Keenan Miller

1. Donut Problem
    a. My algorithm to determine the minimum sum of distances between a corner donut shop and a given set of police cars first has to determine the best location for the donut shop. To do this it takes the mathematical median of all cars' locations, by finding the medians of x and y coordinates (if the set of cars locations is an even number, and since we must use whole-numbers, all four possible locations are considered). Once the median of all car locations is determined, the Manhattan distance is calculated between each given car and the determined median, and the sum is found. If we have multiple possible locations, each sum of distances is found, and the minimum of these is determined. The resulting sum of distances is returned and outputted.
    b. Pseudocode:

```
// average time complexity O(n)
getMedian(int[] array, k) {
	pivot = random entry from array;
	l[] = all numbers from array less than pivot;
	g[] = all numbers from array greater than pivot;
	e[] = all number from array equal to pivot;

	if(k == length of l) {
		return pivot
	} else if(k >= l.length + e.length) {
		return getMedian(g, k - l.length - e.length)
	} else {
		return getMedian(l, k)
	}
}

getBestLocation(points, boolean maxMedianX, boolean maxMedianY) {
	// if we want the "larger" median from either x or y coordinates
	if(maxMedianX) {
		add Integer.MAX_VALUE to points.x
	}
	if(maxMedianY) {
		add Integer.MAX_VALUE to points.y
	}
	return(new Point(getMedian(points.x), getMedian(points.y))
}

getSumDistance(points, location) {
	int sum;
	for(points) {
		// manhattan distance (ABS is absolute value)
		sum += ABS(point.x - location.x) + ABS(point.y - location.y);
```

```
	}
}

main() {
	read in points;
	if(points.length is even) {
		// this means there are 2 medians for both axis' values (and 4 locations)
		possibleLocations[] = {
			getBestLocation(points, false, false),
			getBestLocation(points, false, true),
			getBestLocation(points, true, false),
			getBestLocation(points, true, true)
		}
	} else {
		possibleLocations[] = {getBestLocation(points, false, false)}
	}

	int result = MAX_VALUE;
	for(possibleLocations) {
		result = min(result, getSumDistance(points, possibleLocation));
	}
	print(result);
}
```

c. This algorithm is provably correct because since we take the median of each police car's location, this will always be the best location. Of course if we have an even number of cars to consider, there are multiple (4) locations to consider, so we consider each one equally. Once we have the set of best possible locations, each location has its distances to each car added to find the sum of distances. The minimum of these sums of distances is taken, guaranteeing that we have the best location.
d. This algorithm runs at O(n) average running complexity.
e. We can prove this algorithm runs at O(n) because each location is only ever iterated over using a single, non-nested for loop. Each of these loops is sequential to the next, therefore not increasing the time complexity. Since we take the median using the selection algorithm discussed in class, this finding the median runs at O(n) average running time, and is not nested within any loops, and doesn't increase the overall time complexity.

2. Greedy Longest Increasing Subsequence
    a. Pseudocode (approach 1):

```
findSubsequence(array, count) {
	if(array.length == 0){
		return count;
	}
	min = MAX_VALUE;
```

```
	for i in array {
		if(array[i] < min){
			min = array[i];
		}
	}
	newArray = subarray from min to end of array;
	return findSubsequence(newArray, count++);
}

main(array) {
	subsequence = findSubsequence(array, 0);
	print(subsequence.length);
}
```

b. Since this algorithm loops over the passed (increasingly smaller array) each recurse, and it only recurses once in sequence with each loop, the time complexity of this algorithm should be O($n^2$)

c. This algorithm is incorrect because it will almost certainly always produce an incorrect result.
   i. Counterexample: input {2, 3, 4, 1}
   ii. For this input, the correct maximum increasing subsequence is {2, 3, 4} with length 3.
   iii. The algorithm will begin by finding the minimum value (1) and continue from there determining the result of {1} with length 1 which is not the maximum increasing subsequence length.

a. Pseudocode (approach 2):

```
findMaxSubsequenceFrom(array, n) {

	if(n == array.length - 1)
		return 0;
	min = n + 1;
	for i=n in array {
		if(array[min] < array[i] > array[n]){
			min = i;
		}
	}
	return findMaxSubsequenceFrom(array, min) + 1;
}

main(array) {
	result = 0;
	for i in array {
		result = max(result,findMaxSubsequenceFrom(array, i));
		print(result);
	}
}
```

b. This algorithm runs with complexity O($n^3$) because it considers only one possible

next value for each given start value each recursive call, but this happens within an additional loop, causing the complexity to increase on another order of n. It is essentially a nested loop that recurses within one of the loops, causing it to run with O($n^3$) time complexity.

