Gretel Rajamoney

1/24/2021

Homework #3

Problem 1:

a. Recursive:
In order to solve this problem recursively we start by repeatedly looping through all of the numbers until you find the biggest possible total that is the lowest weight but the highest price. The knapsack() function calls itself recursively until the highest value is found. The values are repeatedly stored into a variable and compared while still excluding the item of large capacity, and storing that in a new variable. Lastly, the two saved variables are compared and the greater value between the two of them is returned to the user.

```
recursiveKnapsack(size, value array, weight array, weight constant)
{
        if size or weight is equal to 0
        {
                print 0
        }
        if value in the last element of weight array is greater than weight constant
        {
                recursively call knapsack, this time with size – 1
        }
        if none of the above qualify
        {
                recursively call knapsack and return the bigger value
        }
}
```

Dynamic:

In order to solve this problem using dynamic programming, we start by creating a two-dimensional array that stores the size as well as the weight constant. We then implement the same logic for the knapsack() function as we did in the recursive method, but this time we will implement checks to see whether we have found the most optimal result through reading and comparing all of the elements indexed in the array by parsing through using a for loop. We continue calculating the value of the bag repeatedly until we have reading the maximum possible value. Lastly, we return this maximum value.

```
dynamicKnapsack(size, value array, weight array, weight constant)
{
        create 2D array of dimensions size by constant weight

        from (0 to n)

        {

                repeatedly parse through array finding the largest value

        }

        return array indexes of the largest value

}
```

b. Both algorithms have been implemented into the same program named knapsack.cpp and submitted in a zipped file uploaded onto TEACH. The recursive algorithm was named recursiveKnapsack and the dynamically programmed algorithm was named dynamicKnapsack.

c. Dynamic Programming:

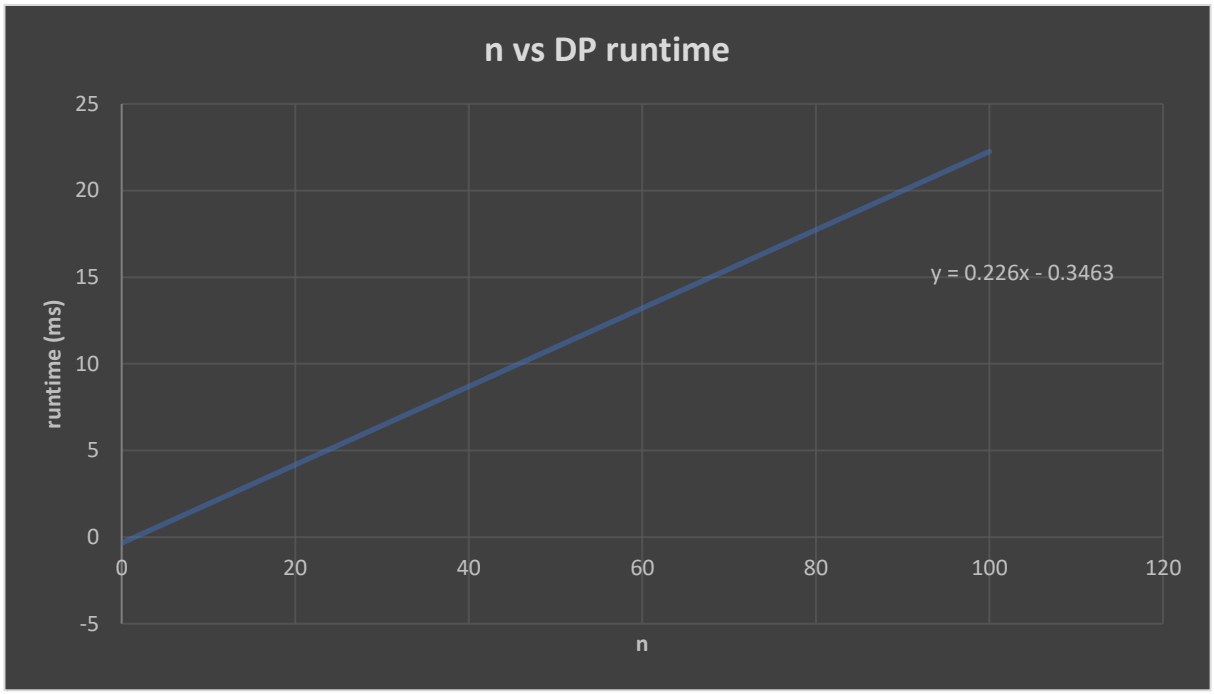
The following data was calculated to record the dynamic programming runtime, the data was calculated in milliseconds. The constant weight was also modified to 10,000 in order create values large enough to generate dynamic programming runtimes. The recursive call was commented out due to such a large constant weight.

