CS 670 Spring 2015 - Solutions to HW 3

Problem 16.2-1

Let $\{v_i\}$ be the set of values and $\{w_i\}$ the corresponding set of weights. Sort the items in decreasing order of value-by-weight ratio. After re-ordering, for all i we have,

$$\frac{v_i}{w_i} \geq \frac{v_{i+1}}{w_{i+1}}$$

The greedy algorithms proceeds by filling the knapsack with as much of the first item as possible. Then the second item and so on until the knapsack is full. In particular we pick w_1 of item 1, w_2 of item 2,..., w_{j-1} of item $j-1$, a_j of item j such that $\sum_{i<j} w_i + a_j = W$. Here $a_j \leq w_j$ and W is the capacity of the knapsack.

Assume that the greedy solution is not optimal. Let OPT be an optimal solution and v_{opt} the optimal value. Clearly, OPT is different the greedy solution and thus contains some amount (say a_k) of item k for some $k > j$. We can replace this with a combination of fractions of items $b_i, i \leq j$ of total weight $\sum_{i \leq j} b_i = a_i$. This would increase the the value of the optimal solution contradicting its optimality.

Thus our greedy solution is indeed optimal.

Problem 16.2-2

Let $OPT(s, w)$ be the optimum value for item set s, if total weight is at most w. Observe that for each item i in set s, the optimum solution will either include it or not. So we have the recurrence:

$$OPT(s, w) = \max(OPT(s \setminus \{i\}, w), OPT(s \setminus \{i\}, w - w_i) + v_i), (w_i \leq w) \tag{1}$$

$$OPT(s, w) = OPT(s \setminus \{i\}, w), (w_i > w) \tag{2}$$

The base cases are $OPT(\phi, w) = 0$.

The correctness is guaranteed by the fact that the optimum solution either includes item i (if legal), and has w_i less weight available for the rest, or it doesn't include item i.
It cost $O(1)$ to compute the opt value in each recurrence step, totally we need to compute nW values of OPT(s,w), so the total running time is $O(nW)$.

Problem 16.2-7

Reorder the elements of A and B iteratively as follows.

Set $(a_1, b_1) = argmax_{a \in A, b \in B}(a^b)$. That is find (a, b) such that a^b is maximum and assign it to (a_1, b_1). Now set $(a_2, b_2) = argmax_{a \in A \setminus \{a_1\}, b \in B \setminus \{b_1\}} a^b$. That is remove $(a1, b1)$ and find (a, b) such that a^b is maximum among the remaining elements. Assign it to (a_2, b_2). Do this inductively to get an ordering.

We claim that this ordering is optimal.
We proceed by induction. The claim is clearly true when $n = 1$.
Induction Hypothesis: Assume that the claim is true for $n = k - 1$. That is, if $\{a'_1, a'_2, \dots, a'_n\}$ and $\{b'_1, b'_2, \dots, b'_n\}$ is any ordering of elements of A and B, then

$$\prod_{i=1}^{k-1} {a_i}^{b_i} \geq \prod_{i=1}^{k-1} {a'_i}^{b'_i}$$

Claim:

$$\prod_{i=1}^{k} {a_i}^{b_i} \geq \prod_{i=1}^{k} {a'_i}^{b'_i}$$

Proof: Let $A' := \{a'_1, a'_2, \dots, a'_k\}$ and $B' := \{b'_1, b'_2, \dots, b'_k\}$. We can permute the indices using a permutation π so that

$${a'}_{\pi(1)}^{b'_{\pi(1)}} \geq {a'}_{\pi(2)}^{b'_{\pi(2)}} \geq \dots \geq {a'}_{\pi(k)}^{b'_{\pi(k)}}$$

Consider,

$$\prod_{i=1}^{k} {a'_i}^{b'_i} = \prod_{i=1}^{k} {a'_{\pi(i)}}^{b'_{\pi(i)}} = {a'_{\pi(k)}}^{b'_{\pi(k)}} \prod_{i=1}^{k-1} {a'_{\pi(i)}}^{b'_{\pi(i)}}$$

$$\leq {a'_{\pi(k)}}^{b'_{\pi(k)}} \prod_{i=1}^{k-1} {a_i}^{b_i} \leq \prod_{i=1}^{k} {a_i}^{b_i}$$

The first inequality is a consequence of the induction hypothesis. The second inequality is a consequence of the ordering, ie ${a_k}^{b_k} \geq {a'_{\pi(k)}}^{b'_{\pi(k)}}$.

