Analysis of Algorithms

Alvin Lin (axl1439) and William Leuschner (wel2138)

August 2017 - December 2017

Problem 0

Algorithm Description

For this problem, we implemented the select linear time median algorithm to find the median of all the points since the median point would have the smallest total Manhattan distance to all the police cars. The integer median guarantees us a point in the neighborhood of the optimal solution, so we simply check the immediate points surrounding our calculated median to find the optimal solution. This brute force check adds a small constant time to our runtime.

Argument of Correctness

Finding the centroid (arithmetic mean of points) does not work because if there is an outlier cop car, it will skew the Manhattan distance sums towards that outlier, which increases the travel time from the main cluster of cop cars. Choosing the median allows us to center the point within the cluster such that the travel time by any car in the cluster is minimized. Note that this is true because travel time is calculated as Manhattan distance.

Running Time Estimate

This algorithm implements the select algorithm to do a linear time median.

$$O(n)$$

Psuedocode

```
def kthSmallest(values, k):
    n = values.length
    if n <= 5:
        return bruteForceKthSmallest(values, k)
    medians = []
    j = 0
    for i = 0 to values.length by 5:
        medians[j] = bruteForceMedian(values[i:i+5])
    medianB = kthSmallest(medians)

    partition = []
    left = 0, right = n - 1
    for i = 0 to values.length:
        if values[i] < medianB:
            partition[left] = values[i]
            left += 1
        else if values[i] > medianB:
            partition[right] = values[i]
            right -= 1
    for i = left to right + 1:
        partition[i] = medianB

    pivot = left
    if pivot != right and right < n / 2:
        pivot = right

    if k < pivot:
        return kthSmallest(partition[0:pivot], k)
    else if k > pivot:
        return kthSmallest(partition[pivot + 1:n])
    return medianB

def sumManhattanDistance(coordinates, x, y):
    sum = 0
    for i = 0 to coordinates.length
        sum += |coordinates[0] - x| + |coordinates[1] - y|
    return sum

def minimizeSumDistance(coordinates):
    n = coordinates.length
    medianX = kthSmallest(coordinates[0], n / 2)
    medianY = kthSmallest(coordinates[1], n / 2)
    distance = sumManhattanDistance(coordinates, medianX, medianY)
    neighborhood = [
```

```
        [-1, 1], [-1, 0], [-1, 1]
        [0, -1], [0, 1],
        [1, -1], [1, 0], [1, 1]
    ]
    for offset in neighborhood:
        distance = min(distance, sumManhattanDistance(
            coordinates, medianX + offset[0], medianY + offset[1]))
    return distance
```

Problem 1

Algorithm Description

For this problem, we used a dynamic programming solution to find the maximum number of classes that can fit into a schedule, allotting for travel time. During each step, it computes the maximum number of classes that can be taken from among the current subset of classes. It finds the largest previous solution, then adds one and sets that value as the current solution, but only if that largest solution fit before the current interval. If it was unable to find a previous solution that fit before the current interval, then the current solution is simply one, consisting only of the the current interval. Heart of the solution:

$$\texttt{S[i]} = \text{the maximum number of non-overlapping classes (including travel time) from the set } a_1 \dots a_j$$

To calculate `S[i]`:

$$\text{Let } \texttt{index} = \texttt{max(S[:i-1])} \text{ such that } a_{\texttt{index}}.end + \text{travel time} \leq a_{\texttt{i}}.start$$

$$\texttt{S[i]} = \texttt{S[index]} + 1 \text{ if such an index exists, otherwise } 1$$

$$\texttt{S[1] = 1}$$

Running Time Estimate

This algorithm needs to iterate through each interval, and during each iteration needs to find the maximum previous solution. This makes it $O(n^2)$.

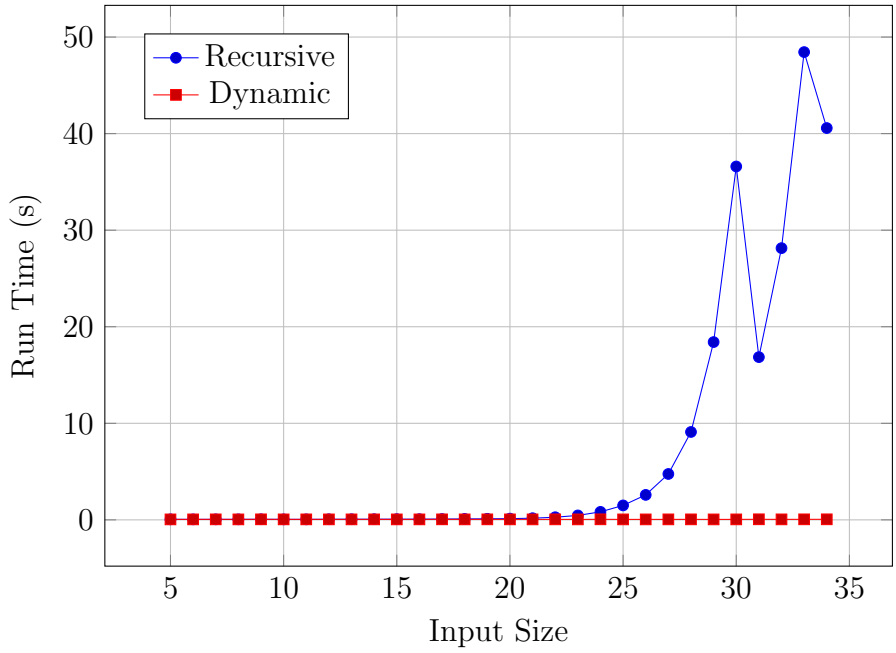
Pseudocode

```
def getEndTimeWithTravel(startClass, endClass, travelTimes):
    return startClass.endTime +
        travelTimes[startClass.index][endClass.index]

def findGreatestNumberOfIntervals(intervals, travelTimes):
    intervals = heapSort(intervals, keeping track of original index)
    let solution = [] of size intervals.length
    solution[0] = 1
    for i = 1 to intervals.length:
        let current = intervals[i]
        solution[i] = 1
        for j = 0 to i:
            tmp = intervals[j]
            if solution[j] >= solution[i] and
                getEndTimeWithTravel(tmp, current, travelTimes):
                solution[i] = solution[j] + 1
    return max(solution)
```