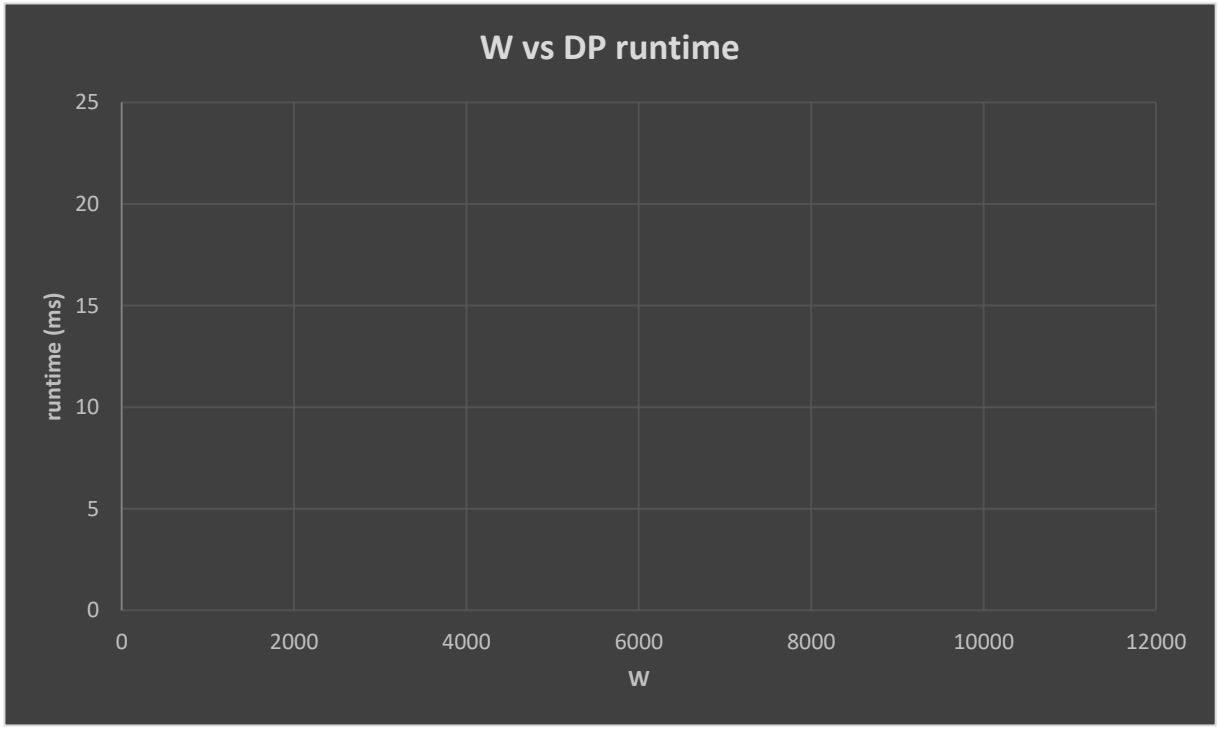
```
n = 0   W = 10000   Rec. Time = 0   DP Time = 0   Rec. Max = 32764   DP Max = 0
n = 5   W = 10000   Rec. Time = 0   DP Time = 0   Rec. Max = 32764   DP Max = 188
n = 10   W = 10000   Rec. Time = 0   DP Time = 0   Rec. Max = 32764   DP Max = 216
n = 15   W = 10000   Rec. Time = 0   DP Time = 0   Rec. Max = 32764   DP Max = 389
n = 20   W = 10000   Rec. Time = 0   DP Time = 10   Rec. Max = 32764   DP Max = 556
n = 25   W = 10000   Rec. Time = 0   DP Time = 0   Rec. Max = 32764   DP Max = 632
n = 30   W = 10000   Rec. Time = 0   DP Time = 10   Rec. Max = 32764   DP Max = 843
n = 35   W = 10000   Rec. Time = 0   DP Time = 10   Rec. Max = 32764   DP Max = 830
n = 40   W = 10000   Rec. Time = 0   DP Time = 10   Rec. Max = 32764   DP Max = 1184
n = 45   W = 10000   Rec. Time = 0   DP Time = 10   Rec. Max = 32764   DP Max = 966
n = 50   W = 10000   Rec. Time = 0   DP Time = 10   Rec. Max = 32764   DP Max = 1321
n = 55   W = 10000   Rec. Time = 0   DP Time = 10   Rec. Max = 32764   DP Max = 1323
n = 60   W = 10000   Rec. Time = 0   DP Time = 10   Rec. Max = 32764   DP Max = 1524
n = 65   W = 10000   Rec. Time = 0   DP Time = 20   Rec. Max = 32764   DP Max = 1595
n = 70   W = 10000   Rec. Time = 0   DP Time = 10   Rec. Max = 32764   DP Max = 1668
n = 75   W = 10000   Rec. Time = 0   DP Time = 20   Rec. Max = 32764   DP Max = 1858
n = 80   W = 10000   Rec. Time = 0   DP Time = 20   Rec. Max = 32764   DP Max = 1994
n = 85   W = 10000   Rec. Time = 0   DP Time = 20   Rec. Max = 32764   DP Max = 2322
n = 90   W = 10000   Rec. Time = 0   DP Time = 20   Rec. Max = 32764   DP Max = 2339
n = 95   W = 10000   Rec. Time = 0   DP Time = 20   Rec. Max = 32764   DP Max = 2104
n = 100   W = 10000   Rec. Time = 0   DP Time = 20   Rec. Max = 32764   DP Max = 2529
```

