EECS 336

Litong Wang

Homework 5

May 9, 2016

Problem 1

Solution:

We should go as far as possible before we need to refill the tank. Let's assume a_i to be the gas station which has distance d_i from Chicago, A to be the set contains gas station where we should stop.

Algorithm:

Data: d_i, m
Result: A
$A = \emptyset$;
$laststop = 0$;
for $i = 1; i \leq n; i++$ **do**
  if $d_i - laststop > m$ **then**
    $A = A \cup a_{i-1}$;
    $laststop = d_{i-1}$;
  end
end
$return\ A$;

Correctness:

In order to prove correctness of this algorithm, we need to show that this problem has the greedy-choice and optimal-substructure properties.

The following lemma proves the greedy-choice property holds.

Lemma 1:

Let A be an optimal solution for stoping and the sequence of stops is $\{a_1, a_2, \cdots, a_k\}, k \leq n$. Assume a^* is the first gas station that we can go as far as possible as the algorithm told us to choose. There is an optimal solution that uses s^* as the first stop.

Proof of Lemma 1:

If $a_1 = a^*$, then A is the optimal solution that uses a^* as the first stop, we don't need to do further work. If $a_1 \neq a^*$, according to the algorithm, we need to choose the first stop as far as possible, then we can know that $d_{a_1} \leq d_{a^*}$. Consider a solution A' which has the sequence of stops is $\{a^*, a_2, \cdots, a_k\}$, since $d_{a_1} \leq d_{a^*}$, then $d_{a_2} - d_{a_1} \geq d_{a_2} - d_{a^*}$. And because

$d_{a_2} - d_{a_1} \leq m$, we know that $d_{a_2} - d_{a^*} \leq m$. Therefore, A' is a legitimate solution. Since $|A| = |A'|$, then we can get the conclusion that A' is an optimal solution.
The following lemma proves the optimal-substructure property holds.
Lemma 2:
Let's assume that A is an optimal solution A to the original problem P, then A includes an optimal solution A' to the problem P' that after stoping at a_i, we need to drive from a_i to Miami given $d_{i+1}, d_{i+2}, \cdots, d_n$ and stop as few as possible.
Proof of Lemma 2:
Let's assume we have an optimal solution A' to problem P', we can find that $A = A' \cup a_i$. Hence, $Cost(A) = Cost(A') + 1$. Therefore, the optimal solution A to the original problem contains an optimal solution A' to a subproblem P'.

Problem 2

Solution:

(a) increasing order of weight:

This algorithm is incorrect. Let's assume that we have two knapsacks both have capacities of 5. And we have five items with the following values and weights:
item1: weight = 1, value = 1, item2: weight = 2, value = 2, item3: weight = 3, value = 3, item4: weight = 4, value = 4, item5: weight = 5, value = 100
According to the algorithm, we should choose the item with less weight to fill our knapsacks first. Then we can get the result: we fill one knapsack with item1 and item4 and fill another knapsack with item2 and item3, hence, the total value of this solution is 10. However, we can easily know that the optimal solution is filling one knapsack with item5 and another knapsack with item1 and item4 or item2 and item3. Then, we can get a total value of 105.

(b) decreasing order of value:

This algorithm is incorrect. Let's assume that we have two knapsacks both have capacities of 5. And we have three items with the following values and weights:
item1: weight = 10, value = 20, item2: weight = 5, value = 15, item3: weight = 4, value = 14
According to the algorithm, we should choose the item with higher value to fill our knapsacks first. Then we can get the following result: we fill one knapsack with half of item1 and fill another knapsack with item2, hence, the total value of this solution is 25. However, we can easily find out that the optimal solution is filling one knapsack with item5 and another knapsack with item3 and one tenth of item1. This solution gives us a higher total value which is 31.

(c) decreasing order of value to weight ratio:

This algorithm is incorrect. Let's assume that we have knapsack with capacity of 10 and one with capacity of 20. And we have two items with the following values and weights:
item1: weight = 10, value = 20, item2: weight = 20, value = 20
According to the algorithm, we should choose the item with higher value to weight ratio to fill the knapsack with larger capacity first. Then we can get the following result: we fill the knapsack with capacity of 20 with item1 and fill the another knapsack with half of the item2, hence, the total value of this solution is 30. However, we can easily find that the optimal solution is filling the first knapsack with item2 and filling the second knapsack with item1. This solution gives us a higher total value which is 40.

