Atanas Dimitrov
Assignment #3 - II
CS 6470
Dr. Cai

Solution to problem 1.

1. Optimal substructure:

Let a_k denote activity k, s_k - starting time of activity k, and f_k - finishing time of activity k.
Let set $S_{ij} = \{a_k \in S : f_i \leqq s_k < f_k \leqq s_j\}$ be the subset of activities in S that can start after activity a_i finishes and finish before a_j starts. Let $f_0=0$ and $s_{n+1}=\infty$, then $S = S_{0,n+1}$ and $0 \leqq i,j \leqq n+1$.

Let all the activities be sorted in monotonically increasing order of starting time:

$$s_0 > s_1 \geqq s_2 \geqq \ldots \geqq s_n \geqq s_{n+1}$$

We can clearly see $S_{ij}=\emptyset$ when $i \geqq j$ since it would lead to contradiction otherwise. So we can state that $0 \leqq i < j \leqq n+1$ and all other S_{ij} are empty.

We can see the optimal substructure property when we take under consideration a subproblem S_{ij} and choose a_k such that $f_i \leqq s_k < f_k \leqq s_j$. We want to show that the optimal solution is the union of a_k S_{ik} and S_{kj}. Suppose A_{ij} is an optimal solution to S_{ij} using a_k. Then the solution A_{ik} to S_{ik} and A_{kj} to S_{kj} are also optimal: suppose they were not then we could substitute with a more optimal solution and thus make A_{ij} more optimal which contradicts the initial assumption that A_{ij} is already optimal.

2. Recursive solution:

Let c[i,j] be the number of activities in a maximum size subset of mutually exclusive activities in Sij then:

$$c[i,j] = \begin{cases} 0 & \text{when } S_{ij}=\emptyset,\ i \geqq j \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\} & \text{when } S_{ij} \not\equiv \emptyset,\ i<j \end{cases}$$

3. It is always safe to make the greedy choice, since the greedy choice is always one of the optimal choices:

Consider any nonempty S_{ij} and let a_m be the activity in S_{ij} with the latest starting time:

$$s_m = \max\{s_k : a_k \in S_{ij}\}$$

Then a_m is used in some maximum size subset of mutually compatible activities of S_{ij} and the subproblem S_{mj} is empty. To prove this, suppose S_{mj} is nonempty and there is an activity a_k such that $s_m < f_m \leqq s_k < f_k \leqq s_j$, which is a contradiction to our initial assumption. We conclude S_{im} is empty.

Now let a_k be the activity with latest starting time in the solution A_{ij}.If $a_k=a_m$ then we are done. if not, then let $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$. Since a_m has later start time then a_k we can be sure that the activities in A'_{ij} are mutually compatible and since A'_{ij} has the same number of activities as A_{ij} then A'_{ij} is the maximum size subset that contains a_m.

4. All but one of the subproblems induced by making the greedy choice are empty:
This step follows from step 3 i.e. “S_{mj} is empty” leaving S_{im} to be the only one that may be nonempty after choosing a_m.

5. Recursive greedy algorithm:
Input: Set of activities with starting time s and finish time f. s - activity starting times array, f - activity finishing times array, i - index, n - index.
Output: Maximum size set of mutually compatible activities in $S_{i,n+1}$.
Algorithm: We assume that the n input activities are sorted by monotonically increasing starting time. If not we can sort them in O(n*logn):

```
Recursive-Activity-Selector2 (s, f, i, n)
    i ← n
    m ← i - 1
    while m>0 and $f_m > s_i$
        do m ← m - 1
    if m>0
        then return {$a_m$} ∪ Recursive-Activity-Selector2 (s, f, i, m)
        else return ∅
```

Assuming the activities are sorted in starting time then T(n) = O(n) because of the while loop. Otherwise we will have T(n) = O(n) + O(nlogn) = O(nlogn)

6. Iterative greedy algorithm:
Input: Given a list of activities the inputs are their starting (s) and finish (f) time.
Output: Maximum size set of mutually compatible activities in the list.
Algorithm: We assume that the n input activities are sorted by monotonically increasing starting time. If not we can sort them in O(n*logn):

```
Greedy-Activity-Selector2 (s, f)
    n ← length[s]
    A ← {$a_n$}
    i=n
    for m ← i - 1 to 1
        do if $f_m \leqq s_i$
            then A ← A ∪ {$a_m$}
                i ← m
    return A
```

Assuming the activities are sorted in starting time then T(n) = O(n) because of the for loop. Otherwise we will have T(n) = O(n) + O(nlogn) = O(nlogn)

Solution to problem 2.
1. Optimal substructure:
Suppose that the total weight of the bag is W and the total number of items is n. Let the list of

items be sorted monotonically decreasing by value, thus the item with largest value (and subsequently smallest weight) is first followed by the item with next largest value etc. Let A_n be an optimal solution which contains some items from the set of n items. First let A_n contain the n-th item the the maximum profit is the profit of this item added to A_{n-1} - the optimal solution for the first n-1 numbers as long as this sum doesn't exceed W. Otherwise we simply do not include item n and thus A only consists of A_{n-1}.
Suppose A_{n-1} is not optimal then we can substitute it with A'_{n-1} which is more optimal and thus making A more optimal but this contradicts the initial assumption that A is optimal.

