Zuhair Ahmed
CS325_400
Spring 2020

Homework #3

Problem 1: *(3 points)* **Rod Cutting: (from the text CLRS) 15.1-2**
Counter example -

Length *i*	1	2	3	4	5	6
Price p_i	$1	$20	$21	$22	$23	$25
Density p_i/i	1	20	21	22	23	1

For a rod of length 6, the optimal solution is clearly to cut into 3 pieces of 2 for a total of $60 (3 x $20). However, according to the greedy algorithm we would cut 1 off leaving 2 pieces of length 1 and 5 for a total of $24 ($1 + $23). This is because this greedy algorithm seeks to maximize density and not total Price hence a dynamic programming approach would be better to correct for this error.

Problem 2: *(3 points)* **Modified Rod Cutting: (from the text CLRS) 15.1-3**
Modification of rod cutting problem to include Cost of cutting (each at a fixed cost) -

```
int rodCuttingwCost(int* price, int number, int cost)
{
	int MAXLENGTH = 999;
	int array[MAXLENGTH];
	Int q = 0;
	int array[0] = 0; //set initial element $0 otherwise value generated without cutting
	for (int j = 1; j < number; j++)
	{
		q = price[j]; //adjust for case where we have to make no cuts
		for (int i = 1; i < j-1; j++) //adjust to j-1 for case to make no cuts
			q = max(q, price[i] + array[j - i] - cost);  //adjust to substract cost per cut
		array[j] = q;
	}
	return array[number];
}
```

Problem 3: *(6 points)* **Making Change:**
a) Describe and give pseudocode for a dynamic programming algorithm to find the minimum number of coins needed to make change for A.

We will split this problem into 2 helper functions, minimCoins and makeChange. minimCoins will seek to take a dynamic programming approach to find the fewest coins possible from an inputted array of denominations (i.e. v1, v2, v3, etc.). This will be done with two for loops, the first cycling through elements from 2 to A, and the second from 1 to the inputted number of coins. If programs reaches second For loop, we will first check if a solution already exists in the solution array (i.e. memoization/tops down approach) and if it doesn't exist then we will calculate and input into the solution array. Lastly, makeChange will be simple while loop to output the solution array in the fewest coins possible.

```
int minimCoins(int* V, int number, int A)
{
	int solutionArray[number];
	int coins[number];
	int min = 0;
	int index = 0;
	coins[0] = 0; //set first element to 0 and second to 1 to ensure solution is generated
	coins[1] = 1;
	for (int x = 2, x < A; j++) //cycle through all values up to A
	{
		for (int y = 1; i < number; i++)
		{
			if (x >= V[y]) //check to ensure still in bounds
			{
				if (coins[x-V[y]] < min ) //check if value is less than previous min
				{
					index = y;
					min = coins[x-V[y]]; //input values into new min
				}
			}
		}
		solutionArray[x] = index;
		coins[x] = min + 1; //move over to next element in coins array
	}
	return solutionArray[A]];
}
```

```
void makeChange(int* V, int* C, int minCoins, int* solutionArray)
{
        while (minCoins > 0) //check to ensure we don't go out of bounds
        {
                C[solutionArray[minCoins]] = C[solutionArray[minCoins]] + 1;
                printf(V[solutionArray[minCoins]]);  //output to console
                minCoins = minCoins – V[solutionArray[n]]; //adjust for new minimum amount
        }
}
```

b) What is the theoretical running time of your algorithm?
minimCoins has run time of Θ(A * number) because of the two For loops (to A and number, respectfully)
makeChange has run time of O(A) since it is upper bounded dependent on length of A
Hence, at large values of number and A, the overall running time is Θ(A * number)

Problem 4: *(18 points)* **Shopping Spree:**
a) A verbal description and give pseudo-code for your algorithm. Try to create an algorithm that is efficient in both time and storage requirements.

