CS 561, HW4

Prof. Jared Saia, University of New Mexico

Due: October 17th

1. Consider the following alternative greedy algorithms for the activity selection problem discussed in class. For each algorithm, either prove or disprove that it constructs an optimal schedule.

   (a) Choose an activity with shortest duration, discard all conflicting activities and recurse

   (b) Choose an activity that starts first, discard all conflicting activities and recurse

   (c) Choose an activity that ends latest, discard all conflicting activities and recurse

   (d) Choose an activity that conflicts with the fewest other activities, discard all conflicting activities and recurse

2. Now consider a weighted version of the activity selection problem. Imagine that each activity, a_i has a *weight*, $w(a_i)$ (weights are totally unrelated to activity duration). Your goal is now to choose a set of non-conflicting activities that give you the largest possible sum of weights, given an array of start times, end times, and values as input.

   (a) Prove that the greedy algorithm described in class - Choose the activity that ends first and recurse - does not always return an optimal schedule for this problem

   (b) Describe an algorithm to compute the optimal schedule in $O(n^2)$ time. Hint: 1) Sort the activities by finish times. 2) Let $m(j)$ be the maximum weight achievable from activities $a_1, a_2, \ldots, a_j$. 3) Come up with a recursive formulation for $m(j)$ and use dynamic programming. Hint 2: In the recursion in step 3, it'll help if you precompute for each job j, the value x_j which is the largest index i less than j such that job i is compatible with job j. Then when

computing $m(j)$, consider that the optimal schedule could either include job j or not include job j.

3. Consider the following problem.
   INPUT: Positive integers $r_1, \ldots, r_n$ and $c_1, \ldots, c_n$.
   OUTPUT: An n by n matrix A with 0/1 entries such that for all i the sum of the ith row in A is r_i and the sum of the ith column in A is c_i, if such a matrix exists.
   Think of the problem this way. You want to put pawns on an n by n chessboard so that the ith row has r_i pawns and the ith column has c_i pawns. Consider the following greedy algorithm that constructs A row by row. Assume that the first $i-1$ rows have been constructed. Let a_j be the number of 1s in the jth column in the first $i-1$ rows. Now the r_i columns with maximum $c_j - a_j$ are assigned 1s in row i, and the rest of the columns are assigned 0's. That is, the columns that still needs the most 1s are given 1s. Formally prove that this algorithm is correct using an exchange argument.

4. Suppose we can insert or delete an element into a hash table in O(1) time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
   - After a deletion, if the table is less than 1/4 full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

   Show that for any sequence of insertions and deletions, the amortized time per operation is still O(1). Hint: Do not use potential functions.

5. Suppose we are maintaining a data structure under a series of operations. Let $f(n)$ denote the actual running time of the nth operation. For each of the following functions f, determine the resulting amortized cost of a single operation.

   - $f(n) = n$ if n is a power of 2, and $f(n) = 1$ otherwise.
   - $f(n) = n^2$ if n is a power of 2, and $f(n) = 1$ otherwise.

6. Describe and analyze a data structure to support the following operations on an array $A[1 \ldots n]$ as quickly as possible. Initially, $A[i] = 0$ for all i.

   - **SetToOne(i)** Given an index i such that $A[i] = 0$, set $A[i]$ to 1.
   - **GetValue(i)** Given an index i, return $A[i]$
   - **GetClosestRightZero(i)** Given an index i, return the smallest index $j \geq i$ such that $A[j] = 0$, or report that no such index exists.

   The first two operations should run in worst-case constant time, and the amortized cost of the third operation should be as small as possible.