Swetha Jayapathy

Student ID : 934041047

CS 325 - Homework Assignment 3

Problem 1: (2 points) Rod Cutting: (from the text CLRS) 15.1-2

Lets consider a rod of length 8

Length i	1	2	3	4	5	6	7	8
Price Pi	1	4	6	8	10	12	14	16
Density(pi/i)	1	2	2	2	2	2	2	2

Greedy:

Let the given rod length be 8

If we follow the greedy approach, we would first cut rod of length 7 and then would take rod of length 1 ,which gives the price as 15.

Length : 7+1

Price value : 14+ 1 = 15

Whereas, we can get the maximum price with other combination as follows :

Length : 6+2

Price value : 12+ 4 = 16

If we take the length 6 and the length 2, we get the price as 16 which is greater than the above. Hence this states as a counter example that greedy does not always gives an optimal solution.

Problem 2: (3 points) Modified Rod Cutting: (from the text CLRS) 15.1-3

In the modified version since the costs comes into the picture, we need to minus the cost from the sum of prices of the all the pieces. Therefore the change is in the body of for loop where q takes the value q=max(q,p[i]+r[j-i]-c)

Algorithm :

```
ModifiedCutRod(p,n,c) {
        Let r[n] be a new array
        r[0] = 0
        for(j=1 to n) {
```

```
            q=p[j]
            for(i=1 to j-1)
            {
                    q=max(q,p[i]+r[j-i]-c)
            }
    r[j] = q
    }
    return r[n] }
```

Problem 3: (10 points) 0-1 Knapsack: Recursive vs DP

Given weights and values of n items, select items to place in a knapsack of capacity W to maximize the total value in the knapsack. That is, given two integer arrays val[] and wt[ ] which represent values and weights associated with n items respectively and an integer W which represents knapsack capacity, determine the maximum value subset of val[] such that sum of the weights of this subset is ≤ W. Items cannot be broken or used more than once, you either select the complete item, or don't select it. Implement both a recursive and dynamic programming algorithm to solve the 0-1 knapsack problem. Both algorithms should return the maximum total value of items that can fit in the knapsack.

a) Give a verbal description and detailed pseudo-code for each algorithm.

Recursive Algorithm for Knapsack :

```
Function knapsack(w[], v[], n, Capacity) {
    If (n <=0)
            return 0;
    else if (w[n-1] > W )
            return knapsack(w, v, n-1, Capacity)
    else
        return max(knapsack(w, v, n-1, Capacity),  v[n-1] + knapsack(w, v, n-1, Capacity – w[n-1]))
}
```

The above code takes the following inputs :

- w[] – array of weights
- v[] – array of value corresponding to the weights.
- n- Total number of Items
- Capacity – The total Capacity which can be accommodated.

According to this recursive algorithm, if the current weight is less than the remaining capacity, it evaluates both the options recursively ; one including the current weight and another without considering the current weight. It calculates the maximum of both the paths and returns the maximum.

This recursive algorithm gives the maximum of 2 values, that is either the optimal solution for n-1 items or the optimal solution including the n^{th} item.

Hence the running time for this is $O(2^n)$

Dynamic Programming Algorithm for Knapsack :

```
Function knapsack(weight[], value[], n, Capacity) {

        If(n< = 0 or Capacity <= 0 )
            Return 0;
        for(j = 0 to Capacity)
            m[0] [j] = 0      //where m represents the table values

    for( i = 1 to n )
            for( j = 0 to Capacity)
                if( w[i-1] > j)
                        m[i][j] = > previous optimal value
                else
                m[i][j] = max( m[i-1][j], m[i-1][j-w[i-1]] + v[i-1] )   -> taking the max of the previous
optimum value and the value after adding the current item }
```

Here, we create a 2 dimensional array m[][] to store the optimal value as we traverse across the list of weights. We also calculate the solutions to the sub problems .

