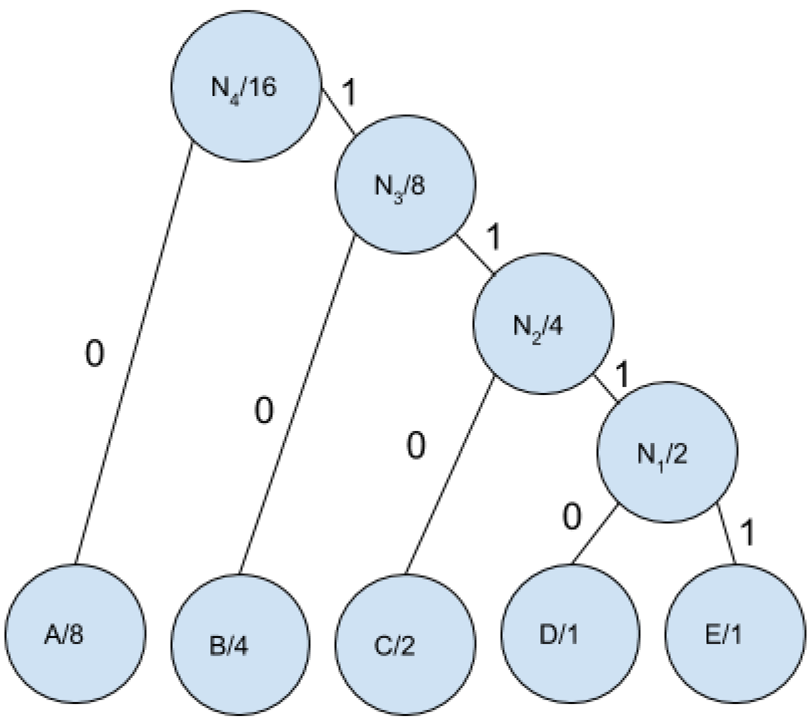
Zuhair Ahmed
CS325_400
Spring 2020

Homework #4

Problem 1. (6pts)
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)

We first list the common denominator of these frequencies which is 16 so the total of frequencies sums to 16 or 100%. A/8, B/4, C/2, D/1, E/1. The Huffman Algorithm states we sum up the two smallest nodes, add back to heap, continue add the next two smallest nodes, again and again until there is only one node remaining (i.e. root node). Finally, if node is to the left we assign it 0 to the branch, and if node to the right we assign it 1 to that branch.



Huffman Code:
A = 0 (1 Bit)
B = 10 (2 Bits)
C = 110 (3 Bits)
D = 1110 (4 Bits)
E = 1111 (4 Bits)

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)

A file consisting of these frequencies at 1 million characters total, implies there 500K A's (8/16), 250K B's (4/16), 125K C's (2/16), 62.5K D's (1/16), and 62.5K E's (1/16). We then multiply each of these by bits of frequencies from Huffman Code in part A above.

Letter	**Frequency (out of 1 million)**	**Hoffman Code**	**Hoffman Code Bit Length**	**Frequency x Bit Length**
A	500K	0	1	500K
B	250K	10	2	500K
C	125K	110	3	375K
D	62.5K	1110	4	250K
E	62.5K	1111	4	250K
TOTAL				**1,875K**

Hence, ignoring bit code length of summary table/tree for conversion, the total length of the encoded file in bits is **1,875K bits.**

Problem 2. (5pts)
Complete problem 16.2-2 on page 427 in the book:
Give a dynamic-programming solution to the 0-1 knapsack problem that runs in O(n W) time, where n is the number of items and W is the maximum weight of items that a thief can put in their knapsack.

```
int** 01_knapSack(n, W)

{

	int table [n + 1] [W +1];

	//initialize table for sections x,0 and 0,y to 0 by looping around each axis

	for (int x = 1; x < n; x++)

		table[x, 0] = 0;

	for (int y = 1; y < W;y++)

		table[0, y] = 0;

	//double for loop to to go through all elements of knapSack

	for (int i = 1; i <n; i++)

	{

		for (int j = 1; j <W; j++)

		{

			if (j < i.weight) //"dynamic" portion as it checks if element already exists

				table[i, j] = table[i - 1, j];

			else //if not exists need to calc max of element weight that can be carried

				table[i, j] = max(table[i - 1, j], table[i - 1, j - i.weight] + i.value);

		}

	}

	return table;

}
```

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; …; 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.)

The optimal approach here is the one that generates the fewest coins while using a greedy algorithm. So for example, if c = 2 and k = 3, we have $\{2^0, 2^1, 2^2, 2^3\}$ or {1, 2, 4, 8}. So if we wanted to find the fewest coins for 15 cents, we start with the largest coin here which is 8 which leaves 7 cents (15 - 8 = 7). We then use 1 coin of 4 which leaves 3 cents (7 - 4 = 3 cents). Lastly, we then use one coin of 2 and one coin of 1. Hence the final solution is 4 coins: 8 cent coin + 4 cent coin + 2 cent coin + 1 cent coin. This is a greedy approach, but also the most globally optimal approach as there is no way to group together any of these combinations to still result in 15 cents total using less than 4 coins.

