论题 2-13 作业

姓名：陈劭源　　学号：161240004

1 [TC] Problem 16.1-2

Selecting the last activity to start that is compatible with all previously selected activities is the best choice locally, and we make the greedy choice in every step, so this is a greedy algorithm.

Let a_i $(1 \leq i \leq n)$ denote the activities with start time s_i and finish time f_i, which are sorted in monotonously decreasing order of the start time, i.e. $s_i \geq s_{i-1}$. Let $S_k = \{a_i \in S : f_i \leq s_k\}$ denote the set of activities that finish before activity a_k starts. We also let S_k denote the subproblem that choosing maximum number of compatible activities from S_k.

We have proved in the textbook that this problem has an optimal substructure, so we only have to prove the greedy property, which reads:

> Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the last start time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Let A_k be a maximum size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the last start time. If $a_j = a_m$ we are done. Otherwise, let the set A'_k be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, because the activities in A_k are disjoint, a_j is the first activity in A_k to start, and $s_m \geq s_j$. Therefore, A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m.

2 [TC] Problem 16.1-3

Counterexample of the approach of selecting the activity of least duration from among those that are compatible with previously selected activity: there are three activities, whose start times are 0, 3, 5, and finish times are 4, 6, 9, respectively. This approach only chooses the second activity, while we can choose the first and the third activities which are compatible.

Counterexample of the approach of always selecting the compatible remaining activity that overlaps the fewest other remaining activities: there are 11 activities, whose start times are 0, 1, 2, 3, 4, 7, 8, 9, 10, 11, 12, and finish times are 4, 7, 6, 5, 8, 9, 12, 15, 14, 13, 16, respectively. This approach chooses the first, the sixth and the last activities, while choosing the first, the fifth, the seventh and the last activities is a better solution.

Counterexample of always selecting the compatible remaining activity with the earliest start time: there are three activities, whose start times are 0, 1, 3, and finish times are 5, 2, 4, respectively. This approach only chooses the first activity, while we can choose the second and the third activities, which are compatible.

3 [TC] Problem 16.2-1

The greedy-choice property of the fractional knapsack problem reads:

> The thief repeats taking the items with the greatest value per pound, until he reaches the weight limit, or there is nothing left, or if he cannot carry the current item in whole, then he carries part of the current item to reach the limit exactly.

Suppose, to the contrary, that the greedy-choice property does not hold. There are two cases:

Case 1: he does not reach the weight limit, but there is still something left. If he continues to take something left, either in part or in whole, the total value increases.

Case 2: he takes something less valuable than something he does not take, either in part or in whole. Instead of taking the things less valuable, he takes the thing more valuable, and the total value increases.

Therefore, the fractional knapsack problem has greedy-choice property.

4 [TC] Problem 16.2-2

ZERO-ONE-KNAPSACK(n, W, v, w)

```
1  let tot[0..n][0..W], c[1..n][0..W] be new tables
2  for j = 0 to w
3      tot[0][j] = 0
4      for i = 1 to n
5          for j = 0 to W
6              if w[i] > j or tot[i-1][j-w[i]] + v[i] ≤ tot[i-1][j]
7                  tot[i][j] = tot[i-1][j]
8                  c[i][j] = FALSE
9              else
10                 tot[i][j] = tot[i-1][j-w[i]] + v[i]
11                 c[i][j] = TRUE
12     j = W
13     for i = n downto 1
14         if c[i][j]
15             print "take item" i
16             j = j - w[i]
```

5 [TC] Problem 16.3-2

If the binary tree is not full, then there must exist a non-leaf node which has less than two children. If it has no child, it is useless and we can remove it. After removing the node, its parent has less than two children. Repeat doing this and we can get a node which has exactly one child. Transplant its subtree to its parent, we get another valid code, where the codeword length of every character in the subtree decreases, which is a better code. Therefore, a binary tree that is not full cannot correspond to an optimal prefix code.