c. This algorithm doesn't work either.
   i. Counterexample: {1, 3, 5, 2}
   ii. For this input, the correct maximum increasing subsequence is {1, 3, 5}, with length 3.
   iii. The algorithm will not determine this because when it attempts to find the maximum increasing subsequence beginning at 1, it will attempt to use the minimum value in the sequence that is greater than 1, which is 2. This iteration will return {1, 2}. The second iteration will begin at 3, and return {3, 5}. The third and fourth iterations will return, respectively, {5}, and {2}. The maximum length of these found subsequences is only 2 which is the incorrect answer.

3. Recursive vs Dynamic Programming: Longest Increasing Subsequence
   a. Recursive approach:
      i. This algorithm loops through each value in the given input array and recursively finds the maximum increasing subsequence that ends at the value given. In each recursive iteration another loop is performed that checks if the next value is less than the ending value of the possible subsequence. If the value is smaller, another recursive iteration is started that follows the same process. The longest subsequence is found and the length is outputted.
      ii. Pseudocode:

```
findMaxSubsequenceTo(array, n) {
	max = 0;
	for i in array {
		if(array[i] < numbers[n]){
			max = max(max, findMaxSubsequenceTo(array, i) + 1);
		}
	}
	return max;
}

main(array) {
	result = 0;
	for i in array {
		result = max(result,findMaxSubsequenceTo(array, i));
		print(result);
	}
}
```

iii. This algorithm runs with complexity O(n!) because it considers (and recalculates) every possible legal subsequence. It recurses once within each loop, and loops once during each recursive loop, meaning the time complexity grows exponentially.

iv. This algorithm works, albeit very slowly for larger inputs (and could even overflow the stack), because it is able to check each possible subsequence. Because it checks each possible subsequence, it will not leave out any candidates and will succeed given enough time and a large enough stack.

b. Dynamic Programming Approach

i. The dynamic programming approach to solving this problem uses a similar method to finding the true maximum increasing subsequence as the recursive approach with some key differences. For one, it begins at the start of the input array, rather than working from the end. The second and most important difference is that instead of recursively re-solving smaller problems like the recursive approach, this algorithm stores the results from subproblems for later use.

ii. Pseudocode:

```
main(array) {
	for i in array {
		OPT[i] = 1;
		for j from 1 to i {
			if array[j] < array[i] and OPT[i] <= OPT[j]) {
				OPT[i] = OPT[j] + 1'
			}
		}
	}
	return max value in OPT;
}
```

iii. This algorithm will always produce the correct answer because it checks every possible subsequence. This simply will produce the correct answer because each largest legal subsequence is kept track of for each ending index of the array. Once the array has been fully checked, the length of the largest subsequence is outputted.

iv. This algorithm runs with O($n^2$) complexity.

v. This algorithm runs with O($n^2$) complexity because it does not employ recursion and simply loops over a given array within a double-nested loop.

c. Data:

i. Below is data and a logarithmic scatter plot of different sized input values and their running times (in ms) for both recursive and DP algorithms to find the maximum increasing subsequence of input values. The data is surprising how fast the recursive algorithm fails for increasing input sizes. The algorithm can't seem to handle inputs much larger than n=150, while

the DP approach can handle much larger inputs. The recursive algorithm seems to fail completely at producing a result before the DP approach even begins to register noticeable running times. The recursive algorithm is obviously much more inefficient than the DP algorithm, which can handle inputs on orders of magnitude greater than the former. The given running times also indicate that the time complexity determined through the analysis of each algorithm is correct; respectively O(n!) and O($n^2$).

| n | Recursive Approach (ms) | DP Approach (ms) |
|---|---|---|
| 10 | 0 | 0 |
| 25 | 1 | 0 |
| 50 | 4 | 0 |
| 75 | 14 | 1 |
| 100 | 453 | 1 |
| 150 | 26474 | 1 |
| 250 | N/A | 1 |
| 500 | N/A | 2 |
| 1000 | N/A | 3 |
| 5000 | N/A | 39 |
| 10000 | N/A | 173 |