CS 325 - Spring 2020

Homework #4

Problem 1. (6 pts)

a) Assume the following symbols *a, b, c, d, e* occur with frequencies 1/2, 1/4, 1/8, 1/16, 1/16 respectively. What is the Huffman encoding of the alphabet? (3 pts)

b) If the encoding is applied to a file consisting of 1 million characters with the same given frequencies, what is the length of the encoded file in bits? (3 pts)

SOLUTION:

a -> 0
b -> 10
c ->110
d -> 1110
e -> 1111

length = $\frac{1000000}{2} \cdot 1 + \frac{1000000}{4} \cdot 2 + \frac{1000000}{8} \cdot 3 + 2 \cdot \frac{1000000}{16} \cdot 4 = 1{,}875{,}000$

Problem 2. (5 pts)

Complete problem 16.2-2 on page 427 in the book

SOLUTION:

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 $1, \ldots, n$. Then $S' = S - \{i\}$ must be an optimal solution for $W - w_i$ pounds and items $1, \ldots, i-1$, and the value of the solution S is v_i plus the value of the subproblem solution S'.

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

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

The last case 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 $1, \ldots, i-1$ 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 $1, \ldots, i-1$ up to the weight limit w, and get $c[i-1, w]$ value. The better of these two choices should be made.

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. (That is, 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)
let c[0..n, 0..W] be a new array
for w = 0 to W
    c[0, w] = 0
for i = 1 to n
    c[i, 0] = 0
    for w = 1 to W
        if w_i ≤ w
            if v_i + c[i − 1, w − w_i] > c[i − 1, w]
                c[i, w] = v_i + c[i − 1, w − w_i]
            else c[i, w] = c[i − 1, w]
        else c[i, w] = c[i − 1, w]
```

We can use the c table to deduce the set of items to take by starting at $c[n, W]$ and tracing where the optimal values came from. If $c[i, w] = c[i-1, w]$, then 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 $\Theta(nW)$ time total:

- $\Theta(nW)$ to fill in the c table: $(n+1) \cdot (W+1)$ entries, each requiring $\Theta(1)$ time to compute.
- $O(n)$ time to trace the solution (since it starts in row n of the table and moves up one row at each step).

Problem 3. (8 pts)

Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

Problem 3.a. (4 points)

a) Suppose that the available coins are in the denominations that are powers of *c, i.e.*, the denominations are $c^0; c^1; \ldots; c^k$ for some integers c > 1 and k _ 1. Show that the greedy algorithm of *picking the largest denomination* first always yields an optimal solution. You are expected to reason about why this approach gives an optimal solution. (*Hint*: Show that for each denomination c^i, the optimal solution must have less than *c* coins.)

Problem 3.b. (4 points)

b) Design an *O(nk)* time algorithm that makes change for any set of *k* different coin denominations, assuming that one of the coins is 3 cents in value.

Solution 1: An algorithm that accepts only denominations that are powers of *c* and implements the greedy approach. This way, all we need is to go through the denomination in a decreasing order and pick as many as the largest denominations we can, subtract the result from *n* and recurse. The complexity of this approach is *O*(*k*).

Example:

```
denomination                    #  Loop to create an array of the different denominations from c and k
Reverse sort (denomination)     #  For the greedy algorithm to pick the largest denomination first
making_change function (n, denomination):
  changed                       #  array to store the coins that are used to make change for n cents
  for i=0 to numbers of denomination        # Make change with the fewest number of coins by picking
                                            # the largest denomination first
    coin ← denomination[i]
    if n > 0                    #  only change if the amount for making change is > 0
      while n >= coin           #  loop to make as many changes for the current largest denomination
        n ← n – coin            #  update the amount
        changed ← coin          #  add the coin that was used to make the change
```

The outer for loop runs n number of times through a set of denominations to use the largest value first and thus has a time complexity of O(n). The inner while loop runs k number of times to make as many changes for a given denomination and thus has a time complexity of O(k). Therefore, the overall time complexity is O(nk).

Solution 2: For the general case, we can reason in terms of the optimal substructure as follows:

Let *count*[*j*] be the minimum number of coins we need to make change for *j* cents. First off, for $j \le 0$ we have *count*[*j*] = 0. Next, if an optimal solution for *count*[*j*] contains a coin from denomination $d_i < j$, then we have *count*[*j*] = 1 + *count*[$j - d_i$]. Thus, we can write the following recursive function that captures the optimal substructure:

If $j \le 0$ then *count*[*j*] = 0. Otherwise (i.e., $j > 1$), *count*[*j*] = 1 + $min_{1 \le i \le k}$\{*count*[$j - d_i$]\}.

Let *denom*[1..*n*] be an array and *denom*[*j*] represent a denomination used in an optimal solution of the make change problem for *j*. Notice that, *count*[*j*] shows how many coins of denomination *denom*[*j*] we have. The sketch of the algorithm would be something similar to the following:

```
MakeChange(k,d,n) { //d1,d2,··· ,dk are the denominations
    for j = 1 to n
        count[j] = ∞
        for i= 1 to k
            if ((j ≥ di) and (1 + count[j − di] < count[j])) then
                    count[j] = 1 + count[j − di]
                    denom[j] = di
    return arrays count[] and denom[];
}
```

Problem 4. (6 pts)

Implementation:
Implement the make change algorithm you designed in the previous problem. Your program should read a text file “data.txt" where each line in “data.txt" contains three values *c, k* and *n*. Please make sure you take your input in the specified order *c, k* and *n*. For example, a line in “data.txt" may look like the following:

    4 3 73

where $c = 4$; $k = 3$; $n = 73$. That is, the set of denominations is $\{4^0; 4^1; 4^2; 4^3\} = \{1; 4; 16; 64\}$, and we would like to make change for n = 73. The file “data.txt” may include multiple lines like above.

The output will be written to a file called “change.txt”, where the output corresponding to each input line contains a few lines. Each line has two numbers, where the first number denotes a denomination and the second number represents the cardinality of that denomination in the solution. For example, for the above input line ‘4 3 73’, the optimal solution is the multiset {64; 4; 4; 1}, and the output in the file “change.txt” is as follows:

    Data input: c = 4, k = 3, n = 73

Denomination: 64 Quantity: 1
Denomination: 16 Quantity: none
Denomination: 4 Quantity: 2
Denomination: 1 Quantity: 1

which means the solution contains one coin of denomination 64, none of 16, two coins of 4, and one coin of 1. You can use a delimiter line to separate the outputs generated for different input lines.

Problem 5 – Extra Credit (4 pts)

a) Using Huffman encoding of n symbols with the frequencies $f_1, f_2, f_3 \ldots f_n$, what is the longest a codeword could possibly be? (2pts)

b) Give at least one example set of frequencies that would produce the case above. (2pts)

SOLUTION:
The longest codeword can be a length of n-1. Encoding of n symbols with of them have probabilities of 1/2, 1/4,….$1/2^{n-2}$. Only 2 of them having a probability $1/2^{n-1}$ to get this value.

Submit a copy of all your code files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will only test execution with an input file named data.txt.