CS 325 – Analysis of Algorithm

Name: Shou-Cheng Wu

ID: 933-920-148

Email: [wusho@oregonstate.edu](mailto:wusho@oregonstate.edu)

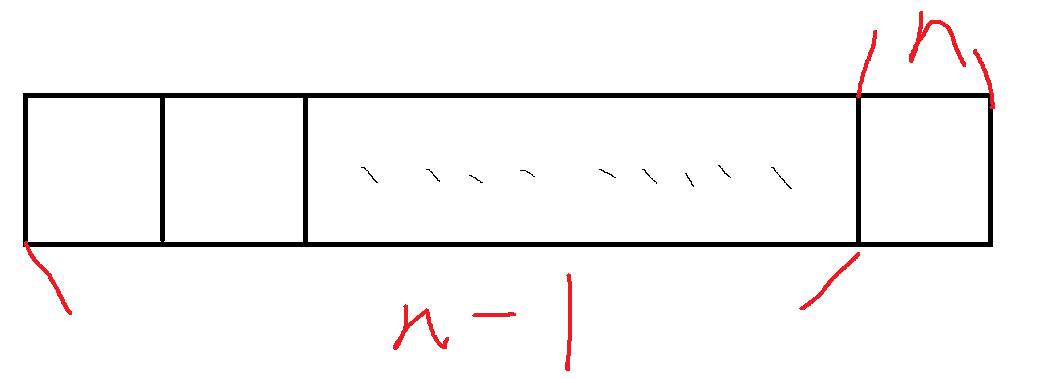
**Problem 1: 0-1 Knapsack: Recursive vs DP**

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

**Recursive:**

**Verbal description:**

If we want to use recursive method to do knapsack problem, what we need to consider first is to compare the last element with previous case. We are supposed to divide this problem into two parts. The detailed graph is shown below.



If the last item’s weight is less or equal to the knapsack’s carrying capacity, we need to compare which value is bigger, last item’s value plus previous case’s maximum value or previous case’s maximum. Otherwise, we move our index forward to keep doing recursive.

**Pseudo-code**

Knapsack\_rec(W, n, wt, val)

if W==0

return 0

if n==0

return 0

if wt[n-1]>W

return Knapsack\_rec(W,n-1,wt,val)

else

return max(val[n-1]+Knapsack\_rec(W-wt[n-1],n-1,wt,val), Knapsack\_rec(W,n-1,wt,val))

**DP:**

**Verbal description:**

About dynamic programming solution for this knapsack problem, what we need to do first is using an empty nxw matrix to record total values. The row and column would be the item and weight limitation respectively. Using current item’s information and comparing the maximum value with previous case step by step, we could get the maximum value from the last element in the matrix.

**Pseudo-code**

Knapsack\_DP(wt, val ,W)

Create a len(val)\*W table, called V and make all of the elements to be 0

for i = 1 to len(val)+1

for w =1 to W+1

if wt[i-1]<=W:

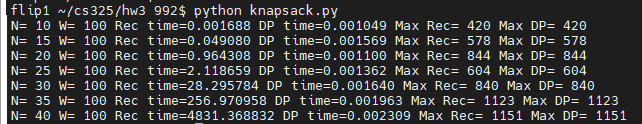
V[i][w] = max(val[i-1]+V[i-1][w-wt[i-1]],V[i-1][w])

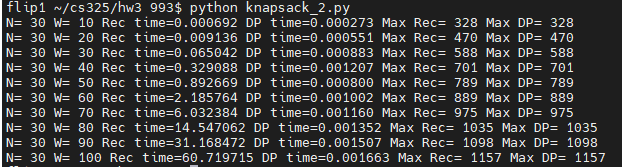
else:

V[i][w] = V[i-1][w]

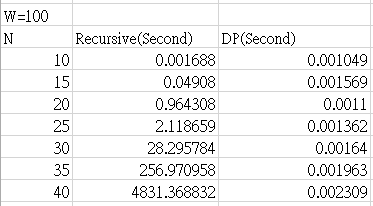
return v[len(val)][W]

