ECE608, Homework #9 Solution

(1) CLR 16.1-3

Let S be the set of n activities. The obvious solution of using GREEDY-ACTIVITY-SELECTOR to find a maximum-size set $S1$ of compatible activities from S for the first lecture hall, then using it again to find a maximum-size set $S2$ of compatible activities from $S - S1$ for the second hall, and so on until all the activities are assigned, requires $\Theta(n^2)$ time in the worst case.

However, there is a better algorithm, whose asymptotic time is just the time needed to sort the activities by time (i.e., $O(n \lg n)$ time), or possibly as fast as $O(n)$ if the times are small integers.

The general idea is to go through the activities in order of start time, assigning each to any hall that is available at that time. To do this, move through the set of events consisting of activities starting and activities finishing, in order of event time. Maintain two lists of lecture halls: Halls that are busy at the current event-time t (because they have been assigned an activity i that started at $s_i \leq t$ but won't finish until $f_i > t$) and halls that are free at time t. (As in the activity-selection problem in Section 16.1, we are assuming that activity time intervals are half open—that is, if $s_i \geq f_j$, activities i and j are compatible.) When t is equal to the start time of some activity, assign that activity to a free hall and move the hall from the free list to the busy list. When t is equal to the finish time of some activity, move the activity's hall from the busy list to the free list. (The activity is certainly in some hall, because the event times are processed in order and the activity must have started before its finish time t, hence must have been assigned to a hall.)

To avoid using more halls than necessary, always pick a hall that has already had an activity assigned to it, if possible, before picking a never-used hall. (This can be done by always working at the front of the free-hall list—putting freed halls onto the front of the list and taking halls from the front of the list—so that a new hall doesn't come to the front and get chosen if there are previously used halls.)

This guarantees that the algorithm uses as few lecture halls as possible: The algorithm will terminate with a schedule requiring $m \leq n$ lecture halls. Let activity i be the first activity scheduled in lecture hall m. The reason that i was put in the m-th lecture hall is that the first $m-1$ lecture halls were busy at time s_i. So at this time, there are m activities occurring simultaneously. Therefore any schedule must use at least m lecture halls, so the schedule returned by the algorithm is optimal.

Run time:

(i) Sort the $2n$ activity-starts/activity-ends events. (In the sorted order, an activity-ending event should precede an activity-starting event at the same time.) $O(n \lg n)$

time for arbitrary times, possibly $O(n)$ if the times are restricted (e.g., to small integers).

(ii) Process the events in $O(n)$ time: Scan the $2n$ events, doing $O(1)$ work for each (moving a hall from one list to the other and possibly associating an activity with it).

Total: $O(n +$ time to sort).

(2) CLR 16.1-4

You are asked for two counterexamples, one for each approach. Show that each is indeed a counterexample. The duration of activity i is $f_i - s_i$. Here is a simple example showing greedy selection by shortest duration is not optimal:

Activity	**Start**	**Finish**	**Duration**
1	0	9	9
2	8	11	3
3	10	18	8

Greedy selection by least duration finds only $\{2\}$, while the optimal selection is $\{1, 3\}$.

Here is a simple example showing greedy selection by fewest overlaps is not optimal:

Activity	**Start**	**Finish**	**Number of overlaps**
1	0	5	3
2	5	10	3
3	10	15	4
4	15	20	4
5	20	25	3
6	25	30	3
7	1	11	5
8	2	12	5
9	3	13	5
10	17	27	5
11	18	28	5
12	19	29	5
13	14	16	2

Greedy selection by fewest overlaps chooses activity 13 first followed by activities 1, 2, 5, and 6, in any order. However, the unique optimal selection is 1,2,3,4,5,6.

(3) CLR 16.2-1

Let $S = \{1, 2, 3, \ldots, n\}$ = the set of items available.

Assume that S is first sorted by weight, and then sorted by value per pound using a stable sort (i.e., $\{\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \ldots \frac{v_n}{w_n}\}$, if $\frac{v_i}{w_i} = \frac{v_{i+1}}{w_{i+1}}$, then $w_i \geq w_{i+1}$).

Let A be the optimal solution to the fractional knapsack problem and assume items in it is sorted in the same way as above. Suppose the total weight is W and the total optimal value is V.

Claim that there is another optimal solution B to S that begins with the greedy choice.

Case 1: Total weight $W \leq w_1$.

Clearly filling up an entire bag with item 1 will give us another optimal solution, since $\frac{v_1}{w_1}$ = the maximum value per pound.

Case 2: Total weight $W > w_1$.