Problem 3

Solution:

The construction of a tree for 6-ary codewords is similar to the construction of a tree for huffman code. Because we have 6 character codes and we need to construct a tree, we would construct a tree with $6 + 5k$ leaves. If we have n characters with frequence $f_i(1 \leq i \leq n)$ and we cannot find a integer k such that $n = 6 + 5k$, we need to find the smallest integer k^* such that $6 + 5k^* > n$. We need to create m dummy nodes with frequence $f_i = 0(1 \leq i \leq m)$ so that $6 + 5k^* = m + n$. Hence, we get m+n characters with frequence $f_i(1 \leq i \leq m + n)$.
Algorithm:
We first put all the characters into a min-priority queue Q using its frequency as key. Then, we find the 6 characters with smallest frequency as leaves, then we construct a node with the sum of its six children's frequency as its frequency and push this node into the queue. We keep doing this until there is no character in queue.

Data: C
Result: A
$n = m + |C|$;
$Q = C + m$ *dummy nodes* ;
for $i = 1; i \leq n - 1; i++$ **do**
    Allocate a new node z ;
    for $j = 1; j \leq 6; j++$ **do**
        $z.j = EXTRACT - MIN(Q)$;
    end
    $z.freq = \sum_{i=1}^{6} z.i.freq$;
    $Q = Q + z$;
end
$return\ EXTRACT - MIN(Q)$;
Correctness:

In order to prove correctness of this algorithm, we need to show that this problem has the greedy-choice and optimal-substructure properties.

Lemma 1:

Let C be an alphabet in which each character $char \in C$ has frequency $char.freq$. Let a, b, c, d, e, f be six characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords a, b, c, d, e, f have the same length and differ only in the last bit. This lemma proves the greedy-choice property holds.

Proof of Lemma 1:

Let g, h, i, j, k, l be the six characters that are sibling leaves with maximum depth in tree T which represents an optimal prefix code. Let's assume that $a.freq \leq b.freq \leq c.freq \leq d.freq \leq e.freq \leq f.freq$, also $g.freq \leq h.freq \leq i.freq \leq j.freq \leq k.freq \leq l.freq$. Since a, b, c, d, e, f are six characters in C having the lowest frequencies, and we can have $a.freq \leq g.freq, b.freq \leq h.freq, c.freq \leq i.freq, d.freq \leq j.freq, e.freq \leq k.freq, f.freq \leq l.freq$. By switching pairs such as $\{a, g\}, \{b, h\}, \{c, i\}, \{d, j\}, \{e, k\}, \{f, l\}$, we can obtain six trees $T_0, T_1, T_2, T_3, T_4, T_5$. The difference in cost between T and T_0 is:

$$
\begin{aligned}
Cost(T) - Cost(T_0) &= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T_0}(c) \\
&= a.freq \cdot d_T(a) + g.freq \cdot d_T(g) - a.freq \cdot d_{T_0}(a) - g.freq \cdot d_{T_0}(g) \\
&= a.freq \cdot d_T(a) + g.freq \cdot d_T(g) - a.freq \cdot d_T(g) - g.freq \cdot d_T(a) \\
&= (g.freq - a.freq)(d_T(g) - d_T(a)) \\
&\geq 0
\end{aligned}
$$

We can apply the same calculation to T_1, T_2, T_3, T_4, T_5 and get the conclusion that $Cost(T) \geq Cost(T_0) \geq Cost(T_1) \geq Cost(T_2) \geq Cost(T_3) \geq Cost(T_4) \geq Cost(T_5)$. Therefore $Cost(T_5) \geq Cost(T)$, since T is an optimal solution, we have $Cost(T) \geq Cost(T_5)$, which implies $Cost(T) = Cost(T_5)$. Thus, T_5 is an optimal tree in which a, b, c, d, e, f appear as sibling leaves of maximum depth, from which the lemma follows.

Lemma 2:

Let C be an alphabet in which each character $char \in C$ has frequency $char.freq$. Let a, b, c, d, e, f be six characters in C having the lowest frequencies. Let C' be the alphabet which removes a, b, c, d, e, f from C and adds a new character z into it. $C' = C - \{a, b, c, d, e, f\} + \{z\}, z.freq = a.freq + b.freq + c.freq + e.freq + d.freq + f.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T , obtained from T' by replacing the leaf node for z with an internal node having a, b, c, d, e, f as children, represents an optimal prefix code for the alphabet C.

