Department of Electronics Engineering
National Chiao Tung University
Algorithms, Fall 2017

December 7, 2017
Iris Hui-Ru Jiang

Sample Solutions to Homework #3

1. (25) The following shows a linear time algorithm for card majority checking problem. We pair up all cards randomly and test all pairs for equivalence. If n is odd, one card is unmatched. For each pair that is not equivalent, discard both cards. For pairs that are equivalent, keep one of the two. If n is odd, keep the unmatched card. This subroutine is named $ELIMINATE$.

   The observation that leads to the linear time algorithm is as follows. If there is a majority calss with more than $n/2$ cards, the same equivalence class must also have more than half of the cards after $ELIMINATE$. This is true, since when we discard both cards in a pair, at most one of them can be from the majority class. One $ELIMINATE$ call on a set of n cards takes $n/2$ test, and have at most $\lceil n/2 \rceil$ cards left. When we are down to a single cards, then its equivalence is the only candidate for having majority. We test this cards against all others to check if its equivalence class has more than $n/2$ elements.

   The algorithm takes $n/2 + n/4 + n/8 + ...$ tests for all the eliminations, plus $n - 1$ tests for the final counting, for a total of less than $2n = O(n)$ tests.

2. (25) The greedy algorithm for fractional Knapsack problem is shown as follows: we sort all the items by their unit value per weight $\frac{v_i}{w_i}$ in non-increasing order. And we take the items according to this order until the knapsack is full. Note that only the last-taken item might be fractional.

   The following proves the optimality of the greedy choice property. Suppose there exist an optimal solution S with one item violating the non-increasing order. It is obvious that there must be at least one item that this solution "skip" selecting; let the skipped item be the item i with weight w_i, and the last item be j with fractional weight w'_j, as shown in Figure 1. Consider a solution S' obtained by substitute the item j by the item i with the same weight. Knowing that $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$, which means the solution S' is at least as good as S', the optimality of the greedy choice property holds. The running time of this algorithm is equal to the sorting scheme, which is $O(n \lg n)$.

$$order: \left\langle \frac{v_1}{w_1}, \frac{v_2}{w_2}, \dots, \frac{v_i}{w_i}, \dots, \frac{v_j}{w_j}, \dots, \frac{v_n}{w_n} \right\rangle$$
$$S: \left\langle \frac{v_1}{w_1}, \frac{v_2}{w_2}, \dots, \cancel{\frac{v_i}{w_i}}, \dots, \frac{v_{j-1}}{w_{j-1}}, \frac{v_j}{w_j} \right\rangle$$
$$S': \left\langle \frac{v_1}{w_1}, \frac{v_2}{w_2}, \dots, \frac{v_i}{w_i}, \dots, \frac{v_{j-1}}{w_{j-1}} \right\rangle$$

Figure 1: Solutions for problem 2.

3. (25) The following shows a linear time algorithm for minimizing the total hotel cost by dynamic programming. Let $C(i)$ be the minimum hotel cost needed to location i, and c_i

be the cost for hotel located at i kilometers. The recursive formula is given as follows:

$$C(i) = \begin{cases} c_1, i = 1 \\ (\min_{k=i-20}^{i-1}\{C(k)\}) + c_i, \forall i = 2 \sim n \\ \infty, \forall i \neq 1 \sim n \end{cases} \tag{1}$$

We need only traverse i from 1 to n one time, with each $C(i)$ takes at most 20 checking in the lookup table. The overall running time is at most $20n = O(n)$.

4. (25) The following shows a linear time algorithm for minimizing schedule cost by dynamic programming. Let $OPT(i)$ denote the minimum cost of a solution for weeks 1 through i. In an optimal solution, we either use company A or company B for the i^{th} week. If we use company A, we pay $rs_i$and can behave optimally up through week $i-1$. If we use company B for week i, then we pay $4c$ for this contract, and so there is no reason not to get the full benefit of it by starting it at week $i-3$; thus we can behave optimally up through week $i-4$, and then invoke this contract. The recursive formula is given as follows:

$$OPT(i) = \min(rs_i + OPT(i-1), 4c + OPT(i-4)) \tag{2}$$

We can build up these OPT values in order of increasing i, spending constant time per iteration, with the initialization $OPT(i) = 0$ for $i \leq 0$. The desired value is $OPT(n)$, and we can obtain the schedule by tracing back through the array of OPT values.