Look at the first w_1 pounds of items in A (Note: A may contain item 1. In this case, the sorting algorithm we used will put item 1 at the beginning of A). Let $B = A - \{\text{first } w_1 \text{ pounds of } A\} + \{w_1 \text{ pounds of item 1}\}$ (i.e., replace first w_1 pounds of A with w_1 pounds of item 1). Then, value of $A = \{\text{Value of the first } w_1 \text{ pounds}\} + \{\text{Value of the rest of the knapsack}\}$. Also, value of $B = \{\text{Value of the first } w_1 \text{ pounds of item 1}\} + \{\text{Value of the rest of the knapsack}\}$. Since $\frac{v_1}{w_1} \geq \frac{v_i}{w_i}$, we have $\{\text{Value of the first } w_1 \text{ pounds}\} \leq v_1$. But since A is the optimal solution, $\{\text{Value of the first } w_1 \text{ pounds}\} = v_1$. Thus, the value of A is the same as the value of B, and both are optimal. Thus, greedy choice of item 1 is an optimal choice.

Since this is established, the problem can be reduced to knapsack-problem with S of $n-1$ items and maximum weight $= W - w_1$. By induction, making the greedy choice at each step produces an optimal solution.

(4) CLR 16.2-2

The solution is based on the optimal-substructure observation in the text: Let i be the highest-numbered item in an optimal solution S for W pounds and items 1 to n. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ pounds and items 1 to $i - 1$, and the value of the solution S is v_i plus the value of the subproblem solution S'.

We can express this in the following formula: Define $c[i, w]$ to be the value of the solution for items 1 to i and maximum weight w. Then

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

This says that the value of a solution for i items either includes item i, in which case it is v_i plus a subproblem solution for $i - 1$ items and the weight excluding w_i, or doesn't include item i, in which case it is a subproblem solution for $i - 1$ items and the same weight. That is, if the thief picks item i, he takes v_i value, and he can

choose from items 1 to $i-1$ up to the weight limit $w-w_i$, and get $c[i-1, w-w_i]$ additional value. On the other hand, if he decides not to take item i, he can choose from items 1 to $i-1$ up to the weight limit w, and get $c[i-1, w]$ value. The better of these two choices should be made.

Although the 0-1 knapsack problem doesn't seem analogous to the longest-common-subsequence problem, the above formula for c is similar to the LCS formula: boundary values are 0, and other values are computed from the inputs and "earlier" values of c. So the 0-1 knapsack algorithm is like the LCS-LENGTH algorithm given in Chapter 16 for finding a longest common subsequence of two sequences.

The algorithm takes as inputs the maximum weight W, the number of items n, and the two sequences $v = \langle v_1, v_2, \ldots, v_n \rangle$ and $w = \langle w_1, w_2, \ldots, w_n \rangle$. It stores the $c[i, j]$ values in a table $c[0..n, 0..W]$ whose entries are computed in row-major order (i.e., the first row of c is filled in from left to right, then the second row, and so on). At the end of the computation, $c[n, W]$ contains the maximum value the thief can take.

```
DYNAMIC-0-1-KNAPSACK(v, w, n, W)
1.  for w ← 0 to W
2.      do c[0, w] ← 0
3.  for i ← 1 to n
4.      do c[i, 0] ← 0
5.          for w ← 1 to W
6.              do if w_i ≤ w
7.                      then if v_i + c[i − 1, w − w_i] > c[i − 1, w]
8.                              then c[i, w] ← v_i + c[i − 1, w − w_i]
9.                              else c[i, w] ← c[i − 1, w]
10.                     else c[i, w] ← c[i − 1, w]
```

The set of items to take can be deduced from the c table by starting at $c[n, W]$ and tracing where the optimal values came from. If $c[i, w] = c[i-1, w]$, item i is not part of the solution, and we continue tracing with $c[i-1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i-1, w-w_i]$.

The above algorithm takes $\Theta(nW)$ total time. It takes $\Theta(nW)$ to fill in the c table—$(n+1)(W+1)$ entries each requiring $\Theta(1)$ time to compute. Also, it takes $O(n)$ time to trace the solution (since it starts in row n of the table and moves up 1 row at each step).

(5) CLR 16.2-5

Note: Closed interval is one that includes the end-points.