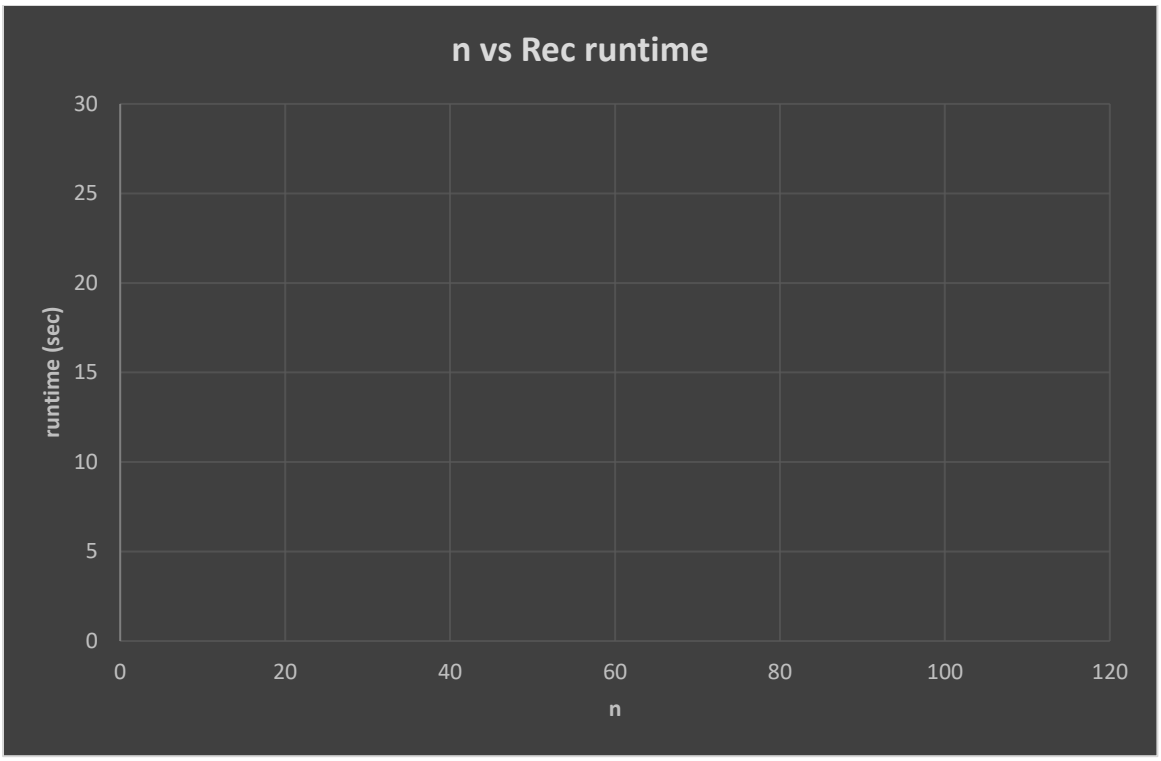
Recursive Data:

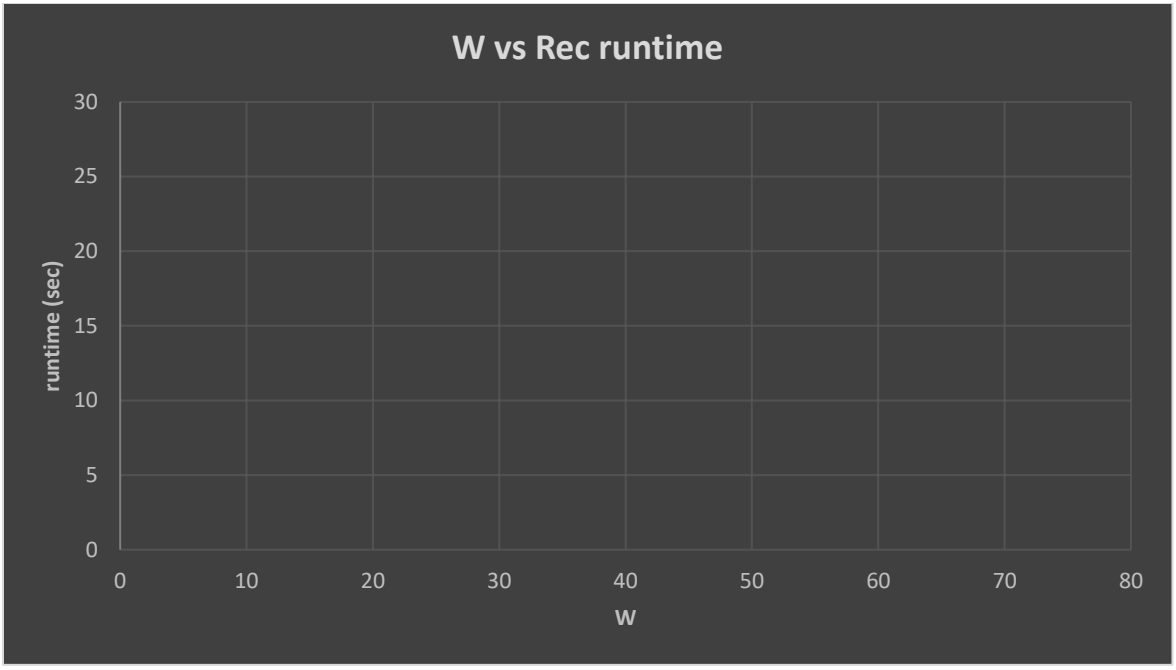
The following data was calculated to record the recursively programmed runtime, the data was calculated in seconds. The constant weight was set to 75 in order to be small enough to generate lower runtimes, but resulting in 0 for the dynamic runtimes.

```
n = 0   W = 75   Rec. Time = 0   DP Time = 0   Rec. Max = 0   DP Max = 0
n = 5   W = 75   Rec. Time = 0   DP Time = 0   Rec. Max = 119   DP Max = 119
n = 10   W = 75   Rec. Time = 0   DP Time = 0   Rec. Max = 112   DP Max = 112
n = 15   W = 75   Rec. Time = 0   DP Time = 0   Rec. Max = 188   DP Max = 188
n = 20   W = 75   Rec. Time = 0   DP Time = 0   Rec. Max = 263   DP Max = 263
n = 25   W = 75   Rec. Time = 0   DP Time = 0   Rec. Max = 264   DP Max = 264
n = 30   W = 75   Rec. Time = 0   DP Time = 0   Rec. Max = 403   DP Max = 403
n = 35   W = 75   Rec. Time = 0.01   DP Time = 0   Rec. Max = 288   DP Max = 288
n = 40   W = 75   Rec. Time = 0.05   DP Time = 0   Rec. Max = 351   DP Max = 351
n = 45   W = 75   Rec. Time = 0.04   DP Time = 0   Rec. Max = 365   DP Max = 365
n = 50   W = 75   Rec. Time = 0.02   DP Time = 0   Rec. Max = 316   DP Max = 316
n = 55   W = 75   Rec. Time = 0.1   DP Time = 0   Rec. Max = 350   DP Max = 350
n = 60   W = 75   Rec. Time = 0.37   DP Time = 0   Rec. Max = 373   DP Max = 373
n = 65   W = 75   Rec. Time = 0.49   DP Time = 0   Rec. Max = 361   DP Max = 361
n = 70   W = 75   Rec. Time = 0.97   DP Time = 0   Rec. Max = 402   DP Max = 402
n = 75   W = 75   Rec. Time = 2.09   DP Time = 0   Rec. Max = 464   DP Max = 464
n = 80   W = 75   Rec. Time = 3.41   DP Time = 0   Rec. Max = 469   DP Max = 469
n = 85   W = 75   Rec. Time = 24.17   DP Time = 0   Rec. Max = 613   DP Max = 613
n = 90   W = 75   Rec. Time = 4.21   DP Time = 0   Rec. Max = 417   DP Max = 417
n = 95   W = 75   Rec. Time = 15.4   DP Time = 0   Rec. Max = 387   DP Max = 387
n = 100   W = 75   Rec. Time = 17.78   DP Time = 0   Rec. Max = 574   DP Max = 574
```


