Oregon State University

CS 325 - Spring 2021

Prof. Julianne Schutford

Homework 2

Knapsack 0-1

by

Abdullah Saydemir

saydemia@oregonstate.edu

April 23, 2021

Q1)

b) Running times are collected on *flip* servers. Before the experiment, uptime command returned *1.67% and 1.98%* user activity on the servers for two runs. The data collected is as follows.

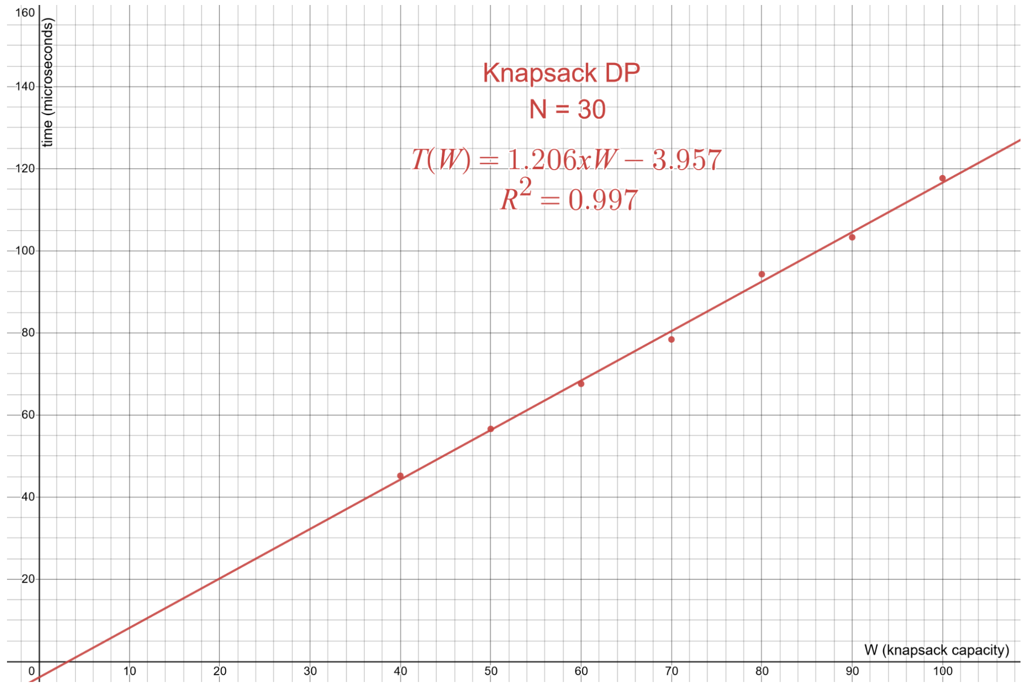
For constant number of items

N = 30	Running Time (milliseconds)	
Capacity (W)	Knapsack Recursive	Knapsack DP
40	89.945	0.045
50	228.961	0.057
60	579.079	0.068
70	1365.099	0.078
80	3356.007	0.094
90	7071.989	0.103
100	13242.491	0.118