Algorithm: Set a *pointer* to negative unfinity. Execute a while loop that conditions on all points not being covered by an interval yet. In each iteration find the lowest value point x_i such that $x_i > pointer$. Create an new interval $[x_i, x_i + 1]$ and add it to the set of intervals. Then set $pointer \leftarrow x_i + 1$ and return to the beginning of the

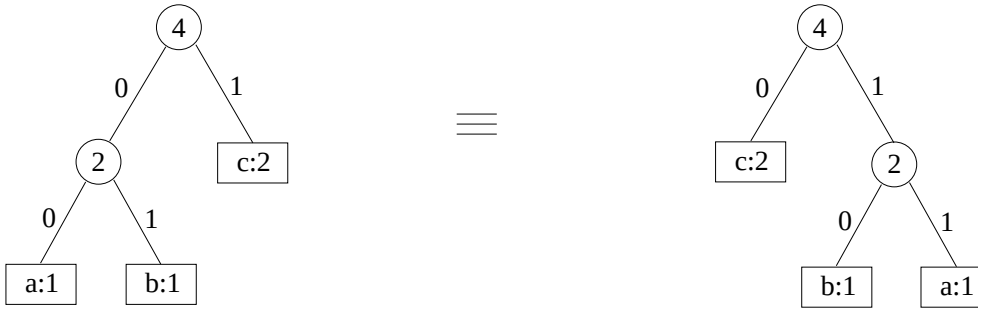

Figure 1: Equivalent Huffman codes for first 3 characters for CLR 16.3-3.

while loop. When the loop exits We have an optimal set of intervals that covers all points.

Correctness: Some interval must cover the lowet valued point x_1. Of all possible intervals that contain x_1 , $[x_1, x_1 + 1]$ is the optimal because no point in the given set has a value is less than x_1. SO setting the lower end of the interval to X_1 maximizes the number of points that this interval can cover. Now we look for the next x_i such that $x_i < x_j$ for $i < j < n$ and that $x_i > x_1 + 1$. There is some interval that must contain the point x_i. Given the first interval that we decided optimal for x_1, this interval is optimal for the point x_i because there is no value smaller than x_i that is not covered and so setting the lower end of this interval to x_i maximizes the number of points that this interval can contain. The same argument can be repeatedly applied for the optimality of each interval chosen.

(6) CLR 16.3-3

Depending on what order the initial queue is in and how the insertion is handled when the priorities are identical, we have two equivalent optimal Huffman codes for the first 3 characters (see Figure 1). If the second tree is used, the Huffman code tree for the first 8 Fibonacci distributed characters is given in Figure 2.

Then, the generalization to the first n Fibonacci distributed characters is as follows:

n^{th} char	0
$(n-1)^{th}$ char	1 0
$\vdots$	
3^{rd} char	1 1 1 $\cdots$ 1 0
2^{nd} char	1 1 1 $\cdots$ 1 1 0
1^{st} char	1 1 1 $\cdots$ 1 1 1 (i.e., (n-1) ones)

(7) CLR 16.3-7

Ternary codes would result in a ternary tree instead of a binary tree. The ternary version of the Huffman algorithm would be the same as shown on pg 388 of CLR

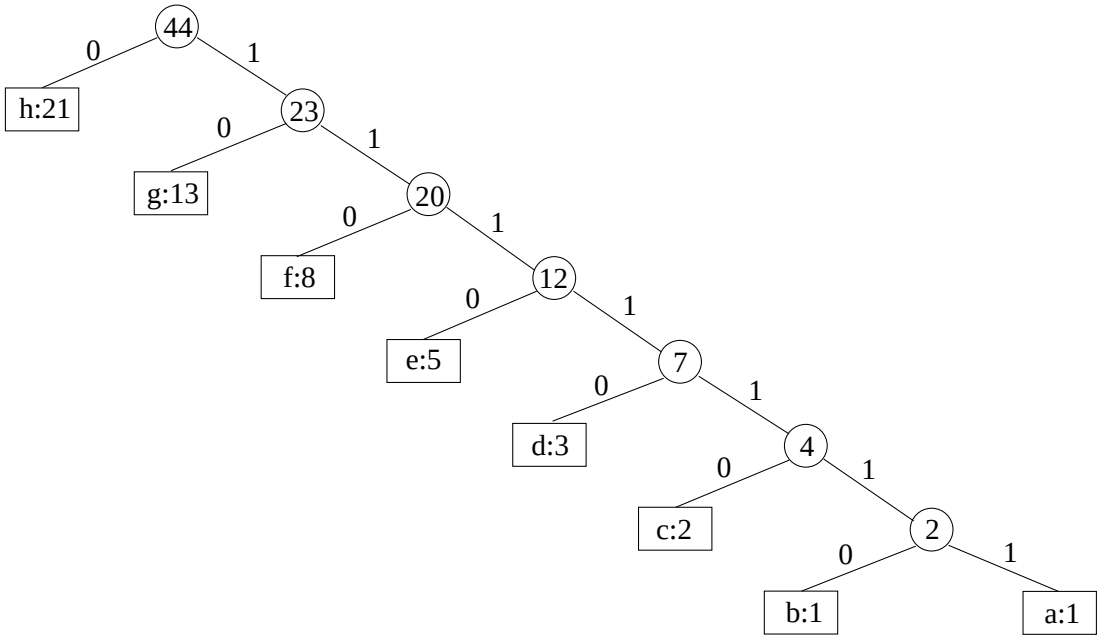

Figure 2: Huffman code for characters with the first 8 Fibonacci numbers frequency distribution for CLR 16.3-3.