**b) Implement both algorithms in one program named knapsack. Your program should randomly generate test cases that are solved using both the DP and Recursive algorithm. The program should output to the terminal: n, W, time for the DP algorithm, max for the DP, time for the Recursive algorithm, max for Recursive. The max values should be the same. Sample output is below**



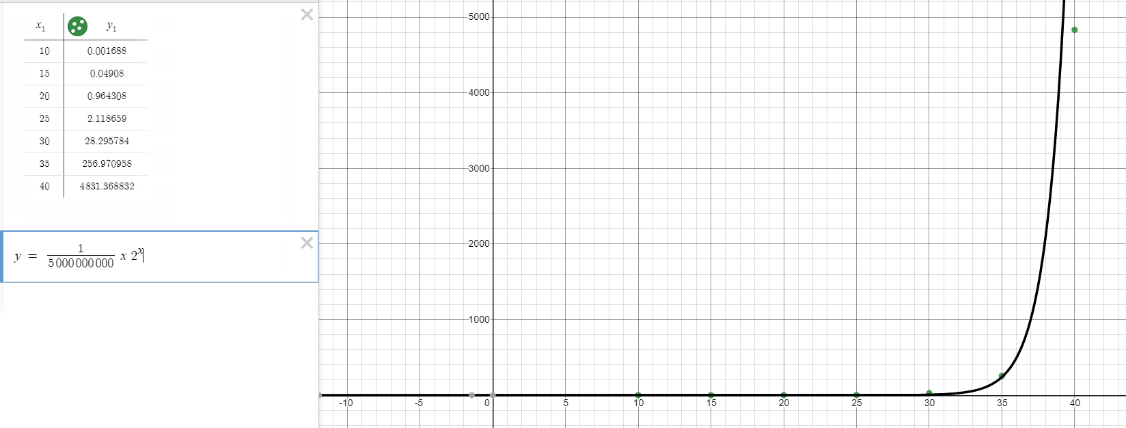


1. **Conduct experiments to collect running times for randomly generated input. Since there are two variables, n and W, you can hold one constant while varying the other and vis-a-versa. This may result in several graphs. Plot the data and fit curves.**