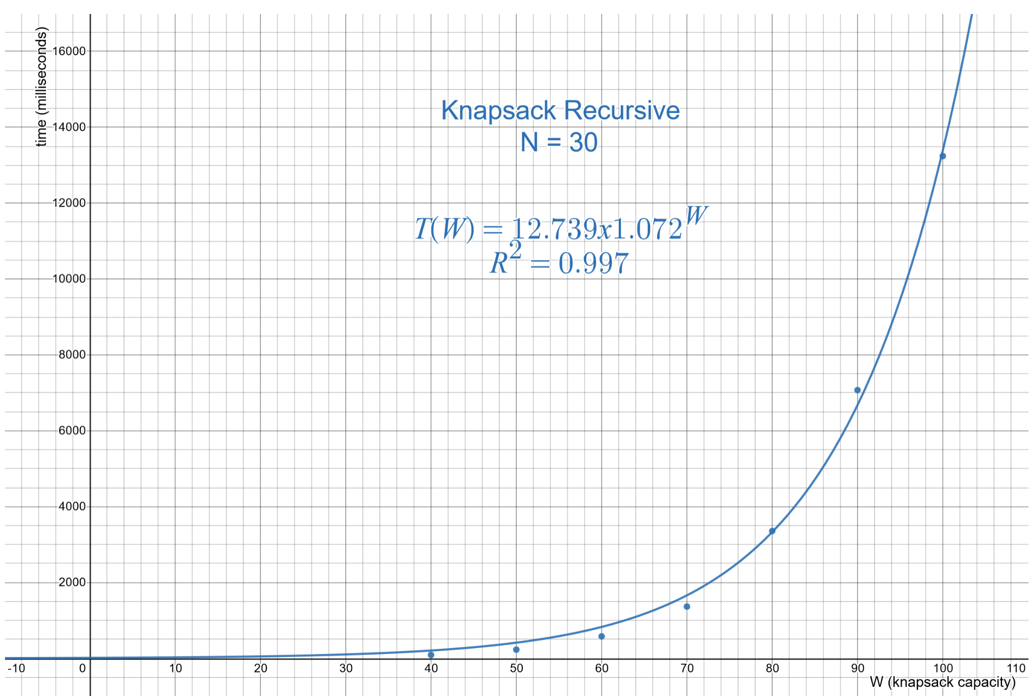
For constant knapsack capacity

W = 100	Running Time (milliseconds)	
# of items (N)	Knapsack Recursive	Knapsack DP
10	0.051	0.041
14	0.800	0.055
18	11.871	0.071
22	144.605	0.085
26	986.795	0.100
30	5311.888	0.115
34	23747.945	0.129

For constant number of items (N = 30)[1]



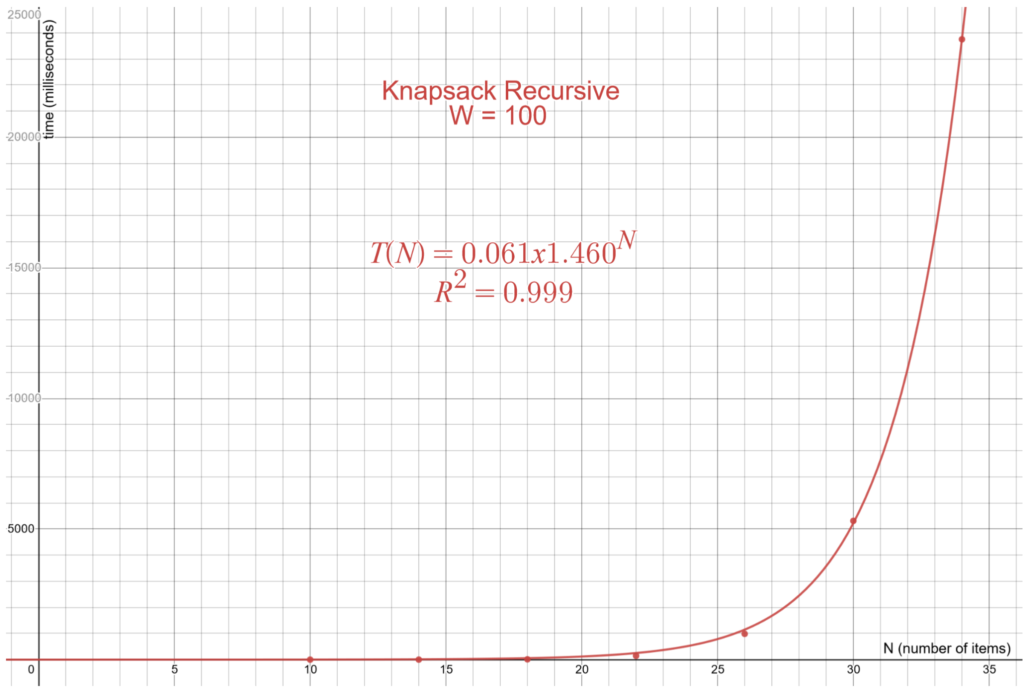
(Graph 1)



