Christopher Merrill

CS325

Homework 3

**Problem 1**

**15.1.2 Show, by means of a counterexample, that the following “greedy” strategy does not always determine an optimal way to cut rods. Define the density of a rod of length I to be pi/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length I, where 1 ≤ i ≤ n, having maximum density. It then continues applying the greedy strategy to the remaining piece of length n – i.**

Consider for example the table from figure 15.1 in the book. We know from the solution on page 361 that the optimal solution for cutting a rod of length 4 is to cut it into two lengths of 2 (for a total value of 10).

If we instead use the greedy strategy above, we would get the following:

For an uncut rod we get: pi/i = 9/4 = 2.25

Cutting a piece n = 3 yields: pi/i = 8/3 = 2.66

Cutting a piece n = 2 yields: pi/i = 5/3 = 2.5

Cutting a piece n = 1 yields: pi/i = 1/1 = 1

Therefore, an initial cut of n = 3 has the highest density. That would leave us with two pieces, one of length 3 and the other of length 1, for a total value of:

8 + 1 = 9

However, we already know that we can reach a value of 10 with two two-inch rods. Thus, the greedy strategy does not hold.

**Problem 2**

**15.1.3 Consider a modification of the rod-cutting problem in which, in addition to a price pi for each rod, each cut incurs a fixed cost of *c*. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem:**

Let r[0…n] and s[1…n] be new arrays and c be the cost to make a cut:

r[0] = 0

FOR j = 1 to n

q = -∞

FOR i = 1 to j

IF j != n

cutCount = 1 + i

ELSE IF i = j

cutCount = 0

ELSE

cutCount = i

IF q < p[i] + r[j-1] - (cutCount \* c)

q = p[i] + r[j-1] - (cutCount \* c)

s[j] = i

r[j] = q

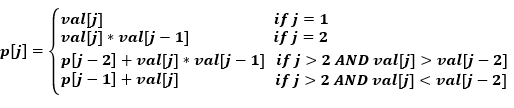
RETURN r and s

Above is a modified version of the BOTTOM-UP-CUT-ROD algorithm provided on page 369 of the textbook (additions are in red). It determines the number of cuts performed and stores them in a variable called cutCount. The max value is then compared to the value of the rod segments minus the price of the cuts.

**Problem 3**

**a)** 27 = 2 + 1 +(3 x 5) + 1 + (4 x 2)

**b)** Let p[j] be the product sum of an arbitrary list of j integers. The dynamic programming optimization formula for computing the product sum of the first j elements is as followed. (Note: j is an index, val[j] is the value of the integer at index j, and p[j] is the product sum of all integers up to and including j):



Note: p[0] = 0

**c)** Only one comparison is made for each value in the sequence (whether or not the value at index j is greater than, less than, or equal to the value at index j -2). That means for a bottom up approach (finding the product sum of all sequences from p[1] to p[j]), we would only have to loop through the sequence one time and the runtime should be T(n) = n. This yields an asymptotic running time of ϴ(n).

**Problem 4**

**a)** I went with a bottom-up approach to this change making problem.The basic idea being to determine all of the solutions from 1 to A, where A is the value you want to make change with. That way, each of the smaller values have been calculated and memoized when you reach the target value of A. The algorithm will have to keep track of the solutions for smaller coins in a results table, as well as the coins used to reach those results.

**Pseudocode:**

A = value of the coin to make change with

n = number of different coin denominations

coins = array of coins of various denominations

results = array that holds the memoized values of sub-problem results

coinsUsed = 2D array that holds which coins were used for each sub-problem result

makeChange(coins[0…n -1], n, A)

FOR each value of coin i, up to and including A

FOR each coin j smaller than i up to n

IF the value of j is less than i

subProblemResult = result[i – coins[j]]

IF the subproblemResult + 1 is less than result[i]

//Update result i

result[i] = subProblemResult

FOR each value up to n

//Copy existing coin counts

coinsUsed[i][k] = coinsUsed[i – coins[j][k]

INCREMENT coinsUsed[i][j]

b) The algorithm above consists primarily of three nested for-loops. The first loops from 0 to A, the second loop from 0 to n and the third also loops from 0 to n. Taken at face value, we should therefore have a run time of:

T(n) = A \* n \* n = An2

However the last loop only runs if a particular condition is met, so this shouldn’t hold weight for large n. Therefore the theoretical running time should instead be:

T(n) = A \* n = An

Which accounts for the main two loops, ignoring the inner most loop and any constant computations.