Proof of Lemma 2:

Cost of T can be represented as:

$$
\begin{aligned}
\sum_{char=a,b,c,d,e,f} char.freq \cdot d_T(char) &= \sum_{char=a,b,c,d,e,f} char.freq \cdot (d_{T'}(z) + 1) \\
&= z.freq \cdot d_{T'}(z) + \sum_{char=a,b,c,d,e,f} char.freq
\end{aligned}
$$

Hence, we can get the equation:

$$
Cost(T) = Cost(T') + \sum_{char=a,b,c,d,e,f} char.freq
$$

Assume that T cannot represent an optimal prefix code for C, then we can find T'' such that $Cost(T'') < Cost(T)$. In this case, we can build a tree T''' represent a prefix code for C'. We also can get the equation:

$$
Cost(T'') = Cost(T''') + \sum_{char=a,b,c,d,e,f} char.freq
$$

Hence,

$$\begin{aligned} Cost(T''') &= Cost(T'') - \sum_{char=a,b,c,d,e,f} char.freq \\ &< Cost(T) - \sum_{char=a,b,c,d,e,f} char.freq \\ &= Cost(T') \end{aligned}$$

Which means the cost of T''' is smaller than cost of T'. This contradicts our assumption that T' is an optimal prefix code for C'. Therefore, T should be an optimal prefix code for C.

Problem 4

Solution:

We should sort sets A and B into monotonically increasing order. Then we can use this order to achieve the maximum payoff $\Pi_{i=1}^{n} a_i^{b_i}$.

Proof:

Consider any two indices such that $i < j$, we can have two items, $a_i^{b_i}$ and $a_j^{b_j}$. We need to prove that the payoff which includes $a_i^{b_i}$ and $a_j^{b_j}$ is greater than payoff that includes $a_i^{b_j}$ and $a_j^{b_i}$. We only need to prove that $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$. Because A and B are sorted into monotonically increasing order, $a_i \leq a_j$ *and* $b_i \leq b_j$. We divide $a_i^{b_i} a_j^{b_j}$ and $a_i^{b_j} a_j^{b_i}$ by $a_j^{b_i} a_i^{b_i}$, then we can have $a_j^{b_j - b_i}$ and $a_i^{b_j - b_i}$. Because $b_j - b_i \geq 0$ and $a_j \geq a_i$, we can have $a_j^{b_j - b_i} \geq a_i^{b_j - b_i}$.

$$a_j^{b_j - b_i} \geq a_i^{b_j - b_i} \Leftrightarrow a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$$

The running time for sorting is $\mathcal{O}(n \log n)$ and the time for computing payoff is $\mathcal{O}(n)$. Hence, the running time for this algorithm is $\mathcal{O}(n \log n)$.

Problem 5

Solution:

Quarter = 25 cents, dime = 10 cents, nickel = 5 cents and penny = 1 cent.

The coin changing problem has optimal substructure. Assume we have an optimal solution for making change for n cents, we use a coin whose value is v and we use k coins in this solution. This optimal solution must contain an optimal solution for problem of making change for n-v cents. k-1 coins are used in the optimal solution to n-v cents problem used within optimal solution to the n cents problem. If we can find a solution that uses coins fewer than k-1 coins for n-v cents problem, then we can cut and paste this solution to the n cents problem. In this case, we can find a solution for n cents problem that uses coins fewer

than k. This contradicts the optimality of the solution.

a) A greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies:
Let's denote $count_{quarter}$ to be the number of quarters used in the solution. $count_{quarter} = \lfloor \frac{n}{25} \rfloor$, and the coins remain are $n_q = n \mod 25$.
Let's denote $count_{dime}$ to be the number of quarters used in the solution. $count_{dime} = \lfloor \frac{n_q}{10} \rfloor$, and the coins remain are $n_d = n_q \mod 10$.
Let's denote $count_{nickel}$ to be the number of quarters used in the solution. $count_{nickel} = \lfloor \frac{n_d}{5} \rfloor$, and the coins remain are $n_p = n_n \mod 5$. $count_{penny} = n_p$
Correctness:
To prove this algorithm yields an optimal solution, we need first shows that the geedy-choice property holds. The geedy-choice property is that an optimal solution to making change for n cents includes a coin of value v, which is the largest value such that $v \leq n$. Consider an optimal solution, if this optimal solution contains v, then we can finish our proving. If the optimal solution doesn't contain value v, then we need to take four cases into consideration.