(Note: The ordering provided in the algorithm is equivalent to sorting A and B in descending order.)

Problem 16-1

(a) The algorithm keeps choosing the coin with greatest denomination less than or equal to the amount left.

Observation. Let k be the number of coins of denomination d_i just enough to be greater than equal to d_{i+1}, where d_{i+1} is the next higher denomination. Then k coins of denomination d_i can be replaced by d_{i+1} and some other coin(s) using less than k coins. One can verify this observation by taking into account the fact that 5 pennies can be replaced by a nickel, 2 nickels can be replaced by a dime and 3 dimes can be replaced by a quarter and a nickel (2 coins).

Let the money to change for be n, $n = 25a + 10b + c + d$. a, b, c, d are the numbers of quarters, dimes, nickels and pennies. We have $b < 3$, $c < 2$ and $d < 5$ (or it will violate the conclusion in the observation, e.g., if $b \geq 3$, we can replace 3 dimes by a quarter and a nickel which results in less coins). We prove that when $n \geq 25$, we should always pick a quarter. As $10b + c + d \leq 10 * 2 + 5 * 1 + 1 * 4 = 29$, it means in the optimal solution, the amount of the money of the dimes, nickels and pennies will be no more than 29 cents. So we only need to prove the cases when n is 25, 26, 27, 28 or 29. One can easily verify that in such cases it is always better to choose a quarter. Similarly we can prove that when $10 \leq n < 25$, we should always pick a dime and when $5 \leq n < 10$ we should always pick a nickel.

(b) Prove it by induction on the number of different kinds of coins. When we have only 1 kind of coins whose denomination is c^0, obviously it is correct that we always choose the coin with greatest denomination. Assume that it holds for m kinds of coins whose denominations are $c^0, c^1, \ldots, c^m$, that is, when we need to change money using these m kinds of coins, the optimal solution will always pick the coin with greatest denomination less than or equal to the amount left. We prove it also holds for $m+1$ kinds of coins whose denominations are $c^0, c^1, \ldots, c^{m+1}$. Suppose the money we need to make change for is n. According to the assumption, $n = a_0c^0 + a_1c^1 + \cdots + a_mc^m$. a_i is the number of the i^{th} coin we choose, we have $a_i < c$ when $0 \le i < m$. If $a_mc^m < c^{m+1}$, then $a_m < c$, we have $n < c^{m+1}$ because $n < (1+a^m)c^m \le c * c^m$ (the money left after picking the c^m coins should be less than c^m, that is, $a_0c^0 + a_1c^1 + \cdots + a_{m-1}c^{m-1} < c^m$) and $c > 1$ (all numbers are integer). In such cases, the money is less than the denomination of the $(m+1)^{th}$ coin, the conclusion is correct as this falls into the same condition as using m coins. If $a_mc^m \ge c^{m+1}$, then $a_m \ge c$. Let $a_m = pc+q$ (p, q are integers and $p \ge 1$, $0 \le q < c$). So $n = a_0c^0 + a_1c^1 + \cdots + qc^m + pc^{m+1}$, we prove that $S = \{a_0, a_1, \ldots, a_{m-1}, q, p\}$ is the optimal selection of the numbers of the $m+1$ coins, and it is exactly the result of our greedy algorithm, which uses the $m+1$ coins to change for money n. Prove it by contradiction. Assume the optimal selection is $S' = \{a'_0, a'_1, \ldots, a'_{m-1}, a'_m, a'_{m+1}\}$, that is $\sum_{i=0}^{m+1} a'_i < \sum_{i=0}^{m-1} a_i + q + p$. We have $0 \le a'_{m+1} \le p$. Consider changing for money $n - a'_{m+1}c^{m+1}$, the selection made by S' will be $\{a'_0, a'_1, \ldots, a'_{m-1}, a'_m, 0\}$, observe that this is actually the same situation as using the first m coins, so according to our assumption, we have $a'_i = a_i$ when $0 \le i \le m-1$. So $S' = \{a_0, a_1, \ldots, a_{m-1}, a'_m, a'_{m+1}\}$. So what we need to prove is $a'_m + a'_{m+1} \ge q+p$ when $a'_mc^m + a'_{m+1}c^{m+1} = qc^m + pc^{m+1}$ and $a'_{m+1} \le p$. We have $a'_m + a'_{m+1}c = q + pc$, so $a'_m + a'_{m+1} = q + pc - ca'_{m+1} + a'_{m+1} = q + pc - (c-1)a'_{m+1} \ge q + pc - (c-1)p = q + p$.

