CSE 101 Homework 5

Winter 2015

This homework is due Friday March 6th at the start of class. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in \LaTeX is recommended though not required.

**Question 1** (Counting Integer Partitions, 30 points). Define a partition of an integer \( n \) to be a sequence of integers \( 0 < a_1 \leq a_2 \leq a_3 \leq \ldots \leq a_k \) so that \( \sum_{i=1}^{k} a_k = n \). So, for example, the partitions of 5 are 1 + 1 + 1 + 1 + 1, 1 + 1 + 1 + 2, 1 + 2 + 2, 1 + 1 + 3, 2 + 3, 1 + 4, 5. Come up with an algorithm that given \( n \) computes the number of partitions of \( n \) in time that is polynomial in \( n \). [10 points for algorithm, 10 points for runtime analysis, 10 points for correctness proof]

**Solution 1.** This problem can be handled in polynomial time using dynamic programming. We first define a broader problem. Let \( P(n, m) \) be the number of partitions summing to \( n \) using no values greater than \( m \). Our original problem becomes \( P(n, n) \). Furthermore, \( P(n, m) \) can be defined recursively:

- **Base Case:** \( P(0, m) = 1 \), since the only way to sum to 0 is to use no integers.

- **Recursive case:**
  \[
  P(n, m) = \sum_{i=1}^{\min\{n, m\}} P(n-i, i)
  \]

**Algorithm:**

```
procedure calcNumPartitions(n, max_val, numPartitions)
  // n = the number to find partitions for
  // max_val = the maximum number allowed to appear in any partition
  // numPartitions = a memoization table for calcNumPartitions

  if n == 0:
    return 1 // only one way; use no values

  // use memoization to avoid repeated work
  if numPartitions[n][max_val] != NULL:
    return numPartitions[n][max_val]

  sum = 0
  for i = 1 to min(n,max_val):
    // try including i as maximum value first
    // then, see how many you can make for what’s left over
    sum += calcNumPartitions(n-i, i, numPartitions)
    numPartitions[n][i] = sum

  // fill in any entries above max_val we may have missed
  for i = max_val to n:
    numPartitions[n][i] = sum

  return numPartitions[n][i]
```


procedure totalPartitions(n):

    // initialize memoization table
    numPartitions = nxn table of integers
    for i = 1 to n:
        for j = 1 to n:
            numPartitions[i][j] = NULL

    return calcNumPartitions(n, n, numPartitions)

Justification: For a given n, any unique partition can be ordered in order of decreasing element value. Under this ordering, we can be sure that all the partitions starting with a value i are distinct from all of those starting with a different highest value. The remaining elements in any (descending-ordered) partition starting with value i must sum to n – i and must use no value greater than i to produce their sum. Hence, the recursive definition.

Runtime Analysis There are \(O(n^2)\) table entries which must be computed in order to compute \(P(n, n)\). Table entry \((i, j)\) requires \(\min\{i, j\} = O(n)\) previously-computed terms to be summed, so overall, this algorithm has complexity \(O(n^3)\).

Question 2 (Random Walks in Graphs, 30 points). Let \(G\) be a graph with a particular vertex \(s\). Consider the discrete time random walk on \(G\) starting at \(s\). Namely, a person starts at \(s\) at time \(t = 0\). Then at each timestep, the person moves to a random adjacent vertex. So if they are at vertex \(v\) at time \(t\), then at time \(t + 1\), they will move to a random adjacent vertex (each with equal probability).

Find an algorithm that given a graph \(G\), a vertex \(s\) and an integer \(t\) and another vertex \(v\) computes the probability that the random walk starting at \(s\) will be at vertex \(v\) at time \(t\). You algorithm should have runtime polynomial in \(t\) and the number of edges and vertices of \(G\). [10 points for algorithm, 10 points for runtime analysis, 10 points for correctness proof]

Solution 2. This problem can be solved using dynamic programming. Let us define \(P(s, v, t)\) as the probability that if we start a random walk from \(s\) (in graph \(G\)) and take \(t\) steps, we will end up at vertex \(v\). We can define this value recursively, as follows:

    • Base Case:
      
      \[ P(s, v, 0) = \begin{cases} 1 & s = v \\ 0 & \text{otherwise} \end{cases} \]

    • Recursive Case:
      
      \[ P(s, v, t) = \sum_{e=(x,v)} P(s, x, t - 1) \cdot \frac{1}{\text{neighbors}(x)} \]

Algorithm:

procedure P(G, s, v, t)

    // G = a graph
    // s = a starting node
    // v = a destination node
    // t = number of time steps

    // Base case
    if t == 0:
        if s == v:

return 1
return 0

// if already memoized
if v.probability[t] != NULL:
    return v.probability[t]

sum = 0
for x in v.neighbors:
    // assume each vertex knows how many
    // nodes it can reach
    sum = sum + P(G, s, x, t-1)/x.neighbor_count

v.probability[t] = sum
return sum

procedure calcProbability(G, s, v, t)
// G = a graph
// need to reverse edges so that each vertex
// knows which vertices it can reach, rather
// than knowing which vertices it can reach;
// only necessary for directed graph
G_R = reverseEdges(G)
// initialize vertex probability arrays
for v in G_R.vertices:
    for i = 0 to t:
        v.probabilities[i] = NULL

return P(G_R, s, v, t)

Justification: The base case follows from the observation that, given no steps, we can only end up where we started, so we end up at s with probability 1 and anywhere else with probability 0.

