CS 341: Algorithms
Module 6: Dynamic Programming

Armin Jamshidpey, Eugene Zima

Based on lecture notes by many previous CS 341 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2019

Yet More Coins

Motivation:

We assume that we have an unlimited number of coins of various denominations: $d_1, d_2, \ldots, d_m$.

We have the same objective as before: exchange a given sum S with the smallest number of coins possible.

The denomination system presents us with three possibilities:

- The d_i values are such that a greedy strategy **can** exchange the sum S in an optimal fashion.
- The d_i values are such that a greedy strategy **cannot** exchange the sum S, in an optimal fashion although such an optimal exchange does exist.
- The d_i values are such that no strategy can exchange the amount S.

  Trivial example: $m = 1$, $d_1 = 2$ and S is an odd amount.

A Recursive Strategy for the Coin Problem

We would like to design an algorithm that produces good results for any denomination system: an optimal solution or an indication that no solution exists.

Notation:

Let *coins*[*i*] be the smallest number of coins required to exchange the sum *i*. We assume *coins*[*i*] will be "undetermined" if no solution exists.

Note that *coins*[0] is 0 (no coins needed).

Idea behind the solution:

Unlike the greedy approach, we try all possibilities for the first coin. Each possibility d_j reduces the problem into the same problem for a sum i diminished by the value of d_j. So, we can establish a recurrence relation between optimal solutions of different sub-problems with the added constraint that we always ask for the minimum coin number:

$$coins[i] = 1 + \min_{\substack{1 \leq j \leq m \\ d_j \leq i}} \{coins[i - d_j]\}$$

This leads to the following naive implementation:

```
function coins(i):
  if (i=0) then return 0; // base case
   // recursion:
  min:=infinity; // Use a very large number here
  for j:=1 to m do
    if (d[j] <= i) then
      smaller_sol := coins(i - d[j]);
    if smaller_sol < min then min := smaller_sol;
  return 1 + min;
```

The main program will call coins(S).

Dynamic Programming for the Coin Problem

The unfortunate aspect of the pseudo-code 1 is that many of the calls are redundant. For example, coins[2] is called several times. Consider computation of Fibonacci numbers to see the problem. Can we eliminate this?
Now consider pseudo-code 2:

```
coins[0] := 0;
for i := 1 to S do
  min := infinity;
  for j := 1 to m do
   if d[j] <= i and coins[i - d[j]] < min then
     min := coins[i - d[j]];
  coins[i] := 1 + min;
return coins[S];
```

What is the execution time for pseudo-code 2?
Note: Unlike pseudo-code 1 it is not recursive.
Just two nested loops with constant time operations at the innermost part, so: $\Theta(mS)$.
Recall that greedy solution:

```
while S > 0 do
  c := value of the largest coin no larger than S;
  num := S div c;
  pay out num coins of value c;
  S := S - num*c;
```

had complexity in $\Theta(m(\log S)^2)$.

Recovery of the Solution

The previous pseudo-code only gives us the coin count.
There is no report on which coins make up the optimal solution.
We need an array (choice[ ]) that will track the best solution.

To recover the whole solution it is sufficient to remember only the last choice, since the previous choices can be recovered from the sub-problems.

Finally: This is dynamic programming!

```
coins[0] := 0;
for i := 1 to S do
  min := infinity;
  for j := 1 to m do
   if d[j] <= i and coins[i - d[j]] < min then
     min := coins[i - d[j]];
     minchoice := j; // Note!
   coins[i]:=1+min;
   choice[i] := minchoice; // Note!
// Now recover the solution
if coins[S] = infinity then
  write 'No solution!';
else
  while S > 0 do
    write d[choice[S]];
    S:=S-d[choice[S]];
```

What is Important About the Last Program?

In dynamic programming the order of computation permits us to look up precomputed values in an array (instead of using recursion).

The precomputed values are considered to be the solutions of subproblems.

For example: in the coin changing problem we used a 1D array coins[1..S] where coins[i] recorded the number of coins necessary to pay out sum i.

A special value of infinity recorded the failure of the algorithm.

Note that there was also the notion of where the answer would be found (in coins[1..S] it was in coins[S]).