(c) Consider the set of denominations 7, 6, 1. When the money is 12 cents. The greedy algorithm will suggest using 1 7-cent coin and 5 1-cent coin, it would be 6 coins in total. While the optimum solution is using 2 6-cent coins.

(d) Use dynamic programming. Let $OPT(i)$ be the optimal number of coins needed to make change for i cents. Let $S = \{d_1, d_2, \ldots, d_k\}$ be the set of denominations of coins. We have the recurrence

$$OPT(i) = \min_{j=1}^{k}\{OPT(n - d_j) + 1\} \tag{3}$$

The base cases are $OPT(d_j) = 1$ for $j = 1 \ldots k$, and when $i < 0$, $OPT(i) = \infty$.

Every step of the recurrence needs to compare k values, totally we need to compute n $OPT(i)$ values, so the total running time is $O(nk)$.

Problem 16-2

(a) Sort the tasks a_i in increasing order based on the their processing time p_i. To process the tasks in this sorted order is the optimal solution which minimizes the average completion time.

Proof of correctness : Let c_i denote the completion time of task a_i. So $c_1 = p_1, c_2 = p_1 + p_2, \ldots$. So $\sum_{i=1}^{n} c_i = np_1 + (n-1)p_2 + \ldots + p_n$ and we want to minimize $(1/n)\sum_{i=1}^{n} c_i$.

It follows that sorting the tasks would give us the optimal solution because otherwise assuming that in the optimal solution $(a^*_1, \ldots, a^*_n)$, a^*_x is placed before a^*_y and $p^*_x > p^*_y$ we observe that swapping a^*_x with a^*_y results in a solution which is better than the optimal. This contradicts with the assumption.

Sorting the tasks costs $O(nlgn)$, processing the n sorted tasks costs $O(n)$, so the total running time is $O(nlgn)$.

(b) Sort the r_i's in time order. Maintain a priority heap of jobs with the key being the time the job still needs for completion.

At r_1, schedule a_1. If at r_2, a_1 is still running, preempt a_1, insert it into the priority queue. Also, add a_2 to the priority queue. Similarly, at each r_i,

1. Preempt the current task, add it to the priority queue with the its key as the time it still requires for completion.
2. Add a_i to the priority queue with its key as the time it requires for completion.
3. Remove the task with the least key and process it.

Whenever a task completes, we remove the next task from the priority queue and process it.

Time Complexity: Sorting takes $O(nlgn)$. Insertion and removing the minimum element from a heap takes $O(lgn)$ time. These operations are made at most $2n$ times, as there are only n release events and n completion events. Hence, the total time taken by the algorithm is $O(nlgn)$.

According to our algorithm, at every time point we are always running the job with least time left to complete. We prove that this will result in least average completion time. Assume there is an optimal solution which at some time point t, didn't run the job with least time left to complete (let it be a_i), but another job a_j. Let c_i be the completion time of job a_i, and c_j be the completion time of a_j. Let s_i be the starting time of a_i and s_j be the starting time of a_j. We divide the time line as time units, every job occupies an amount of time units which could be scattered on the time line. According to the assumption, job a_i starts later than a_j, and the time units occupied by a_i is less than the time units occupied by a_j. There are two cases to consider: In the first case, job a_i is completed before a_j is completed ($c_i < c_j$). We exchange the time units of a_i with the time units of job a_j from the end to the front (replacing its time units backwards from c_j). After exchanging, we have $c'_i = c_j$ and $c'_j < c_i$, which means the sum of the total running time is less, thus we get a better solution. In the second case, job a_i is completed after a_j or at the same time ($c_i \geq c_j$). We exchange the time units of job a_j starting from s_j with the time units occupied by a_i. So we have $c'_i < c_j$ and $c'_j = c_i$, which means the sum of the total running time is less, thus we also get a better solution. This contradicts with the assumption.