Coin and Knapsack Problems with Dynamic Programming

CIS 315

coin problem

- given coins of denominations v_1, v_2, …, v_n
- and a target W
- question: is there a combination of coins that sum up to W?
- optimization question: minimize the number of coins that add to W
- greedy doesn't always work
- other issue: is there unlimited supply of each denomination?

coin problem with repetition

- we'll go with input $v_1, v_2, \ldots, v_n$ and target W
- unlimited supply of each coin v_i
- is possible to supply change for W?

subproblem: C[w] is true iff it is possible to make change for w (for any integer $w \leq W$)

recurrence:

- C[w] = false if $w<0$
- C[0] = true
- $C[w] = V_{1 \leq i \leq n}\ C[w-v_i]$

coin problem without repetition

- denominations v_1, v_2, …, v_n and target W
- only one coin of each denomination

subproblem: C[w,k] is true iff it is possible to make change for w (for any integer w≤W) using coins v_1, v_2, …, v_k

recurrence:

- C[w,k] = false if w<0
- C[0,k] = true for all k
- C[w,0] = false if w>0
- C[w,k] = C[w,k-1]VC[w-v_k,k-1]

knapsack problem

- a bag (knapsack) holds up to W pounds
- choice of n items of weights w_1, w_2, …, w_n
- the items have value v_1, v_2, …, v_n
- maximize value of items that fit into bag subject to the weight constraint
- items cannot be broken up (“integer knapsack”
- as before, issue of repetition of items

knapsack with repetition

define K[w] = maximum value achievable with a knapsack of capacity w

$$K[w] = \text{MAX} \{ K[w-w_i] + v_i \mid 1 \le i \le n, w_i \le w \}$$

note: MAX Ø = 0

algorithm

```
K[0] = 0

for w=1 to W
    K[w] = 0
    for i = 1 to n
        if w_i≤w then
            K[w] = max[ K[w], K[w-w_i]+v_i ]

return K[W]
```

- time is O(Wn)
- this is actually not polynomial – input length is n + lg W
- it is *pseudo-polynomial* time

example

item (i)	1	2	3	4
weight (w_i)	6	3	4	2
value (v_i)	30	14	16	9
v_i/w_i	5	4.66	4	4.5

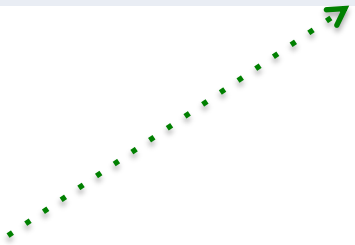
With W=10, optimal solution is one item 1 and two item 4's, for a total of $48.

If no repetition, optimal solution is items 1 and 3, for $46.

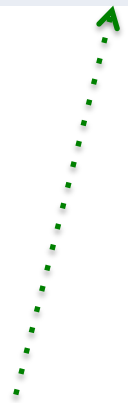
Neither of these optimal solutions comes from the obvious greedy approach (best value-to-weight ratio first).

using $K[w] = MAX \{ K[w-w_i] + v_i \mid 1 \le i \le n, w_i \le w \}$

	0	1	2	3	4	5	6	7	8	9	10
K	0	0	9	14	18	23	30	32	39	44	48
max i	-	-	4	3	4	2	1	2	1	1	1



(i=1)	$K[1]+v_1=0+30$
(i=2)	$K[4]+v_2=18+14$
(i=3)	$K[3]+v_3=16+14$
(i=4)	$K[5]+v_4=23+9$



(i=1)	$K[4]+v_1=18+30$
(i=2)	$K[7]+v_2=32+14$
(i=3)	$K[6]+v_3=30+14$
(i=4)	$K[8]+v_4=39+9$

knapsack no repetition

define K[w,j] = maximum value achievable with a knapsack of capacity w using items 1,2,…,j

$K[w,j] = MAX[\ K[w,j-1],\ K[w-w_j,\ j-1]+v_j\ ]$

don't use item j

use item j (and test if $w \geq w_j$)

```
public static int recCoins(int t)  // pure recursion
    {
        int curCoin, prevNum, curNum;

//      if (t<0) return Integer.MAX_VALUE-1;
        if (t==0) return 0;

        curNum = Integer.MAX_VALUE-1;
        for (int i=0; i<denom.size(); i++)
            {
                curCoin = denom.get(i);
                if (t>=curCoin)
                {
                    prevNum=recCoins(t-curCoin); // recursively
                    if (1+prevNum<curNum)      // determine #coins
                        curNum = 1+prevNum;
                }
            }

        return curNum;
    }
```

code in class to determine min number of coins needed, pure recursion

```
public static int memoCoins(int t)
    // recursive but memoized (results cached in hashmap, tho
array would be fine also)
    {
        int curCoin, prevNum, curNum;
        if (t==0) return 0;
        if (mapCoins.containsKey(t))
            return mapCoins.get(t); // return if already known

        curNum = Integer.MAX_VALUE-1;
        for (int i=0; i<denom.size(); i++)
        {
            curCoin = denom.get(i);
            if (t>=curCoin)
            {
                prevNum=memoCoins(t-curCoin);  // recursive call
                if (1+prevNum<curNum)   // to memoized routine
                    curNum = 1+prevNum;
            }
        }

        mapCoins.put(t,curNum); // save computed value
        return curNum;    }
```

```
public static int iterCoins(int T)
    // iterative, results in array, calculated from smallest to
largest
    {
        int curCoin;
        numCoins = new int[T+1];

        numCoins[0]=0;
        for(int t=1; t<=T; t++)
        {
            numCoins[t] = Integer.MAX_VALUE-1;  //hack
            for (int i=0; i<denom.size(); i++)
            {
                curCoin = (Integer)denom.get(i);
                if (curCoin<=t && 1+numCoins[t-
curCoin]<numCoins[t])
                    numCoins[t]= 1+numCoins[t-curCoin];
            }
        }

        return numCoins[T];

    }
```