Shopping Spree appears to be very similar to Knapsack problem described in textbook. Goal is to maximize profit by finding items for each family member that do not exceed their carrying weight per person but at same time maximizes profit for total family. This can be done at a high level summary / pseudo-code by first reading T test cases from input file "shopping.txt". This will be main loop sequence in main function. For each iteration we then read the number of item N, number of family member F, and then calc the maximimum price of items each can cary at their max using the knapSack helpfer function. We also need to keep track of max price which we will need to aggregate at end. Lastly, we need to output all these items in specified format to "results.txt" file.

b) What is the theoretical running time of your algorithm for one test case given N items, a family of size F, and family members who can carry at most M_i pounds for $1 \le i \le F$.

$O(NM_1 + NM_2 + \ldots + NM_F) = O(N.\sum_{i=1}^{F} M_i)$

c) Implement your algorithm by writing a program named “shopping” (in C, C++ or Python) that compiles and runs on the OSU engineering servers. The program should satisfy the specifications below.

Uploaded below to CANVAS and TEACH

```
/*******************************
 * Author: Zuhair Ahmed (ahmedz@oregonstate.edu)
 * Date Created: 4/19/2020
 * Filename: shoppingSpree.cpp
 * Overview: This program is a modified version of Knapsack problem. Goal
 *           is to find which items each person can carry such that family
 *           overall can maximize their total profit.
 * Input: "shopping.txt" which consts of T test cases
 * Output: "results.txt"
 * ****************************/

#include<fstream>
#include<iostream>

//Max and knapSack functions based on pseudo-code from
//Introduction to Algorithms, 3rd Ed. (pages 366-370)
//by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein
int max(int x, int y)
// finds max of 2 ints and returns largest
{

    if (x > y)
        return x;
    else
        return y;
}

int knapsack(int W[], int P[], int N, int M)
// finds max price of items carried by a person each than carries max weight M
{
    int K[N + 1][M + 1]; //create double array of max items and weight
    for (int a = 0; a <= N; ++a)
    {
        for (int b = 0; b <= M; ++b)
        {
            if ((a == 0) || (b == 0)) //case 1 - both are empty
            {
```

```
                K[a][b] = 0;
            }
            else if (W[a - 1] <= b) //case 2 - b greater
            {
                K[a][b] = max(P[a - 1] + K[a - 1][b - W[a - 1]], K[a - 1][b]);
            }
            else //case 3 - a greater
            {
                K[a][b] = K[a - 1][b];
            }
        }
    }

    return K[N][M];  //maximum item prices carried by person
}

int main()
{
   // item weights
    int W[100];
    // item prices
    int P[100];
     // # items
    int N;
    // # test cases
    int T;
   // max weight carried
    int M = 0;
    // # people per family
    int F;

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

```
        std::cout << "ERROR: FILE CANNOT OPEN" << std::endl; //output to console
        return 1;
    }

    inputFile >> T; //input # of test cases from file into T

    for (int counter1 = 0; counter1 < T; ++counter1) //main sequence per test case
    {
        inputFile >> N; //# of items

        for (int counter2 = 0; counter2 < N; ++counter2)
        {
            inputFile >> P[counter2]; //read price
            inputFile >> W[counter2]; //read weight
        }
        int maxTPrice = 0;

        inputFile >> F; //# of Family members

      //Leverage knapSack helper function to calc max items each member can carry
        for (int counter3 = 0; counter3 < F; counter3++)
        {
            inputFile >> M; //max weight per member
            maxTPrice = maxTPrice + knapsack(W, P, N, M); //calc max total price
        }

        // Write output results to file
        outputFile << "Test Case " << counter1+1 << std::endl;
        outputFile << "Total Price " << maxTPrice << std::endl;
        outputFile << "Member Items " << std::endl;
        for (int i = 1; i < F+1; ++i)
          outputFile << "    " << i <<": " << knapsack(W, P, N, M) << std::endl;
    }

    outputFile.close();
    inputFile.close();
    return 0;
}
```