To then generalize this approach for all c > 1 and k ≥ 1, we first show that the number of coins of any denomination c^i (excet c^k) used must be less than c, where i < k. This can be shown by assuming that a set $\{x_0, x_1, x_2...x_k\}$ did exist that was the optimal solution where x_i represents the number of coins in demonimination c^i. From here, we can assume that some j, such that $x_j \geq c$ we can replace c number of c_j denomination coins by c_{j+1}. Hence, x_j decreases by c and increases by x_{j+1} by 1 which implies the number of coins used is also decreased by c-1. This is a contradiction since our initial assumption was that the set $\{x_0, x_1, x_2...x_k\}$ was the optimal solution with fewest possible coins. This shows that the optimal solution must have $x_j < c$ for any denomination c^j except for c^k.

This only leaves one possible solution that then means this condition: picking $x_k = \text{floor}(n/c^k)$ and for j < k where $x_j = \text{floor}((n * \bmod c^{j+1}) / c^j)$.

∴ The Greedy Algorithm always produces an optimal solution for denominations c^0, c^1, c^2, $c^3 ... c^k$.

Problem 3.b. (4 points)
b) Design an O(n*k) time algorithm that makes change for any set of k different coin denominations, assuming that <u>one </u>of the coins is 3 cents in value.

We can initially solve this problem recursively. Assuming c_i be the minimum number of coins required to make change for i cents and $\{x_0, x_1, x_2,...x_k\}$ being the denominations. This can be computed: c_i = 0 if i = 0, else 3 + min(c[i-d_j]) if i > 0. However this results in $O(n^2*k^2)$ and we can do better with a dynamic programming approach should yield O(n*k). One algorithm to solve this dynamically could be:

```
int* dynamicMakeChange(int* d, int n, int k)
{
	const inst MAX = 999;
	int coins [MAX];
	for(int i = 1; i < n; i++)
	{
		coins [i] = i;
		for (int j = 1; j < k, j++)
		{
			if (i >= d_j)
			{
				if  (1 + c[i-d_j] < c[i]) //this is “dynamic” part, checks if value exists
					c[i] = 1 + c[i-d_j];
				else //else has to calculate value of d_j each time
					c[i] = d_j;
			}
		}
		return coins;
	}
}
```

With this approach, outer loop does runs n times and inner loop runs k times. Hence total run time is **O(n*k).**

Problem 4. (6 pts)
Submitted to TEACH + CANVAS. Implementation below:

```
/*******************************************************
 * Author: Zuhair Ahmed (ahmedz@oregonstate.edu)
 * Date Created: 4/26/2020
 * Filename: makeChange.cpp
 * Overview: This program is a modified version of the make change problem. Goal
 *          is to find minimum denomination of coins for change while using a
 *          greedy algorithm approach.
 * Input: "data.txt" which consists of input values c (base), k (max exponent),
 *       and n (change to make in cents)
 * Output: "change.txt"
 *******************************************************/

#include<fstream>
#include<iostream>
#include<cstdio>
#include<cmath>

int main()
{
   const int MAX = 9999;
   int array[MAX]; // array to store coin denominations

   //create input/output file variables + check error if files cannot open
   std::ofstream outputFile;
   std::ifstream inputFile;
   inputFile.open("data.txt");
   if (!inputFile.is_open())
   {
      std::cout << "ERROR: FILE CANNOT OPEN" << std::endl; //output to console
      return 1;
   }
   outputFile.open("change.txt");
   if (!outputFile.is_open())
   {
      std::cout << "ERROR: FILE CANNOT OPEN" << std::endl; //output to console
      return 1;
   }
```

```
    while (!inputFile.eof())
    {
        int c; //base denomination of coins
        int k; //max exponent to raise base to
        int n; //total in cents to then break into change
        int count; //count of each coin denomination required

        //store values from file into associated variables
        inputFile >> c;
        inputFile >> k;
        inputFile >> n;

        //calc each value of denomination and store into array
        for(int x = 0; x <= k; x++)
            array[x] = pow(c, x);

        //print summary of inputs from data.txt file
        outputFile << "Data input: c = " << c << ", k = " << k << ", n = " << n << std::endl;

        for(int y = k; y >= 0; y--)
        {
            count = n / array[y];
            n = n % array[y]; //adjusting value of n for remaining coin denominations
            if(count != 0)
                outputFile << "Denomination: " << array[y] << " Quantity: " << count << std::endl;
            else
                outputFile << "Denomination: " << array[y] << " Quantity: " << "none" << std::endl;
        }

        outputFile << std::endl << std::endl;
    }

    //close input/output files and exit main
    inputFile.close();
    outputFile.close();
    return 0;
}
```

Problem 5 – Extra Credit (4 pts)

a) Using Huffman encoding of n symbols with the frequencies f1, f2, f3… fn, what is the longest a codeword could possibly be? (2pts)

The longest codeword can be of length n-1 bits as shown with problem 1a above in this homework assignment. An encoding of n symbols with n-2 of them having probabilities 1/2, 1/4, …, $1/2^{n-2}$ and two of them having probability $1/2^{n-1}$ achieves this value. Therefore the longest codeword is n-1 bits.

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

An example could be when n = 5, this produces a Huffman Tree shown below. Here A is 1 bit ("0"), B is 2 bits ("10"), C is 3 bits ("110"), D is 4 bits ("1110"), and E is 4 bits ("1111"). This creates the longest codeword of 5 -1 (or n-1) of length 4 bits total.