This would mean give an asymptotic runtime of ϴ(An).

**Problem 6**

**a)** To collect running times for each plot, I started a timer right before each run of my makeChange algorithm and stopped it right after. To generate n values, I would start with n = 1 and then randomly add a value between 1 and 10 to create the next value.

For the n vs. t plot, I held A constant at A = 330,000. I started n at n = 100 and then doubled n at each iteration. I collected a total of 8 data points.

For the A vs. t plot, I held n constant at n = 100. I started A at A = 10,000 and then doubled A at each iteration. I collected a total of 8 data points.

For the nA vs. t plot, I started n at n = 100 and A at A = 10,000 and then doubled both at each iteration. I collected a total of 8 data points.

The following is a sample of the code I used to collect runtime data:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int main() {

vector<int> coins; //Holds the array of coins

int n = 100; //The number of coins

int A = 330000; //The value that we are making change for

clock\_t start; //Start time

double runtime; //Run time

int i, j, k, subResult, luck, val, prevN, count = 0;

//Seed the random number generator

srand(time(NULL));

//Perform 8 rounds of makeChange, doubling n each time

while (count < 8) {

//First value should be 10

val = 1;

prevN = 1;

//Add n coins to the vector

for (i = 0; i < n; i++) {

coins.push\_back(val);

val++;

//Generate a random value from 1 to 10 to add to the current coin value

luck = rand() % 11;

val = prevN + luck;

}

//Start the clock

start = clock();

int results[A + 1]; //Stores the min number of coins required for each value up to A+1

static int coinsUsed[A+1][n]; //Stores the number of each denomination of coin used to reach the result

//Base case

results[0] = 0;

//Initialize all results values to be infinite

for (i = 1; i <= A; i++) {

results[i] = INT\_MAX;

}

//Initialize all coinsUsed values to be 0

for (i = 0; i <= val; i++) {

for (j = 0; j < n; j++) {

coinsUsed[i][j] = 0;

}

}

//Determine the min coins required for all values from 1 to A

for (i = 1; i <= A; i++) {

//Loop through all coins smaller than i

for (j = 0; j < n; j++) {

//Check if the coin value is less than A

if (coins[j] <= i) {

//subResult will hold the result of the current value minus the denomination of coins[j]

subResult = results[i-coins[j]];

//If subResult is not infinity, and provides a result with less coins than the current

//result, add it to the results table

if (subResult != INT\_MAX && subResult + 1 < results[i]) {

//Update results table

results[i] = subResult + 1;

//Copy existing coin counts from i - coins[j]

for (k = 0; k < n; k++) {

coinsUsed[i][k] = coinsUsed[i - coins[j]][k];

}

//Increment the coinsUsed array for the current coin j

coinsUsed[i][j]++;

}

}

}

}

//Calculate the runtime

runtime = (clock() - start) / (double) CLOCKS\_PER\_SEC;

cout << " RUNTIME FOR " << n << " COINS AND A-VALUE " << A << ": " << runtime << endl;

//Clear coins contents

coins.clear();

//Increment count and double n-value

count++;

n \*= 2;

}

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**b)**

All three graphs have linear behavior, which agrees with the theoretical result determined in problem 4. (ϴ(nA)).

When A is held constant, the asymptotic runtime determined in problem 4 would be ϴ(n).

Likewise when n is held constant, the asymptotic runtime determined in problem 4 would be ϴ(A).