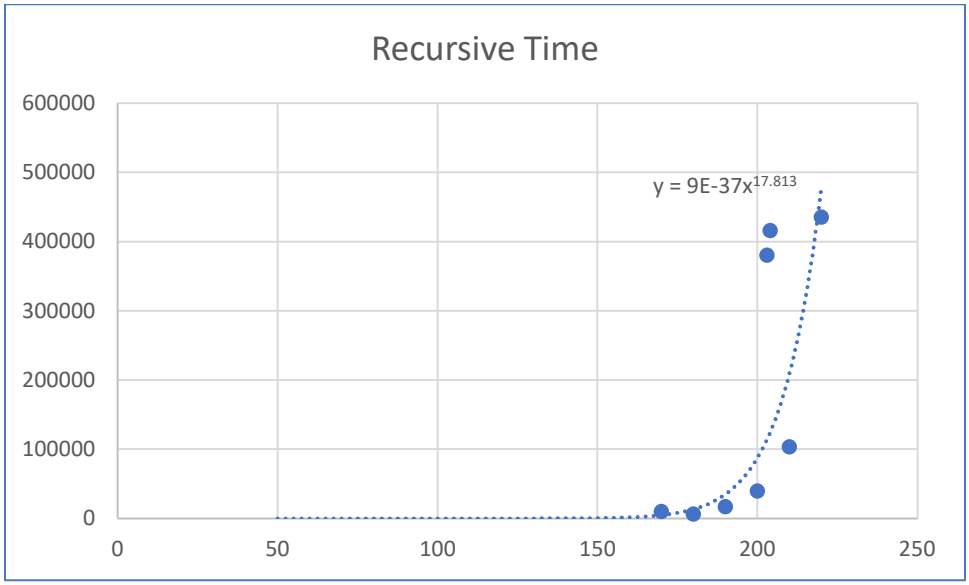
1. At first it checks the number of items and the Capacity are valid.
2. It evaluates the price when the current item is included
3. It evaluates the price without considering current item
4. Take the best from the above (Greater one)

b) Program has been submitted in teach

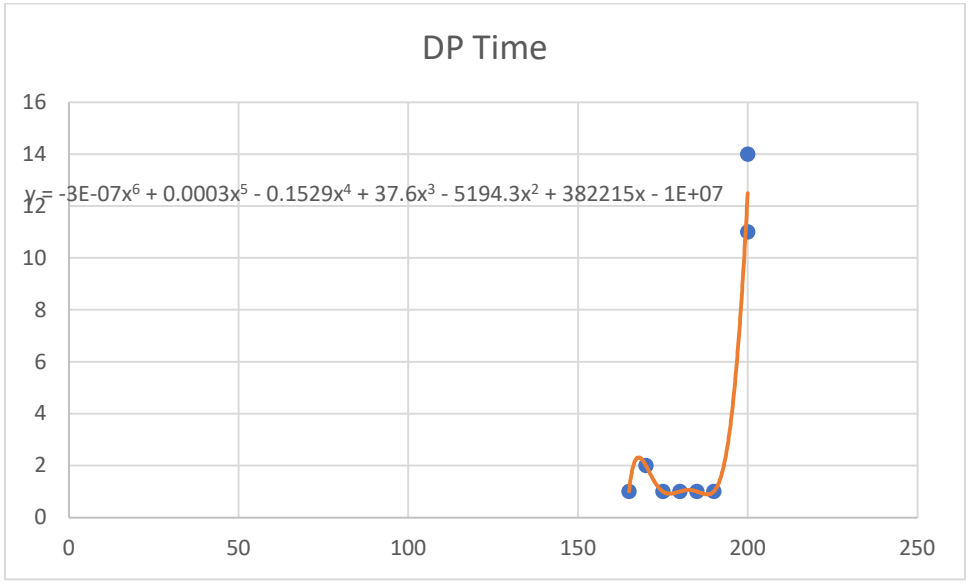
c) Plotting values for Running time :

No of Items	Weight	Recursive Time	DP Time	Max Recursive	Max DP
165	1000	84557	1	10167	10167
170	1000	9928	2	9992	9992
175	1000	51769	1	10886	10886
180	1000	6310	1	9929	9929
185	1000	253746	1	11560	11560
190	1000	16763	1	10287	10287
200	1000	39738	11	8866	8866
200	1000	394072	14	11327	11327
210	1000	103461	2	10782	10782
220	1000	435264	6	10108	10108

