Analysis of Algorithms

Alvin Lin (axl1439) and William Leuschner (wel2138)

August 2017 - December 2017

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.

Argument of Correctness

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]`:

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

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

```
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 $O(n^2)$.

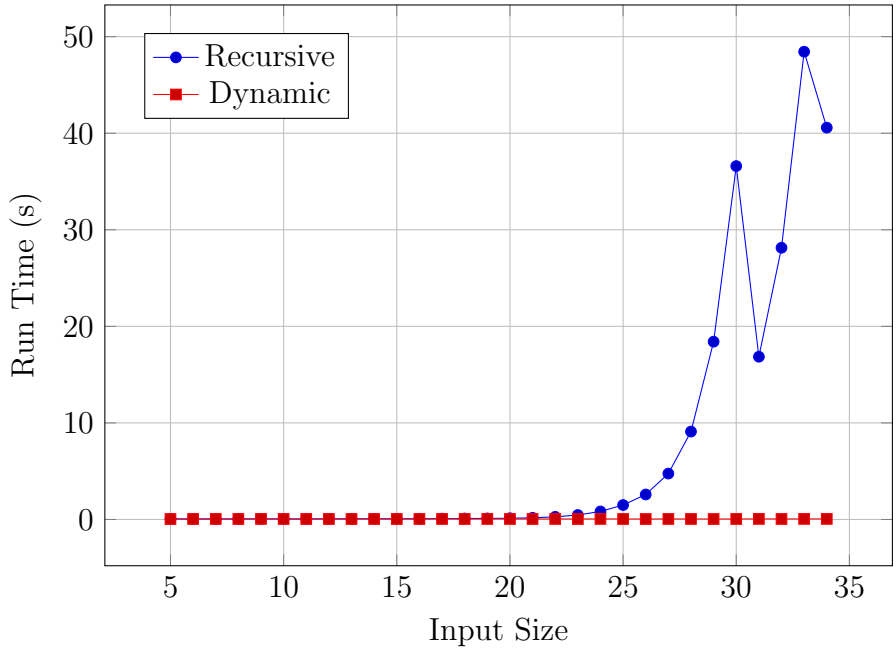
Pseudocode

```
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.

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

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.

Argument of Correctness

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:

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

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
```