except that the lines 5 and 6 would be replaced by three similar lines:
5. $left[z] \leftarrow w \leftarrow$EXTRACT-MIN($Q$)
6. $middle[z] \leftarrow x \leftarrow$EXTRACT-MIN($Q$)
7. $right[z] \leftarrow y \leftarrow$EXTRACT-MIN($Q$)

And line 7 would be replaced by:
$f[z] \leftarrow f[w] + f[x] + f[y]$

Correctness: The correctness of the huffman code relies on Lemmas 16.2 and 16.3 and we shall describe how to modify these lemmas to hold for the ternary version presented.

Lemma 16.2: The proof presented in the book proves that $B(T) - B(T') \geq 0$, where T' is the tree in which $x =$ EXTRACT-MIN(Q) is used as one of the lowest level leaves instead of the arbitrary leaf a in tree T. We can reapply the same proof to the replacement of b by y as claimed in CLR pg 390. Just as the text extends the proof from leaf x to leaf y, we can extend the proof from y to w by the same argument: T'' is the tree in which y replaces a lowest level leaf of T', and the same proof shown on pg 390 can be used to show that $B(T') - B(T'') \geq 0$. Likewise T''' is the tree in which w replaces a lowest level leaf of T'' and by the same proof we can show that $B(T'') - B(T''') \geq 0$. Thus we have established the ternary version of Lemma 16.2.

Lemma 16.3: Use the following expressions for the ternary version:
$C' = C - \{w, x, y\} \cup z$

$f[z] = f[w] + f[x] + f[y]$
T is the tree built with w,x,y as the lowest level leaves. T' is the tree with node z replacing w,x,y and their parent. The proof for the lemma, similar to pg 391 of CLR, is as follows:

$$f[w]d_T(w) + f[x]d_T(x) + f[y]d_T(y) = (f[w] + f[x] + f[y])(d_{T'}(z) + 1)$$
$$f[w]d_T(w) + f[x]d_T(x) + f[y]d_T(y) = f[z]d_{T'}(z) + f[w] + f[x] + f[y]$$

From whish we conclude that $B(T) = B(T') + f[w] + f[x] + f[y]$, and from that point onwards the rest of the proof is identical to that of lemma 16.3 on pg 391.

(8) CLR 16-1abc

First, let us prove that the coin-changing problem has optimal substructure. Suppose we have an optimal solution for a problem of making change for n cents, and we know that this optimal solution uses a coin whose value is c cents; let this optimal solution use k coins. We claim that this optimal solution for the problem of n cents must contain within it an optimal solution for the problem of $n-c$ cents. We use the usual cut-and-paste argument. Clearly, there are $k-1$ coins in the solution to the $n-c$ cents problem used within our optimal solution to the n cents problem. If we had a solution to the $n-c$ cents problem that used fewer than $k-1$ coins, then we could use this solution to produce a solution to the n cents problem that uses fewer than k coin, which contradicts the optimality of our solution.

a. A greedy algorithm to make change using quarters, dimes, nickels, and pennies works as follows:

- Give $q = \lfloor n/25 \rfloor$ quarters. That leaves $n_q = n \bmod 25$ cents to make change.
- Then give $d = \lfloor n_q/10 \rfloor$ dimes. That leaves $n_d = n_q \bmod 10$ cents to make change.
- Then give $k = \lfloor n_d/5 \rfloor$ nickels. That leaves $n_k = n_d \bmod 5$ cents to make change.
- Finally, give $p = n_k$ pennies.

An equivalent formulation is the following. The problem we wish to solve is making change for n cents. If $n = 0$, the optimal solution is to give no coins. If $n > 0$, determine the largest coin whose value is less than or equal to n. Let this coin have value c. Give one such coin, and then recursively solve the subproblem of making change for $n - c$ cents.