The “personality” of the algorithm will be established by the strategy that allows you to build solutions for bigger problems from the solutions of smaller problems (which is drive by the DP-recurrence). For coin changing:

$$coins[i] = 1 + min_{\substack{1 \leq j \leq m \\ d_j \leq i}} \{coins[i - d_j]\}$$

A Recipe for Designing a D. P. Algorithm

1. Identify the subproblem.
   Typically the computation of solutions of the subproblems will make it natural to retain the solutions in an array.
   We need to: know the matrix dimensions; specify the precise meaning of the value in any cell of the array;
   specify where the answer will be found in the array.
2. Specify how a subproblem contributes to the solution of a larger subproblem (establish DP-recurrence).
   How does the value in a cell of the array depend on the values of other cells in the array?
3. Set values for the base cases.
4. Specify the order of computation. The algorithm must clearly state the order of computation for the cells.
5. Recovery of the solution (if needed). Keep track of the subproblems that provided the best solutions.
   Use a traceback strategy to determine the full solution.

Longest Common Subsequence

Definition

A *subsequence* of a string X is a string that can be obtained by deleting some of the characters from X. Note that characters in the subsequence have the same order as they did in the given sequence X.

Finding the longest common subsequence:
Given two strings: $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$.
The problem is to find the longest string Z which is a subsequence of both X and Y.
Notation:
X_i for a given string X denotes the prefix of length i: $x_1, \ldots, x_i$.

Example

$X =$ `A L G O R I T H M`

`A G O T M` is a subsequence.

Problem instance. Given two sequences:

$X =$ `A L G O R I T H M`

$Y =$ `L O G A R I T H M`

What is the LCS?

Note, that LCS is not unique.

The structure of the optimal solution for LCS

Let $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$ be sequences with LCS $Z = z_1, \ldots, z_k$.
Then:

- If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
  For if $z_k \neq x_m = y_n$ we can simply append x_m to Z and obtain an LCS of X and Y with $k+1$, contradicting the assumption that Z was the LCS.
  Then it is clear that Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} because if there was an LCS W of X_{m-1} and Y_{n-1} more than $k-1$ characters we would simply append x_m to it to get an LCS of X and Y longer than Z (contradiction).

The structure of the optimal solution for LCS continued

For $x_m \neq y_n$ we have the following two cases:

- $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
  If $z_k \neq x_m$, then Z must be a common subsequence of X_{m-1} and Y. Furthermore, Z must be an LCS of X_{m-1} and Y since if there was a longer common subsequence of X_{m-1} and Y (say W with length greater than k), then since W is also a subsequence of both X and Y we would have a common subsequence of X and Y longer than Z (contradiction).
- $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}. (symmetric reasoning)

A DP Recurrence for LCS

Let $C[i,j]$ be the length of the LCS of X_i and Y_j.
Note, that $C[i,0] = C[0,j] = 0$ for all i,j.

$$C[i,j] = \begin{cases} C[i-1,j-1]+1, & \text{if } \ X[i] = Y[j], \\ \max\{C[i-1,j], C[i,j-1]\}, & \text{if } \ X[i] \neq Y[j]. \end{cases} \qquad (1)$$

Order of computations:
To compute the value of $C[i,j]$, we need to know the values of $C[i-1,j-1], C[i-1,j]$, and $C[i,j-1]$.
Filling in the cells from the top-most corner by rows will ensure that these values will be ready before computing $C[i,j]$.
Our answer for the length of an LCS will be found in $C[n,m]$.

LCS D.P. Pseudocode

```
// base cases
for i := 0 to m do C[i,0] := 0;
for j := 0 to n do C[0,j] := 0;
// filling the matrix
for i := 1 to m do
for j := 1 to n do
  if X[i] = Y[j] then C[i,j] := C[i-1, j-1] + 1;
  else C[i,j] := max(C[i-1,j], C[i,j-1]);
return C[m,n];
```