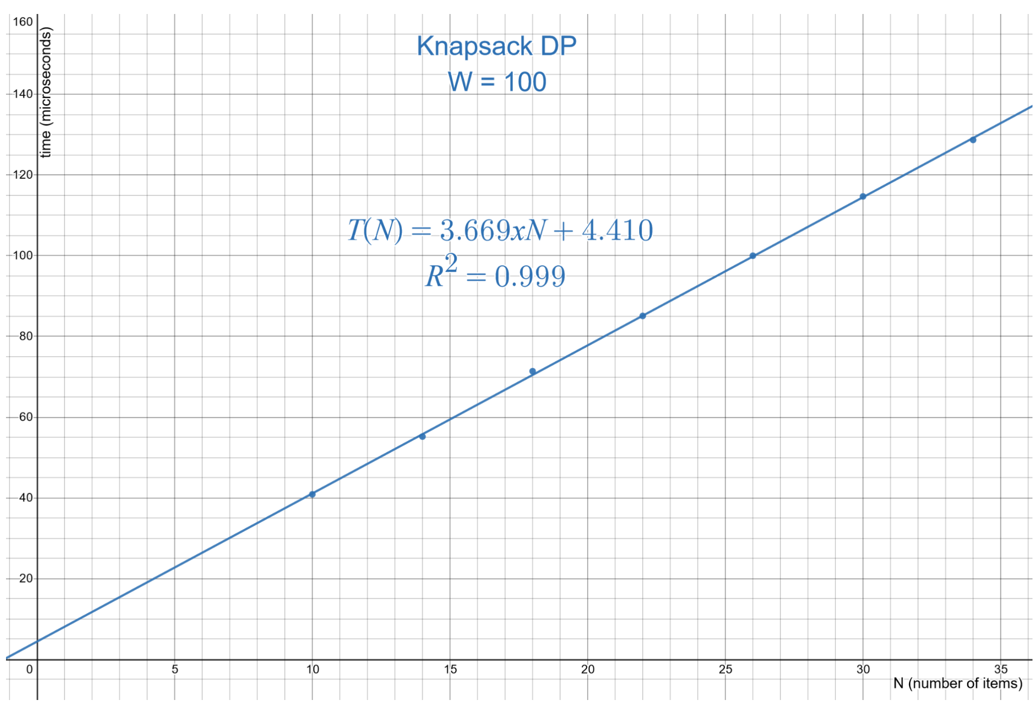
(Graph 2)

[1] Desmos link of Graph 1 and Graph 2: https://www.desmos.com/calculator/njyas3lhpu

For constant knapsack capacity (W = 100)[2]



(Graph 3)



(Graph 4)

---

[2] Desmos link of Graph 3 and Graph 4: https://www.desmos.com/calculator/m2mrel5xm3

c)

It is important to get uniformly random values because otherwise it affects the running time. If the weights were too close then the algorithm would choose the items just looking at the values. Similarly, if the values were the same then the algorithm would choose the light items until the knapsack is full. To eliminate these possibilities I used a *Mersenne Twister (line 78 in Figure 2)* engine with *uniform_int_distibution (lines 79-80 in Figure 1)* which provides both random and uniformly distributed integers.

```
74  int main()
75  {
76      // initialize random number generator engines
77      random_device rd;
78      mt19937 rng(rd());
79      uniform_int_distribution<int> unival(1,100);
80      uniform_int_distribution<int> uniwt(1,15);
81
```

(Figure 1)

Because the minimum capacity was 40 in the runs that item size is constant, I collected random integers between [1,15) to have multiple items that can fit into knapsack. If the capacity to average weight ratio were too small then this would also decrease the running time since we could put very few items to the knapsack. This effect actually can be observed in the graphs on Desmos. For larger values of *W* the algorithms work very consistently. However, for small values of *W*, there are gaps between the curves and the points.

To smooth the curves and minimize the problems with the *flip* server, I ran the algorithm 10 times for each item size & capacity *(line 109 in Figure 2)* and took the average of them *(lines 126-127 in Figure 2)*. In the submitted code, however, it takes the average of 3 runs, not to take your time.

```
        for (int t = 0; t < 10; ++t)
        {

            auto start = chrono::steady_clock::now(); // start the clock
            max_rec = knapsack_recursive(wt,val,i,N); // run the code (Recursive)
            auto end = chrono::steady_clock::now(); // stop the clock
            time_rec_avg += chrono::duration_cast<chrono::nanoseconds>(end - start).count();


            start = chrono::steady_clock::now(); // start the clock
            max_dp = knapsack_dp(wt,val,i,N);  // run the code (Dynamic)
            end = chrono::steady_clock::now(); // stop the clock
            time_dp_avg += chrono::duration_cast<chrono::nanoseconds>(end - start).count();

        }

        // take the average, turn to milliseconds
        time_dp_avg = time_dp_avg *1e-6 / 10.0;
        time_rec_avg = time_rec_avg *1e-6 / 10.0;
```

(Figure 2)

Another important note is that the timescale is different in above graphs. Since the DP algorithm works very fast compared to the recursive one, I had to change the scale of the DP graphs to microseconds.

Graphs 1 and 2 show that capacity of the knapsack affects the running time of both algorithms. For the DP algorithm, the relation is pretty much linear with an R^2 value of *0.997,* as expected. Because DP implementation is $\Theta(W \: x \: N)$, holding *N* constant should give $\Theta(\,W)$ running time as it is in the graph. Memoization is not used in the recursive implementation. Therefore, the algorithm calculates the subproblems over and over again and the running time is some kind of exponential. The formula I used to approximate was $a \: x \: b^{\,W}$ and the curve came out really good with an R^2 value of *0.997.*

For the constant capacity *(W)* and varying item size (*N*), curves seem similar to the first two graphs. Since the capacity *W* is constant, curves depend on the item size *N*. The more items it gets, the more it compares the items and slows. Still, the recursive algorithm runs in exponential time and the DP algorithm runs in linear time.