****

N=10,20…100, W=100

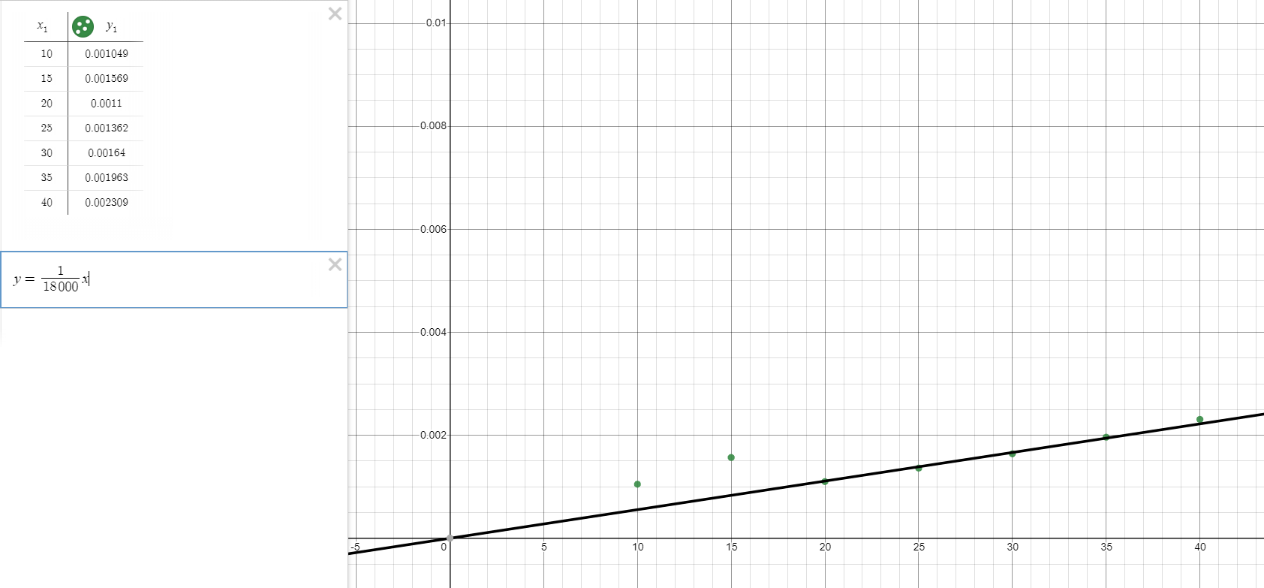
**Recursive-method:**



N=10-40, W=100

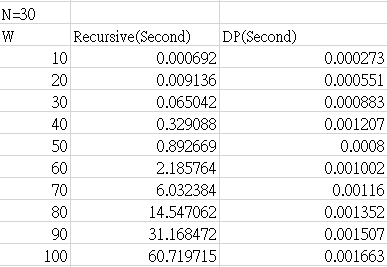
y = x

DP-method:



N=10-40, W=100

y = x



N=30, W=10,20…100

1. **If the recursive algorithm is too slow you can collect data using different values of W and n. Discuss your implementation, results and how you collected the data. How does W change the running time?**

I repost the two tables I show on question c below. If we hold one variable, W or N,

The running time would increase as another variable gets higher. This makes sense.

For example, if we hold variable,n, increasing w means that it is possible that we

could put more items in the knapsack. It would generate more possibilities. Take DP-

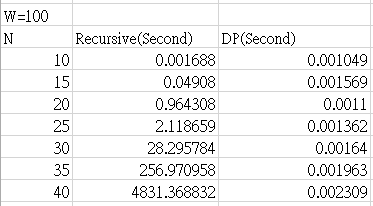
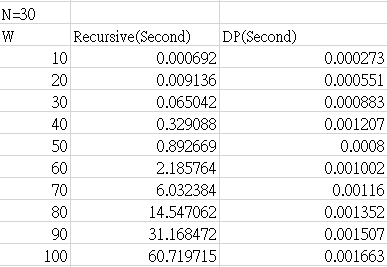
algorithm for example, if we increase w from 10 to 20, we need to make our value

table larger. The table size would change from 10xN to 20xN. Because we need to fill

the table based on our DP algorithm, its original time complexity, O(10N), would

become O(20N). Similarly, if we hold w and change n, it would also make our value

table larger and increase its time complexity.

**** 

**Problem 2: Shopping Spree:**

1. **Give a verbal description and give pseudo-code for your algorithm. Try to create an algorithm that is efficient in both time and storage requirements.**

**Verbal description:**

This question is pretty similar as the previous one. There are two things that are different. First, we need to process the data into a correct format. In other words, we need to group data by family. Second, we need to compare the values in our matrix and return the items that person picks up. More specifically, we assume that this matrix v is a NxK matrix. If v[n][k] is not equal to v[n-1][k], this means that this is an item that person should pick up. And the total price the family could get can be obtained by summing up all the values returned from the previous function.

**Pseudo-code**

ShoppingSpree(wt, val ,W)

Create a len(val)\*W table, called V and make all of the elements to be 0

for i = 1 to len(val)+1

for w =1 to W+1

if wt[i-1]<=W:

V[i][w] = max(val[i-1]+V[i-1][w-wt[i-1]],V[i-1][w])

else:

V[i][w] = V[i-1][w]

i = len(val)

k =W

table = []

while i != 0:

if v\_table[i][k] != v\_table[i-1][k]

store i into the table array

i = i – 1

k = k – wt[i]

else:

i = i – 1

return table, V[len(val)][W]

w is a vector to store the maximum weight each person can carry in a family

wt is a vector to store the items’ weight for each case

val is a vector to store the items’ value for each case

for i =0 to size(w)

a, b = ShoppingSpree(wt,val,w[i])

a means the actual knapsack items a person can carry

sum all of b here could get the maximum total price that family would obtain

**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 Mi pounds for 1 ≤ i ≤ F.**

Based on the pseudo-code I write above, we can guess the running time of one person is N+N. If want to know the running time of whole family, it would be .

**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.**

I submit it through teach system.

