n / (3 * 3)) + n + n + 1 + 1$
$= 3 * 3 * 3 * T(n / (3 * 3 * 3)) + n + n + 1 + 1$
$= 3^k * T(n / 3k) + kn + k$ where $k = \log_3(n)$
$= 3^{\log_3(n)} * T(1) + n * \log_3(n) + \log_3(n)$
$= n + 1 + n * \log_3(n) + \log_3(n)$
$= O(n * \log_3(n))$

2a. List all inversions of the array [14, 12, 17, 11, 19]

(14, 12); (14, 11); (12, 11); (17, 11);

2b. What subset has the most inversions? How many inversions does it have?

The array containing all elements {1, 2, … , n} in descending order has the most inversions. It has (n - 1) + (n - 2) + (n - 3) + … + 1 inversions.

2c. Give a divide-and-conquer algorithm that computes the number of inversions in an array A in O(n * log(n)) time.

Given a merge sort that operates in-place on A that divides A into subarrays of size 1 (such as the example we did in class), simply create a variable to hold the number of inversions and increment this variable during the merge step of the sort whenever a number from the higher indexed subarray is chosen instead of a number from the lower-indexed subarray. If you pass this variable as a parameter by reference, after the end of the sort it will hold the total number of inversions. Because it is well-known that merge sort operates in O(n * log(n)) time, and because incrementing a value during the merge step only adds constant time to each iteration, it follows that this algorithm will also operate in O(n * log(n)) time.

3a. Solve $T(n) = 2 * T(n / 2) + n^4$.

$= 2 * 2 * T(n / 4) + n^4 + n^4.$
$= \ldots$
$= 2^k * T(n / 2^k) + kn^4.$ Where k = log(n)
$= 1 * T(1) + \log(n)n^4.$
$= O(n^4 * \log(n)).$

3b. Solve $T(n) = 4 * T(n / 2) + n$.

$= 4^k * T(n / 2^k) + kn$ Where k = log(n)
$= 2 * T(1) + n * \log(n)$
$= O(n * \log(n)).$

3c. Solve $T(n) = 2 * T(n / 2) + n * \log(n)$.

$= 2^k * T(n / 2^k) + k * n * \log(n)$ Where k = log(n)
$= 1 * T(1) + n * \log^2(n)$
$= O(n \log^2(n)).$

3d. Solve $T(n) = T((2 / 3) * n) + n$.

$= T((2 / 3)^k * n) + kn$

$(2 / 3)^k * n = 1$
$(2 / 3)^k = n^{-1}$
$\log_{(2/3)}(n^{-1}) = k$

$= T(1) + \log_{(2/3)}(n^{-1}) * n$
$= O(n * \log_{(2/3)}(n^{-1}))$

4. Solve the given problem in O(n * log(n)) time.

We will create an algorithm that recursively finds the largest difference between consecutive numbers. Given an array A of n numbers, we begin by recursively dividing the array up into subarrays of size $n \leq 2$. At this level, we define two base cases. If n = 1, we return an ordered pair containing that single number twice. If n = 2, we check if the numbers are sorted in ascending order or not. If they are, we return those two numbers in an ordered pair. If they are not in ascending order. We return a value to indicate that there is no way to make money for those numbers.

With our base cases handled, we then create the recursive step, dividing the array A into half repeatedly until the base cases are reached. Once this step has completed, we will have one of several cases:

Case 1: Both halves of the recursion return false

This is a trivial case, if both halves of the recursion return false we also return false.

Case 2: One half returns false, the other half does not.

Also simple, if one half of the given array has a valid solution, we return it.

Case 3: Both halves return valid pairs

This is the complex case, and we must break the problem into subcases. For now, let firstPair represent the pair returned by the first half of the recursive step, and secondPair represent the pair returned by the second half.

Case 3a: The union of ranges in firstPair and secondPair is a valid pair

If the smaller member of firstPair is smaller than the smaller member of secondPair, and if the larger value of firstPair is smaller than the larger value of secondPair, it then follows that the largest difference is found between the smallest member of firstPair and the largest member of secondPair.

Case 3b: One of the two pairs is contained within the other pair

If one of the two pairs has both a smaller start and a larger end than the other pair, we can simply return that larger pair as the biggest difference.

Case 3c: The range of the two pairs, but do not form a valid pair

Here we have two valid, incompatible ranges. This is the naive case, we simply return the pair with the largest difference between its members.

If we allow this recursive algorithm to run to completion, we will be left with a pair of entries representing the greatest increase in price over the time period, or a pair signifying that no answer could be found (either the special value, or a pair containing the same entry for both the high and the low). To illustrate this algorithm, some C++ style pseudocode can be found below.

```
pair<int, int> getRange(array<int> A) {

      // Base Cases
      if (A.size() == 1) { return pair(A[0], A[0]); }
      if (A.size() == 2) {
            if (A[0] < A[1]) { return pair(A[0], A[1]); }
            else { return pair(-1, -1)); }
      }


      // Recursive Step
      firstPair = getRange(array(A.begin(), A.mid()));
      secondPair = getRange(array(A.mid(), A.end()));


      // Trivial Cases
      if (firstPair == pair(-1, -1) && secondPair == pair(-1, -1))
            { return pair(-1, -1); }

      if (firstPair != pair(-1, -1) && secondPair == pair(-1, -1))
            { return firstPair; }

      if (firstPair == pair(-1, -1) && secondPair != pair(-1, -1))
            { return secondPair; }


      // Given firstPair = (a, b), secondPair = (c, d)
      // If (a, d) is a valid pair
      if (firstPair[0] < secondPair[0] && firstPair[1] < secondPair[1])
            { return pair(firstPair[0], secondPair[1]); }

      // If (c, d) is contained within (a, b)
      if (firstPair[0] < secondPair[0] && firstPair[1] > secondPair[1])
            { return firstPair; }

      // if (a, b) is contained within (c, d)
      if (firstPair[0] > secondPair[0] && firstPair[1] < secondPair[1])
            { return secondPair; }
```

```
	// Otherwise we have competing pairs.
	else {
		firstDifference = firstPair[1] - firstPair[0];
		secondDifference = secondPair[1] - secondPair[0];

		if (firstDifference >= secondDifference) { return firstPair; }
		else { return secondPair; }
	}
}

// Given our input A
auto resultPair = getInput(A);
if ((resultPair == pair(-1, -1)) || (resultPair[0] == resultPair[1])) {
	printf("There was no way to make money in the provided days.");
}
else {
	printf("The most money could have been made by buying on day %d and
selling on day %d", days.find(resultPair[0]), days.find(resultPair[1]));
}
```