The recursive case follows from the observation that, for each neighbor x of v, there is some probability that you end up on x after t-1 steps. The probability that you end up on v on the final step is the sum over all of v’s neighbors of the probability that, on the final step, you go from x to v, multiplied by the probability that you were on x at time t-1. Since, at any vertex, a random walk moves to its neighbors with equal probability, the probability that v in particular is visited from x is \( \frac{1}{\text{neighbors}(x)} \).

Runtime Analysis: Assume addition and multiplication are constant-time operations, and the number of neighbors of x can be determined in constant time. Each vertex can maintain an array of probabilities that a random walk with i steps ends up on it, for all time values 0 ≤ i ≤ t − 1. This requires a total of O(|V| · t) entries. In the worst case that the graph is complete, the computation of each entry requires |V| multiplications to be computed summed, so the worst case runtime is O(|V|^2 · t).

Question 3 (Tour Scheduling, 40 points). Hal is planning a concert tour for his band. Several possible venues have given him offers to hold shows. The ith venue is willing to pay Ri for his band but would keep them occupied for an interval of time Ii. Hal knows that he cannot book his band at two venues for overlapping time intervals.

(a) Give an algorithm to find a schedule of venues to perform at that makes as much total money as possible without booking two venues for overlapping times. [10 points for algorithm, 10 points for runtime
(b) Hal realizes that there is also a cost of travelling between venues. Suppose that for each pair of venues $i$ and $j$ there is a cost $C(i,j)$ to travel from one location to the other. How would you need to modify the above algorithm to minimize the total revenue minus travel costs? [10 points for new algorithm]

The algorithms in each of these parts should be polynomial time in the number of venues.

Solution 3. This problem is similar to Interval Scheduling, where the $I_i$ are the intervals. However, since the intervals are now weighted by $R_i$, a greedy algorithm may not compute an optimal solution. The new problem can be solved with a dynamic programming algorithm.

(a) As in the greedy algorithm, suppose each interval $I_i$ is given by the start and finish times $s_i, f_i$, respectively. Sort the intervals by finish time so that $f_1 \leq f_2 \leq \cdots \leq f_n$ if $n$ is the number of venues offering shows. Say that two intervals $I_i$ and $I_j$ are compatible if they are non-overlapping. Finally, define $p(i)$ to be the largest $j < i$ such that $I_i$ and $I_j$ are compatible, or $0$ if there is none.

We will keep track of two things: the optimal schedule of venues and the payout of the tour. Define the following subproblems:

$$TS(i) = \text{the optimal tour schedule whose last appearance is } i \text{ (or the empty tour if } i = 0)$$
$$PO(i) = \text{payout of optimal tour schedule } TS(i)$$

- **Base Case:** $PO(0) = 0, TS(0) = \emptyset$.
- **Recursive Case:**

$$PO(i) = \max_{j \leq p(i)} (PO(j) + R_i)$$
$$TS(i) = TS(j) \circ i \text{ if } PO(j) + R_i \text{ attains max}$$

- **Final Solution:** $TS(j)$, where $PO(j)$ is maximal.

Note that the $PO$ values will be dollar amounts, and the $TS$ values will be strings of venues. For example if $TS(5) = 1, 3, 5$, this means the optimal tour schedule using the venues $1-5$ is venue 1, then 3, then 5 scheduled for times $I_1, I_3, I_5$. To this end, the $\circ$ denotes concatenation.

**Algorithm:** For the algorithm statement, assume all the $I_i$ are given by $I$, and the $R_i$ are given by $R$.

```plaintext
procedure tourSchedule(I, R):
    // I is the list of intervals
    // R is the list of payments

    // preprocess
    sort the venues by finish time

    for every venue i:
        compute $p(i)$ and store in a lookup table

    // initialize payout array and tour schedule linked list
    PO = array of size $n + 1$
    TS = linked list of size $n + 1$

    // base case
    PO[0] = 0
    TS[0] = null

    // compute optimal tour schedules
```
for i = 1 to n:
    max = 0
    for j = 1 to p(i):
        if PO[j] > PO[max]:
            max = j
    PO[i] = PO[max] + R(i)
    TS[i] = TS[max].add(i)
max = 0
for j = 1 to n:
    if PO[j] > PO[max]:
        max = j
return TS[max]

Justification: We prove correctness by induction. First, if the tour is scheduled with no venues, the tour schedule must be empty and the payout 0, so the base case holds. Next, assume we have scheduled the optimal tour and computed the payout from among the first i venues. Suppose that we have a tour whose last appearance is at the i+1st venue. Then the previous visit is to the jth venue for some j ≤ p(i). The best we can do is to pick the j so that a tour ending with the jth venue does as well as possible.

Runtime Analysis: There are total of n subproblems, each of which involves a maximization, for a total time of O(n^2). Sorting the intervals takes O(n log n) time and computing the p(i) takes O(n^2). The total running time including preprocessing is O(n^2).

(b) For this problem, we need to keep track not only of the best tour using the first i we modify the calculation of tour payout to account for the cost of travel. The subproblems will be defined as in part (a), and we modify the recursive formulation to include the cost from the second to last stop on the tour to the last. Also, define C(∅, i) = 0 for all i. Then,

• **Base Case:** PO(0) = 0, TS(0) = ∅.

• **Recursive Case:**

\[
PO(i) = \max_{j \leq p(i)} (PO(j) + R_i - C(j, i))
\]

\[
TS(i) = TS(j) \circ i \text{ if } PO(j) + R_i \text{ attains max}
\]

Final Solution: TS(j), where PO(j) is maximal.

The algorithm is otherwise unchanged, and the running time is the same as part (a).