2. Recursive Solution:
Let S be a set of n items S[1..n], w_i - the profit of item i, p_i - the profit of item i, P[i][w] be the optimal solution (profit) from choosing from the first i items not exceeding w in weight then:

$$P[i][w] = \begin{cases} \max\{P[i-1][w], P[i-1][w-w_i]+p_i\} & \text{when } w_i \leqq w \\ P[i-1][w] & \text{when } w_i > w \end{cases}$$

3. It is always safe to make the greedy choice, since the greedy choice is always one of the optimal choices:
Suppose we have arrived at an optimal solution P[i][w] among items 1 to i by choosing item k then suppose that in items 1 through i there is item m with the largest value and smallest weight which has not been chosen. If m=k then we are done since we can interchange them and get an equivalent optimal solution. If m is not equal to k then we can substitute k with m because we are guaranteed that m has less weight then k and larger profit then k. Thus m is always a part of an optimal solution since m,k,i, and w are arbitrary.

4. All but one of the subproblems induced by making the greedy choice are empty:
Since we are discarding or including one element at a time and considering as a subproblem the rest of the elements then we only have 1 subproblem which may be non-empty each time.

5. Recursive greedy algorithm:
Input: Knapsack and set of items. List of items I, indexes i and n, W - maximum capacity left in the knapsack, w - array to hold weights, p - array to hold profits (values).
Output: List of items, which maximize the value of items in the knapsack while minimizing its weight.
Algorithm: Suppose that the list of elements are sorted by weight, then:

```
Recursive-Item-Selector (I, i, n, W, w, p)
    $m \leftarrow i$
    if $w_m \leqq W$ and $m \leqq n$
        then return $\{I_m\} \cup$ Recursive-Item-Selector (I, m+1, n, W - $w_m$, w, p)
    else
        return $\varnothing$
```

The time complexity is O(n) since we go through at most n elements - or n calls to the function -

(when all items fit in the knapsack) + time for sorting -- O(nlogn), thus T(n) = O(n) + O(nlogn) = O(nlogn)

6. Iterative greedy algorithm:
Input: Knapsack and set of items. List of items I, indexes i and n, W - maximum capacity left in the knapsack, w - array to hold weights, p - array to hold profits (values).
Output: List of items, which maximize the value of items in the knapsack while minimizing its weight.
Algorithm: Suppose that the list of elements are sorted by weight, then:

```
Item-Selector (I, n, W, w, p)
        $A \leftarrow \emptyset$
        $m \leftarrow 1$
        while $w_m \leqq W$ and $m \leqq n$
                then $W \leftarrow W - w_m$
                     $A \leftarrow A \cup \{I_m\}$
                     $m \leftarrow m + 1$
        return A
```

The time complexity is O(n) since we go through at most n elements - or n calls to the function - (when all items fit in the knapsack) + time for sorting -- O(nlogn), thus T(n) = O(n) + O(nlogn) = O(nlogn)

Solution to problem 3:
As the textbook suggests (p.386) an optimal code for a file is always represented by a full ternary tree. Let the alphabet from which characters are drawn be C. Then the optimal solution will have at least |C| leaves. In the tree every parent/internal node can have 3 children. Given a tree T corresponding to a prefix code, let f(c) to denote the frequency of character c in the file, and $d_T(c)$ be the depth of c's leaf on the tree, then the number of bits $B(T)=(c\in C)\sum f[c]d_T(c)$.