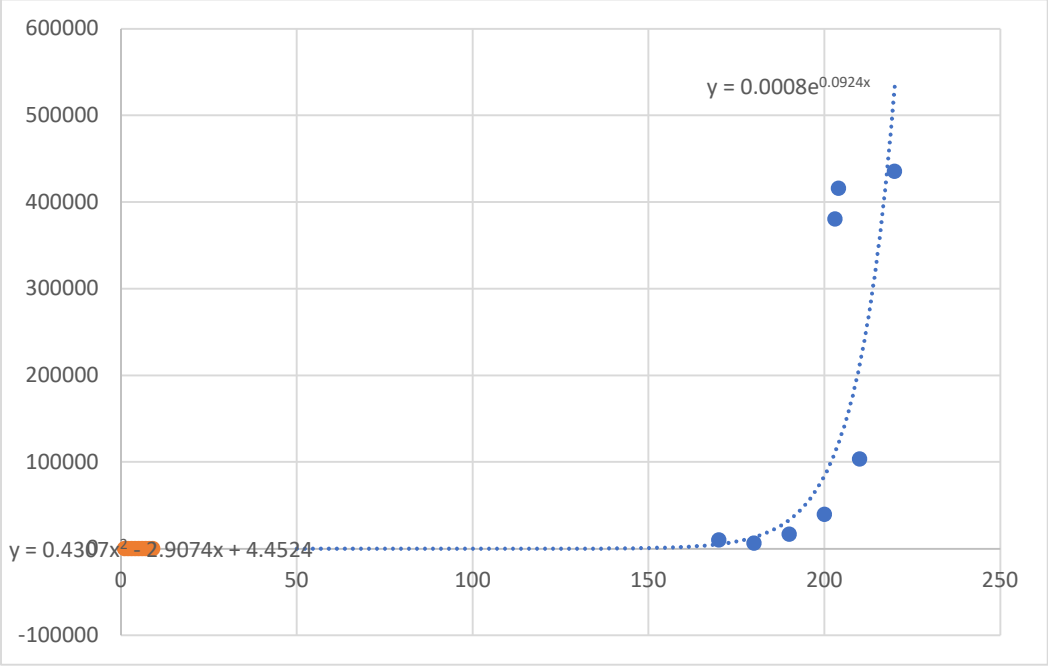
Plotting for Recursive:



Plotting for Dynamic Programming :



Combined chart for Recursive (Blue Curve) and Dynamic Programming(Orange Curve)



d) From the above combined graph it is clear that Dynamic Programming seems to be run faster than the Recursive algorithm.

Initially I was not able to get the value for DP as it constantly stayed between 0 and 1 milli seconds. So when the number of items and Capacity were increased to larger values there was a shift. Hence we can say the following :

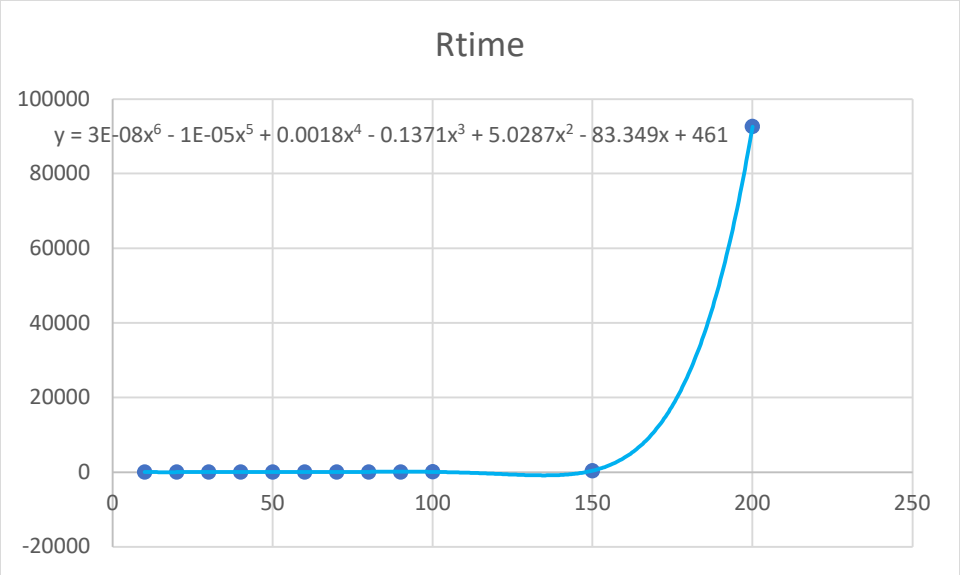
- Running Time for Recursive – O(2^n)
- Running Time for Dynamic Programming– O(nW)

As we can see, the running time for Dynamic Programming is O(nW), where W is the Capacity,this is a pseudo-polynomial which is so much better than the running time for recursive algorithm - O(2^n).