Q2)

a)

Verbal description of the main algorithm is as follows :

- For each test case, read the number of items.
- For each item, read the value and the weight
- Read the number of family members.
- For each family member run the knapsack algorithm and learn which items that member will carry.
- Learn total value by summing all the items in the list.
- Print test case, total value, each family member and the items he/she carries

Verbal description of knapsack algorithm is as follows :

Goal : Given max capacity, items, values and weights return the maximum value of items that can be carried in the knapsack. Let this be *OPT(n, W)*

- If there is no item in the list, then return an empty set.
- If the knapsack capacity is 0, then return an empty set.
- For all items i in the list
  - If *OPT(i, w)* does not select i^{th} item in the list then *OPT(i, w)* selects the best of the rest { 1,2,3, .., i-1 }
  - If *OPT(i, w)* selects the item *(* $w_i <= w$*)* then take the item, set the new weight limit to $w - w_i$ and *OPT(i, w)* selects the best of the rest {1,2,.., i-1 } with the new weight.
- Return selected items.

Pseudocode of the *shopping.cpp* is as follows.

Please do not get disturbed by the i and i-1 difference. Since I shifted the table by adding base cases to the left, I had to get rid of that +1 somewhere. That's why you see i-1 instead of i. Also, I used array notation [i][j] instead of coordinate notation [i , j], because it was easier to change it to actual code.

```
FUNCTION knapsack_dp ( Vector val, Vector wt, int W, int n)
    LET table <- int[ n+1 ][ w+1 ]                          // initialize the table

    FOR i <- 0 to n DO                                      // set base cases to 0
        table[ i ][ 0 ] <- 0
    END FOR
    FOR j <- 0 to W DO
        table[ 0 ][ j ] = 0
    END FOR

    FOR i <- 1 to n+1 DO                                    // for each item i
        FOR w <- 1 to W+1 DO                                // and capacity w
            IF w_{i-1} <=  w THEN                           // if there is enough space
                table [ i ][ w ] <- max { table [ i-1 ][ w ] ,
                                          v_{i-1} + table [ i-1 ] [ w-w_{i-1} ] }
            ELSE THEN                                       // if there is not enough space
                table [ i ][ w ] <- table [ i-1 ] [ w ]
            END IF
        END FOR
    END FOR

    LET i <- n and j <- W
    LET item_ids <- new Vector()                            // ids of the items in the knapsack

    WHILE  i > 0 and  j > 0 DO
        IF table [ i ] [ j ] != table [ i-1 ][ j ] THEN     // if the weight differs
            item_ids.insert ( i -1)                         // then that item is included
            j = w - wt[ i - 1 ]                             // reduce the weight
        END IF
        i--
    END WHILE
    RETURN item_ids
END FUNCTION
```

```
FUNCTION main()
    FILE input := read(shopping.txt)
    IF file is NOT correctly opened THEN
        print "Error reading file"
        exit
    END IF

    LET T <- cin                                        // number of test cases
    FOR i <- 0 to T DO
        LET N <- cin                                    // number of items

        LET wt <- new Vector()                          // weights of items
        LET val <- new Vector()                         // values of items
        LET total_price <- 0

        FOR j <- 0 to N DO                              // read integers from the file
            val.insert( cin )
            wt.insert( cin )
        END FOR

        LET F <- cin                                    // number of family members
        LET family_items <- new Vector<Vector>()        // item list

        FOR k <- 0 to F DO                              // for each family member

            LET carry <- cin
            LET var <- knapsack_dp ( wt, val, carry, N)     // get the items
            FOR p <- 0 to length( var ) DO
                total_price += val [ var[ p ] ]         // add up the values
            END FOR
            family_items.insert( var )                  // insert items to family list
        END FOR
```

```
            print “Test Case ” + ( i + 1 )
            print “Total Price ” + total_price

            FOR q <- 0 to length( family_items ) DO
                print (q+1) + “: “                                  // print the member
                FOR s <- 0 to length( family_items[ q ] ) DO
                    print ( family_items[ q ][ s ] + 1 ) + “  “      // and his/her items
                END FOR
            END FOR
            Clear all the variables
        END FOR
END FUNCTION
```

+1 comes from difference of starting index and counting numbers