n vs DP runtime
25
20
15
10
5
0
-5
runtime (ms)
0
20
40
60
80
100
120
n
y = 0.226x - 0.3463


W vs DP runtime
25
20
15
10
5
0
runtime (ms)
0
2000
4000
6000
8000
10000
12000
W





d. Since the recursive algorithm was extremely slow, I had to calculate my runtimes by lowering my constant value for W all the way down to 75, meanwhile for my dynamic programming runtimes I had to increase it all the way up to 10,000 in order to see times even when it was set to calculate time in milliseconds. Increasing W increases the runtime for both the recursive and the dynamically programmed algorithms.

Problem 2:

a. In order to create an algorithm that is efficient we must incorporate dynamic programming and recursive methodology. We start by creating an array to hold all the objects, as well as an array to hold all of the family members carrying capacities. Next, we must find which set of items each family member will be responsible for carrying, as well as ensure that it is within the bounds of their carrying capacity. In order to do this, we repeatedly compare each item with each family members carrying capacity, once we find a set of items (only one of each item is allowed), we must total up the weight of these items and ensure that it is less than or equal to the carrying capacity of the family member responsible for carrying it. In order to prints out the items each family member carried we must create a vector to store each value to be printed later.

```
dynamicKnapsack(total number of objects, array of object prices, array of object
weights, carrying capacity of each family member, vector of objects carried)
{
        create 2D array of dimensions size by constant weight
        from (0 to n)
        {
                repeatedly parse through array finding the highest total price
        }
        return array indexes of the highest possible price
        stores objects each family member carried into the vector
}
int main()
{
        opens shopping.txt file and stores data into variables
        opens results.txt file to store calculates values into
        for (every family member)
        {
                finds objects to carry and repeatedly adds prices
                returns the total price carried by family
        }
```

for (every family member)

{

prints out what objects they carried

}

closes shopping.txt file

closes results.txt file.

}

b. Because we are approaching this problem using dynamic programming, the running time of this algorithm can be represented by $O(NM)$. The variable N represents the number of items from the shopping list. The variable M represents the carrying capacity of each family member. The variable F represents the total number of family members. Therefore, the theoretical run time of this algorithm is:
$O(NM_1 + NM_2 + \cdots NM_F)$ =
$O\left(N \cdot \sum_{i=1}^{F} M_i\right)$

c. Algorithm has been implemented into a program named shopping.cpp and submitted in a zipped file uploaded onto TEACH.