Problem 2

For this problem, we implemented both the recursive solution and the dynamic programming solution.



The recursive solution is significantly slower than the dynamic programming solution.

Recursive Solution Psuedocode

```
def getLongestIncreasingSubsequenceLength(sequence):
    maxLen = 0
    for i = sequence.length - 1 to 0:
        maxLen = max(incrSubseqRecursive(j, sequence).length, maxLen
           )
    return maxLen

def incrSubseqRecursive(index, A):
    possibilities = new max heap of arrays comparing by length
    for i = index - 1 to 0:
        if A[i] < A[index]:
            possibilities.add(incrSubseqRecursive(i, A))
    if not possibilities.empty():
        answer = possibilities.pop()
    else:
        answer = []
    answer.add(A[index])
    return answer
```

Out of curiosity, we implemented memoization into the recursive algorithm just to see what would happen. It drastically cut down on the runtime and it was comparable to the runtime of the dynamic programming solution, but still slower. The main thing slowing down the recursive algorithm is simply the fact that it has to recompute the previous solutions, unlike the dynamic programming solution.

Problem 3

Part 1 Psuedocode

```
def longestIncreasingSubsequence(sequence, start):
    if start == sequence.length - 1:
        return sequence[-1:]
    minVal = sequence[start]
    minIndex = start
    for i = start to sequence.length:
        if sequence[i] < minVal:
            minVal = sequence[i]
            minIndex = i
    return [minVal] + [findNextItem(sequence, minIndex + 1)]
```

Part 1 Running Time Estimate

$$O(n^2)$$

Part 1 Counterexample

Sequence: [5, 6, 7, 8, 9, 0, 1, 2, 3]
Greedy Algorithm Output: [0, 1, 2, 3]
Optimal Solution: [5, 6, 7, 8, 9]

Part 2 Pseudocode

```
def longestIncreasingSubsequence(sequence, start):
    let num_loops = 0
    let subsequences = []
    for l = 0 to sequence.length:
        num_loops += 1
        let i = l
        subsequence = []
        subsequence.append(sequence[l])
        while i < sequence.length:
            num_loops += 1
            if sequence[i] < sequence[j]:
                subsequence.append(sequence[j])
                i = j
                break
            if j == sequence.length - 1:
                i = sequence.length
                break
        subsequences.append(subsequence)
    sort = sorted(subsequences by length of subsequences from max to
        min)
    return sort[0]
```

Part 2 Running Time Estimate

$$O(n^2)$$

Part 2 Counterexample

Sequence: [1, 8, 3, 6, 5, 4, 7, 2, 9]
Greedy Algorithm Output: [3, 6, 7, 9]
Optimal Solution: [1, 3, 5, 7, 9]

Problem 4 (Our solution is wrong, please read the addendum!)

Algorithm Description

For this problem, we implemented a dynamic programming solution that computed the solution by building it one interval at a time. It computes the maximum number of possible non-overlapping interval subsets at each step and subtracts the number of overlaps the new interval has with the current solution.

Given a single interval i_1, the number of non-overlapping interval subsets `S[1]` is two, consisting of the sets $\emptyset, \{i_1\}$. If we add a second interval i_2 that does not overlap with i_1, then the number of solutions `S[2]` is 4, or `2S[1]`, consisting of the interval subsets $\emptyset, \{i_1\}, \{i_2\}, \{i_1, i_2\}$. If interval i_2 does overlap with i_1, then we subtract the number of intervals it overlaps with, making the solution `2S[1]` $- 1$, which consists of the intervals $\emptyset, \{i_1\}, \{i_2\}$. The solution `S[n]` with n intervals $i_1, i_2, \ldots, i_n$ can be generalized inductively as:

$$\texttt{S[n]} = \texttt{2S[n-1]} - i_n\text{'s overlaps with previous solution}$$

where:

$$\texttt{S[0]} = 1$$

Running Time Estimate

This algorithm runs a single pass through all the intervals, and on each interval it must calculate the number of overlaps with all the intervals in the current solution. This makes it $O(n^2)$.

Psuedocode

```
def countIntervals(intervals):
  let solution = 1
  for i = 1 to intervals.length:
    solution *= 2
    for j = 1 to i:
      if interval[i] overlaps interval[j]:
        solution -= 1
  return solution
```

Addendum

Upon further testing, we realize this solution was wrong. We tested a lot more solutions but we were unable to come up with one that worked and satisfied the time constraint of $O(n^2)$.

It might be possible to fix the solution that we have, if we can somehow calculate the following pieces of information:

1. How many times each interval appears in a set
2. How many times each interval appears in a set with every other interval

With that information, we think it might be possible to calculate what amount should be added to the total each time through the loop, but we aren't sure, and we couldn't figure out a way to calculate that information that didn't balloon our time complexity into the exponential realm.