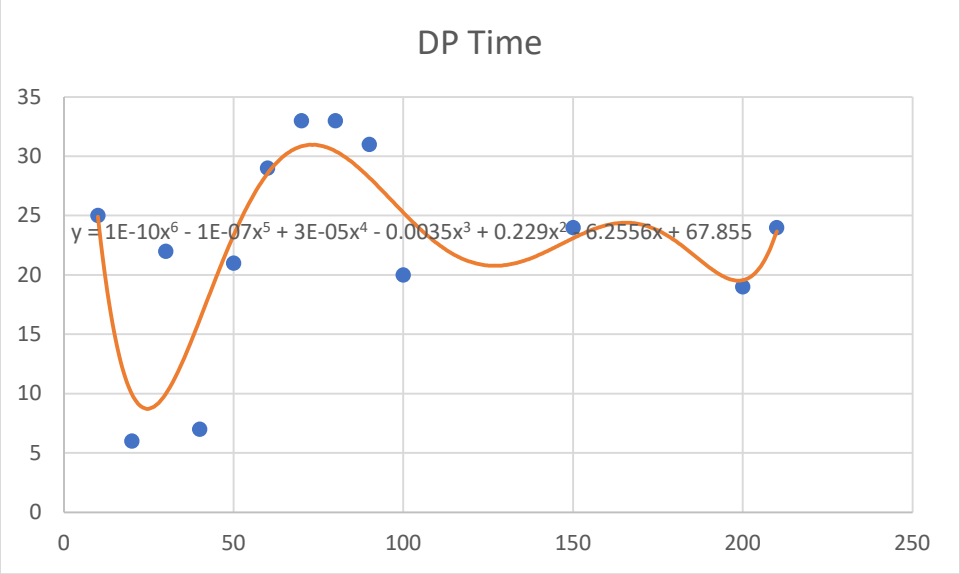
When the weight was increased,

Plotting for increased weight , weight = 20000

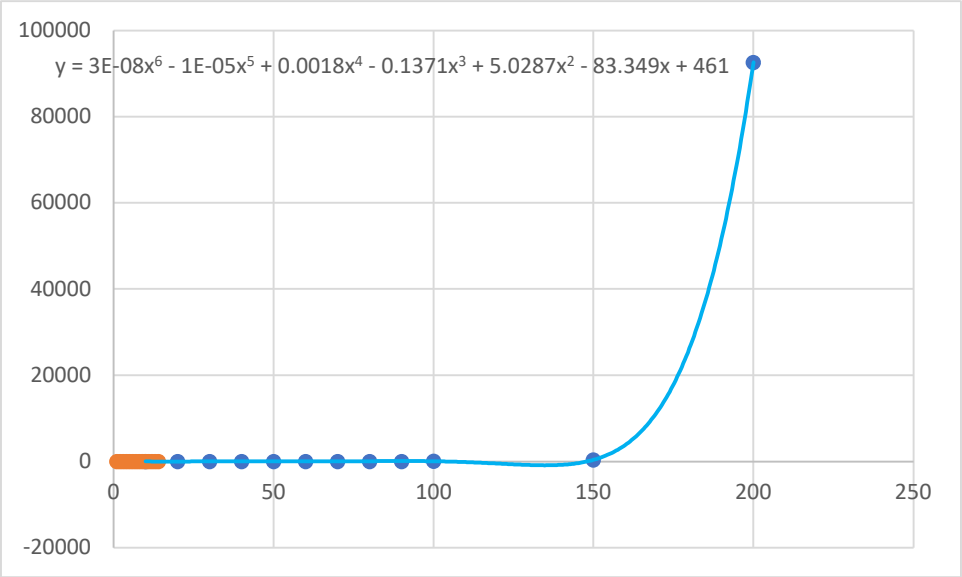
Recursive Algorithm :



Dynamic Programming :



Combined graph : Orange -DP and Blue – Recursive for weight as 20000



Problem 4: 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.

```
Function Shopping()
{
        //input file is read
        T = No of test Cases
        M = array representing the table

        for(i=1 to T)
        {
                for(j=1 to n)
                        w[] = reading file and storing the weight
                        p[] = reading file and storing the price
        }
        //let F be the number of family members
        for(x=1 to F)
        {
                Knapsack(w,p,n,mw,m)
        }
Writing back to the results file.
}
```

This is same as the previous knapsack algorithm. Here, for each of the family member, the knapsack algorithm is called.

b) Shopping spree is same as the knapsack problem which has been implemented in question 3. Hence the running time of Shopping spree is O(nK) which is a pseudo-polynomial.

   Lets consider test case 2

   - We have family members as 4
   - N= 6
   - W for 1st member is 25

   Hence 4 times the knapsack algorithm is invoked.

   Running time is (4*(nW))

   Member 1 =4*(6*25) =600

c) Code has been attached in teach.