6 [TC] Problem 16.3-5

Suppose, to the contrary that there does not exist such optimal code, i.e. in the optimal code, there exists two characters, say 'a' and 'b', such that 'a'.$freq >$ 'b'.$freq$ and the codeword of 'a' is longer than 'b'. Swap the codeword of 'a' and 'b', we get another valid code, but is more efficient than the original one. Therefore, the codeword lengths must be monotonically increasing if the frequency of the characters are monotonically increasing.

7 [TC] Problem 16.3-8

Since the maximum character frequency is less than twice the minimum character frequency, if we merge any two of them, the frequency of the new node is larger than the 256 original nodes, which means the new nodes will not be merged until all the old nodes are merged. Consider the 128 new nodes, the maximum frequency is still less than twice the minimum frequency. Likewise, the 64 new nodes will not be merged until all the 128 nodes are merged. So do the 64, 32, 16, 8, 4, and 2 nodes. Finally, we obtain a tree, in which every leaf has the same height 8. Therefore, Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

8 [TC] Problem 16-1

a. We can repeat choosing the coin with the largest denomination from those that do not exceed the remaining part.

MAKE-CHANGE-GREEDY(n)

1 $quarters = \lfloor n/25 \rfloor$
2 $n = n - 25 * quarters$
3 $dimes = \lfloor x/10 \rfloor$
4 $n = n - 10 * dimes$
5 $nickels = \lfloor n/5 \rfloor$
6 $pennies = n - 5 * nickels$
7 print $quarters$, $dimes$, $nickels$, $pennies$

Let Q, D, N, P denote the numbers of the quarters, dimes, nickels and pennies, respectively. To prove the correctness of the algorithm, we have to prove $10D + 5N + P < 25$ (*), $5N + P < 10$ (**) and $P < 5$ (***). For optimal solution, we claim that $D < 5$, $N < 2$ and $P < 5$. Otherwise, we can replace these coins with two quarters, a dime or a nickel, respectively, which yields a better solution. By $P < 5$, the inequality (***) holds. By $P < 5$ and $N < 2$, the inequality (**) holds. When $D = 3$, we can replace them with a quarter and a nickel; when $D = 4$, we can replace then with a quarter, a dime and a nickel, which are all better solutions. When $D = 2$ and $5N + P \geq 5$, by $P < 5$ and $N < 2$ we know that $N = 1$. Replacing the two dimes and a nickel with a quarter yields a better solution. Hence, either $D < 2$ or $D = 2$ and $5N + P < 5$ holds, thus inequality (*) holds. Therefore, this algorithm yields an optimal solution.

b. Let $p_0, p_1, \cdots, p_k$ denote the numbers of coins with denominations $c^0, c^1, \cdots c^k$, respectively. For optimal solution, we claim that $p_0, p_1, \cdots p_{k-1} < c$, otherwise, we can replace these coins with the coin with denomination $c, c^2, \cdots c^k$, respectively. Now we are going to prove that $\sum_{i=0}^{j} n_i c^i < c^{j+1}$ holds for all $j < k$:

$$\sum_{i=0}^{j} p_i c^i \leq \sum_{i=0}^{j} (c-1)c^i = \sum_{i=1}^{j+1} c^i - \sum_{i=0}^{j} c^i = c^{j+1} - 1 < c^{j+1}$$

Therefore, the greedy algorithm always yields an optimal solution.

c. The coins are in the denominations 1, 3 and 4 cents, and we are going to make a 6-cent change. The greedy algorithm gives a solution using one 4-cent coin and two 1-cent coins. However, using two 3-cent coins is a better solution.

d. The pseudocode is shown below. d is an array consisting of the denominations available. This procedure returns an array, consisting the number of every kind of coin.

```
MAKE-CHANGE-DP(d, n)
 1  k = d.length
 2  let p[0..n], t[0..n], c[1 → k] be new arrays
 3  p[0] = 0
 4  for i = 1 to n
 5      p[i] = ∞
 6      for j = 1 to k
 7          if i ≥ d[j] and p[i − d[j]] + 1 < p[i]
 8              p[i] = p[i − d[j]] + 1
 9              t[i] = j
10  for j = 1 to k
11      c[j] = 0
12  i = n
13  while i ≠ 0
14      c[t[i]] = c[t[i]] + 1
15      i = i − p[t[i]]
16  return c
```