- If $1 \leq n < 5$, then we have $v = 1$. Ths solution can only consist of pennies, and so it must contain the greedy choice.
- If $5 \leq n < 10$, then we have $v = 5$. By supposition, the optimal solution doesn't contain a nickel. We can replace five pennies by a nickel, in this case, we can get a solution with four coins fewer.
- If $10 \leq n < 25$, then we have $v = 10$. By supposition, the optimal solution doesn't contain a dime. We can replace pennies and nickels which adds up to 10 cents by a dime, in this case, we can get a solution with fewer coins.
- If $25 \leq n$, then we have $v = 25$. By supposition, the optimal solution doesn't contain a quarter. The solution can only consist of pennies, nickels and dimes. If we have three dimes, we can replace them with a quarter and a nickel. In this case, we can get a solution with one fewer coin. If pennies, nickels and dimes adds up to 25 cents, we can replace them with a quarter. In this way, we can get a solution with fewer coins.

Thus, we can see that there is always an optimal solution that includes the geedy choice. Therefore, we can combine the greedy choice with an optimal solution to the subproblem to produce the optimal solution to the original problem.
Time complexity:
We choose one coin at a time and compute the subproblem recursively, the running time is $\Theta(k)$, k is the number of coins used in an optimal solution. Because $k \leq n$, the running time is $\mathcal{O}(n)$. In each computation, the running time is $\mathcal{O}(1)$. Hence, the running time of this

algorithm is $\mathcal{O}(n)$.

b) Show that the greedy algorithm always yields an optimal solution.

The greedy algorithm for making change for n cents is to find the denomination c^j such that $j = \max\{0 \leq i \leq k : c^i \leq n\}$. And then we compute the subproblem of making change for $n - c^j$. Correctness:

If in the optimal solution, we use a_i coins of denomination c^i to make change for n cents, then $a_i < c$. If $a_i \geq c$, we can replace c c^i with a c^{i+1}, then we can reduce the total number of coins by $c - 1$.

If we don't use a greedy algorithm to solve the problem of making change for n cents, we can get a solution which uses denominations $c^0, c^1, \cdots, c^{j-1}$ and $\sum_{i=0}^{j-1} a_i c^i = n$, also $n \geq c^j$. Because $n \geq c^j$, $\sum_{i=0}^{j-1} a_i c^i \geq c^j$. Let's assume that the non-greedy algorithm is an optimal solution, then we can get $a_i \leq c - 1$ for $i = 0, 1, \cdots, j - 1$. Then we can get the following conclusion:

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

This conclusion contradicts our previous conclusion that $\sum_{i=0}^{j-1} a_i c^i \geq c^j$, hence, the non-greedy strategy is not an optimal solution. In other words, only a greedy algorithm is an optimal solution.

c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution.

We can easily find a set of coin denominations for which the greedy algorithm cannot yield an optimal solution. Let's say we have a set of coin denominations which havs value $\{1, 4, 5\}$ and we have 8 cents. A greedy algorithm will give us the solution $8 = 5+1+1+1$ which uses four coins, while we can easily find out that a non-greedy solution will give us the optimal solution $8 = 4 + 4$ which uses only two coins.

d)Give an $\mathcal{O}(nk)$-time algorithm that makes change for any set of k different coin denomi-

nations, assuming that one of the coins is a penny.

Solution:

We could use dynammic programming strategy to solve this problem.

Table:

$OPT[i]$ = the minimum number of coins used for making change for i cents. $0 \leq i \leq n$

Output for the problem:

The shortest maximum-weight of a common subsequence.

Base Case:

$i = 0, OPT[i] = 0$

Recursion Case:

$OPT[i] = \min\{1 + OPT[i - d_j] : 1 \leq j \leq k\}$

Extra information:

The coin we chose for i cents

Algorithm:

Data: d_j, n
Result: A
$OPT[0] = 0$;
for $i = 1; i \leq n; i++$ **do**
  $OPT[i] = \infty$;
  $choice[i] = 0$;
  for $j = 1; j \leq k; j++$ **do**
    if $j \geq d_j$ *and* $OPT[i] > 1 + OPT[i - d_j]$ **then**
      $OPT[i] = 1 + OPT[i - d_j]$;
      $choice[i] = d_j$;
    end
  end
end
return A ;

Time complexity:

There are two loops in this algorithm, so the running time for this algorithm is $\mathcal{O}(kn)$