To prove that this algorithm yields an optimal solution, we first need to show that the greedy-choice property holds, that is, that some optimal solution to making change for n cents includes one coin of value c, where c is the largest coin value such that $c \leq n$. Consider some optimal solution. If this optimal solution does not include a coin of value c. We have four cases to consider:

- If $1 \leq n < 5$, then $c = 1$. A solution may consist only of pennies, and so it must contain the greedy choice.
- $5 \leq n < 10$, then $c = 5$. By supposition, this optimal solution does not contain a nickel, and so it consists only nickels and pennies. Some subset of the nickel to give a solution with four fewer coins.
- If $10 \leq n < 25$, then $c = 10$. By supposition, this optimal solution does not contain a dime, and so it contains only nickels and pennies. Some subset of the nickels and pennies in this solution adds up to 10 cents, and so we can replace these nickels and pennies by a dime to give a solution with (between 1 and 9) fewer coins.
- If $25 \leq n$, then $c = 25$. By supposition, this optimal solution does not contain a quarter, and so it contains only dimes, nickels and pennies. If it contains three dimes, we can replace these three dimes by a quarter and a nickel, giving a solution with one fewer coin. If it contains at most two dimes, then some subset of the dimes, nickels, and pennies adds up to 25 cents, and so we can replace these coins by one quarter to give a solution with fewer coins.

Thus, we have shown that there is always an optimal solution that includes the greedy choice, and that we can coombine the greedy choice with and optimal solution to the remaining subproblem to produce and optimal solution to our original problem. Therefore, the greedy algorithms produces an optimal solution.

For the algorithm that chooses one coin at a time and then recurses on subproblems, the running time is $\theta(k)$, where k is the number of coins used in an optimal solution. Since $k \leq n$, the running time is $O(n)$. For our first description of the algorithm, we perform a constant number of calculations (since there are only 4 coin types), and the running time is $O(1)$.

b. When the coin denominations are c^0, c^1,...,c^k, the greedy algorithm to make change for n cents works by finding the denomination c^j such that $j = max\{0 \leq i \leq k : C^i \leq n\}$, giving one coin of denomination c^j, and recursing on the subproblem of making change for $n - c^j$ cents. (An equivalent, but more efficient, algorithm is to give $\lfloor n/c^k \rfloor$ coins of denomination c^k and $\lfloor (nmodc^{j+1})/c^j \rfloor$ coins of denomination c^j for $i = 0, 1, ..., k-1$.)

To show that the greedy algorithm produces an optimal solution, we start by proving the following lemma:

Lemma
For $i = 0, 1, ..., k$, let a_i be the number of coins of denomination c^j used in an optimal solution to the problem of making change for n cents. Then for $i = 0, 1, ..., k-1$, we have $a_i < c$.

Proof If $a_i \geq c$ for some $0 \leq i < k$, then we improve the solution by using one more coin of denomination c^{j+1} and c fewer coins of denomination c^j. The amount for which we make change remains the same, but we use $c - 1 > 0$ fewer coins.

To show that the greedy solution is optimal, we show that any non-greedy solution is not optimal. As above, let $j = max\{0 \leq i \leq k : c^j \leq n\}$, so thatthe greedy solution uses at least one coin of denomination c^j. Consider a non-greedy solution, which must use no coins of denomination c^j or higher. Let the non-greedy solution use a_i coins of denomination c^i, for $i = 0, 1, ..., j-1$; thus we have $\sum_{i=0}^{j-1} a_i c^j = n$. Since $n \geq c^j$, we have that $\sum_{i=0}^{j-1} a_i c^j \geq c^j$. Now suppose that the non-greedy solution is optimal. By the above lemma, $a_i \leq c - 1$ for $i = 0, 1, ..., j-1$. Thus,

$$\begin{aligned}
\sum_{i=0}^{j-1} a_i c^i &\leq \sum_{i=0}^{j-1} (c-1)c^i \\
&= (c-1)\sum_{i=0}^{j-1} c^i \\
&= (c-1)\frac{c^j - 1}{c-1} \\
&= c^j - 1 \\
&< c^j,
\end{aligned}$$

which contradicts our earlier assertion that $\sum_{i=0}^{j_1} a_i c^j \geq c^j$. We conclude that the non-greedy solution is not optimal. Since any algorithm that does not produce the greedy solution fails to be optimal, only the greedy algorithm produces the optimal solution.

c. With actual U.S. coins, we can use coins of denomination 1, 10, and 25. When $n = 30$ cents, the greedy solution gives one quarter and five pennies, for a total of six coins. The non-greedy solution of three dimes is better.

The smallest integer numbers we can use are 1, 3, and 4. When $n = 6$ cents, the greedy solution gives one 4-cent coin and two 1-cent coins, for a total of three coins. The non-greedy solution of two 3-cent coins is better.