Coin and Knapsack Problems with Dynamic Programming

CIS 315
coin problem

• given coins of denominations $v_1, v_2, \ldots, v_n$
• 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:**
- $C[w] = \text{false}$ if $w < 0$
- $C[0] = \text{true}$
- $C[w] = V_{1 \leq i \leq n} C[w - v_i]$
coin problem without repetition

• denominations $v_1, v_2, \ldots, v_n$ 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 $w \leq W$) using coins $v_1, v_2, \ldots, v_k$

recurrence:
• $C[w,k] = \text{false}$ if $w<0$
• $C[0,k] = \text{true}$ for all $k$
• $C[w,0] = \text{false}$ if $w>0$
• $C[w,k] = C[w,k-1] \lor C[w-v_k, k-1]$
knapsack problem

• a bag (knapsack) holds up to $W$ pounds
• choice of $n$ items of weights $w_1, w_2, \ldots, w_n$
• the items have value $v_1, v_2, \ldots, v_n$
• 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 $K[w] = \text{maximum value achievable with a knapsack of capacity } w$

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

note: $\text{MAX } \emptyset = 0$
algorithm

\[
K[0] = 0
\]

\[
\text{for } w=1 \text{ to } W \\
\quad K[w] = 0 \\
\quad \text{for } i = 1 \text{ to } n \\
\quad \quad \text{if } w_i \leq w \text{ then} \\
\quad \quad \quad K[w] = \max[ K[w], K[w-w_i]+v_i ]
\]

\text{return } K[W]

- time is \( O(Wn) \)
- this is actually not polynomial – input length is \( n + \lg W \)
- it is \textit{pseudo-polynomial} time
example

<table>
<thead>
<tr>
<th>item (i)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight ( w_i )</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>value ( v_i )</td>
<td>30</td>
<td>14</td>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>( \frac{v_i}{w_i} )</td>
<td>5</td>
<td>4.66</td>
<td>4</td>
<td>4.5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>14</td>
<td>18</td>
<td>23</td>
<td>30</td>
<td>32</td>
<td>39</td>
<td>44</td>
<td>48</td>
</tr>
</tbody>
</table>

$\text{max } i$  
-  
-  
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, ..., \( j \)

\[
K[w,j] = \max \left[ K[w,j-1], K[w-w_j, j-1] + v_j \right]
\]

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;
}
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-curfCoin]<numCoins[t])
                    numCoins[t]= 1+numCoins[t-curfCoin];
            }
        }

        return numCoins[T];
    }