Input: Block of characters C.
Output: Optimal prefix code tree for C.
Algorithm: The idea is the same as in the algorithm outlined in the textbook however there will be less merges (n/2 n is the number of leaves in a full ternary tree) due to the fewer internal nodes needed. Q is a binary min-heap, right[i]/middle[i]/left[i] describe the right/middle/left elements respectively. Extract-min(Q), Insert(Q, i) are functions associated with manipulating the binary min-heap for extracting the minimum element and inserting an element respectively as described in Chapter 6. Initialization of Q can be done by Build-Min-Heap() from Section 6.3.

```
Huffman-ternary(C)
        $n \leftarrow |C|$
        $Q \leftarrow C$
        for $i \leftarrow 1$ to floor(n/2)
        do allocate a new node w
            left[w] ← x ← Extract-Min(Q)
            middle[w] ← y ← Extract-Min(Q)
            right[w] ← z ← Extract-Min(Q)
            f[w] ← f[x] + f[y] + f[z]
            Insert(Q, w)
    return Extract-Min(Q)
```

Each heap operation requires O(lgn) time. The heap is built in O(n). The for loop will execute O(n/2)=O(n) times. Thus the time complexity for the algorithm T(n) = O(n) + O(n)*O(lgn) = O(n) + O(nlgn) = O(nlgn).

To show that **the greedy property holds** we prove the following:
Let C be an alphabet in which each character c∈C has frequency f[c]. Let x, y, z be three characters in C having the lowest frequencies then there exists an optimal prefix code for C in which the codewords for x, y, z have the same length and differ by the last bit (i.e. 0,1, or 2).
Proof: Let a, b, d be three characters that are sibling leaves of maximum depth in T, we assume f[a] ≦ f[b] ≦ f[d], and f[x] ≦ f[y] ≦ f[z]. Since f[x], f[y], f[z] are the three lowest leaf frequencies, in order, and f[a], f[b], f[d] are arbitrary then we have f[x]≦f[a], f[y]≦f[b], and f[z]≦f[d]. Let us exchange the positions of a and x to produce tree T', b and y to produce T'', and d and z to produce T'''. Thus using the equation for the cost we have:

$$\begin{aligned} B(T) - B(T') &= (c\in C)\sum f(c)d_T(c) - (c\in C)\sum f(c)d_{T'}(c) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) + f[a]d_{T'}(a) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) + f(a)d(x) \\ &= (f[a] - f[x])(dT(a) - dT(x)) \\ &\geqq 0 \text{ since both terms are nonnegative (same as in textbook).} \end{aligned}$$

Since, B(T) - B(T') ≧ 0, then B(T) ≧ B(T'). Similarly we prove the relationship B(T') ≧ B(T''), and B(T'')≧B(T'''), thus we can conclude B(T) ≧ B(T''') Since T is optimal then it must be the case that B(T) = B(T'''), thus T''' is optimal and x, y, and z appear as sibling leaves at maximum depth. (In the above discussion we assume that the ternary tree is full, however it could be the case that there are not enough characters in C to fill the tree. Then we can simply fill in with 0's and the argument will still hold. For the remaining of the solution I assume that there will always be enough characters in C to comprise a full ternary tree) Thus we can conclude that the greedy choice of always merging the least frequent characters while building the tree is always safe.

The **optimal-substructure property also holds**. Let C be a given alphabet with frequency f[c] defined for each c∈C. Let x, y and z be three characters in C with min frequency. Let C'=C - {x,y,z} ∪ {w} where z is a character added to C' while x, y, z are removed. define f for C' the same but without f[w]=f[x]+f[y]+f[z]. Let T' be any tree representing an optimal prefix code for C'. Then the tree T obtained from T' by replacing the leaf node w with an internal node having x, y, z as children, represents an optimal prefix code for alphabet C.

Proof:
First we show that $B(T)$ of T can be expressed using $B(T')$ of T'. For all $c \in C - \{x,y,z\}$ we have $d_T(c) = d_{T'}(c)$, and hence $f[c]d_T(c) = f[c]d_{T'}(c)$. Also we know that $d_T(x) = d_T(y) = d_T(z) = d_{T'}(w) + 1$. Thus we have:

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) + f[z]d_T(z) &= (f[x]+f[y]+f[z])(d_{T'}(w) + 1) \\ &= f[w]d_{T'}(w) + (f[x] + f[y] + f[z]) \end{aligned}$$

Thus we can conclude that:

$$B(T) = B(T') + (f[x] + f[y] + f[z]) \text{ or } B(T') = B(T) - (f[x] + f[y] + f[z])$$

Suppose that T doesn't represent an optimal prefix code for C. Then there exists T'' such that $B(T'') < B(T)$. We know from the previous proof that in T'' x, y, z are siblings. Let T''' be the tree T'' with the common parent of x,y,z replaced by a leaf w with frequency $f[w]=f[x]+f[y]+f[z]$. Then:

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] - f[z] \\ &< B(T) - f[x] - f[y] - f[z] \\ &= B(T') \end{aligned}$$

Thus this is a contradiction that T' is optimal for C'. Then T must be optimal prefix code for C. Also, procedure **Huffman-ternary(C) does produce an optimal prefix code** based on the above proofs.