Algorithms Homework 7: Andrew Osborne

As noted in the text, the activity problem exhibits an optimal structure which facilitates a dynamic programming approach, and my algorithm uses memoization to achieve a polynomial runtime in the modified activity selection problem.

Data Structures

I use an object-oriented structure to store activities rather than some combination of arrays which are used in parallel. Each activity, it's start time, it's ID, it's finish time, and it's value are stored in a single object. The primary reason for this is to make sorting more convenient; more on that later. Yet another class is used to store an array of activities in addition to the interval information and total number of activities which are provided in the input file. I also add two "ficticious" activities to facilitate the solution of our problem. One of these activities ends at time 0 and the other begins at the end of the specified time interval. By doing this, we can restrict ourselves to the problem of finding the optimal schedule of activities between two activities with no loss of generality. If we did not have lower and upper bounds for our time interval, we would be unable to use this without additional preprocessing.

Algorithm

For the remainder of this section, I will use f_i to denote the finish time of the activity with ID i, and I will use s_i to denote the start time of the activity with ID i. In my analysis, I will assume that ID's are assigned in order of nondecreasing finish time, but in my code, I remap ID's to thier i-th order statistic in order of increasing finish time. That is, I will here pretend that, at the time the algorithm begins, the activities have already been sorted in order of increasing finish time. In my code, I use insertion sort ($\Theta(n^2)$ worst–case runtime) to sort activities, but the object oriented structure above preserves activity IDs so that reporting activity values in the way that they were provided is trivial. Furthermore, I will refer to the activity with ID i as a_i. I will also use c_{ij} to refer to the value of an optimal schedule which takes place between f_i and s_j.

We may think of the un–modified activity selection problem as a special case of our activity problem wherein the value of each activity is 1, or any constant value for that matter. Furthermore, we are concerned with finding the optimal value of a schedule between any two activities, and we want to be able to reconstruct the schedule corresponding to any such value. I present the following pseudocode for my algorithm.

```
// assuming each activity has .id, .start, .end, .value properties
read(file)
  let A[0..n] be a new array of activities

  let A.n = file.number_of_activities + 2
  let A.u = file.max_time

  let A[0].finish = 0
  let A[n].start = u

  for i = 1 to n-1:
    let A[i] = file.ith_activity
  return A

solve(A)
  let c[0..A.n,0..A.n] be a new array of values
  let p[0..A.n,0..A.n] be a new array of integers
  let flags[0..A.n] be a new array of booleans

  for i = 2 to A.n-1 // O(n)
    for j = 0 to A.n - i // O(n - i)
      k = j + i
      l = k - 1
```

```
    while A[j].finish < A[l].finish // will terminate at or before l = j
      // O(k - j) = O(i) worst case
      newValue = A[l].value + c[j,l] + c[l,k]
      if A[j].finish <= A[l].start and A[l].finish <= A[k].start
        if newValue > c[j,k] and
          c[j,k] = newValue
          p[j,k] = l
          // if we just beat the optimal solution, there are no
          // other solutions which have been found which are optimal
          flags[everywhere] = false
        else if newValue == c[j,k] and j == 0 and k == n
          // if we are on the last iteration and we have a duplicate
          // optimal value, we may have found a nonunique solution
          flags[l] = true
      l = l-1
return flags, c, p
```

We are above checking for the optimal value of the schedule of events in order of increasing width. That, is, we check between a_0 and a_2 (noting that only a_1 may fill this space becasue of the sorting of activities), and then we chech in between a_1 and a_3 and so on. We continue in this way, checking all schedules of width i each of which consist of at most $i-1$ checks and of which there are $n-i$ so that, our total runtime, summing over i, is:

$$\sum_{i=2}^{n} i\,n - i^2 = O(n^3)$$

in the worst case. Then, after our main function runs, our optimal value will be stored in `c[0,n]` and our optimal schedule can be ascertained by calling

```
schedule(p,i,j)
  print p[i,j]
  schedule(p,i,p[i,j])
  schedule(p,p[i,j])

schedule(p,0,n)
```

And, from our earlier function, if any of the elements of `flags[i]` is true and `i` is not in the schedule reported by `schedule(p,0,n)` then our solution is not unique. Otherwise, if the only values of `flags` which are true are elements of the reported schedule, then our solution is unique. Checking this can be done in $O(n)$ time. In my code, things are a bit more complicated because I wanted to recover all optimal schedules, which can also be done in $O(n)$ time, but which requires some more involved code.

Testing

To test this code, I shuffled the provided input files to verify that my code gives an answer which agrees with my inspection, and I then wrote a python script which generates random input files in which ID values and finish times are shuffled randomly. I shuffled lists of length 100000 1000 times and verified that my answers agreed on all trials. My code is correct and complete.