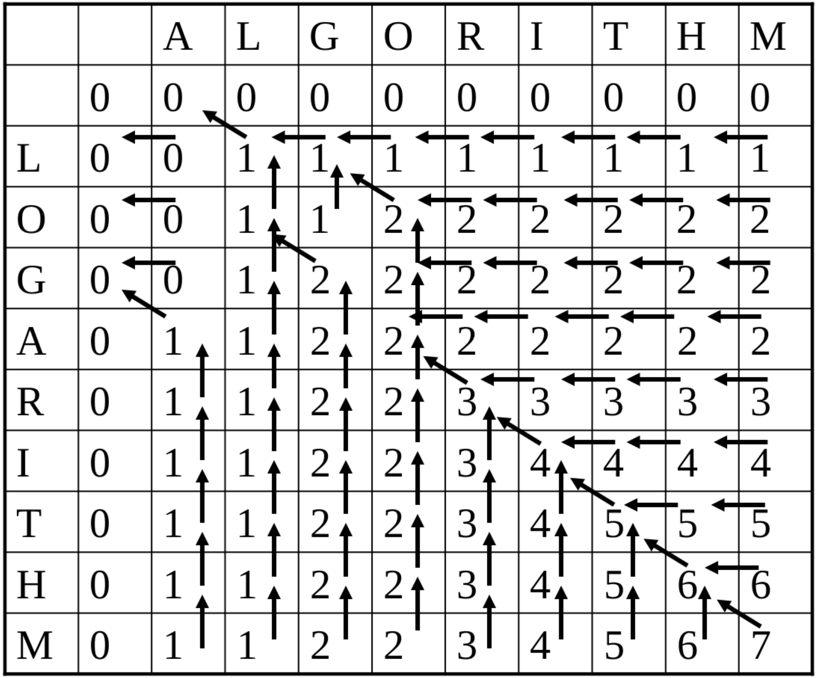
Complexity:
$\Theta(1)$ per entry, $\Theta(mn)$ entries, $\Theta(mn)$ total.

		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Recovering the Actual LCS.

Note that this pseudo-code only computed the length of the LCS.
To get the actual LCS:
For each entry, keep a backpointer $D[i,j]$ to one of $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$ (depending on which of the three choices determined $C[i,j]$).
We use D for Direction and give each direction a name:

$$D[i,j] = \begin{cases} \text{up-left if } \quad C[i,j] = 1 + C[i-1, j-1] \\ \text{up if } \quad C[i,j] = C[i-1, j] \\ \text{left if } \quad C[i,j] = C[i, j-1] \end{cases} \tag{2}$$

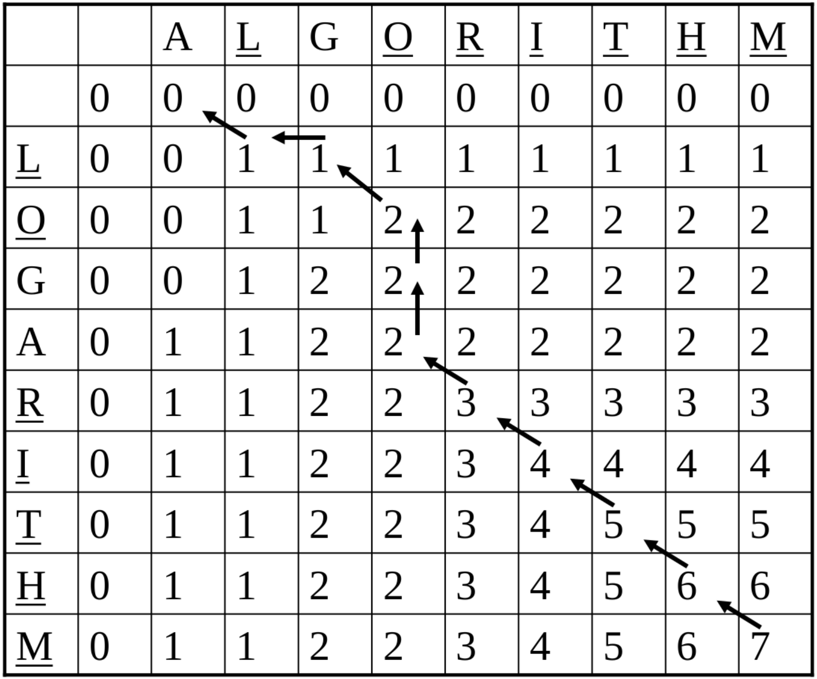


		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Using the Backpointers

We work backwards from D[m,n]:
When $D[i,j]$ is $(i-1, j-1)$, i.e. $up-left$, then $X[i] = Y[j]$ and this character is in the LCS.

```
row := m; col := n; lcs := "";
while (row > 0 and col > 0) do
  if (D[row,col] = upleft) then // X[row] = Y[col]
    lcs := lcs.X[row];
    row := row-1; col := col-1;
  else if (D[row,col] = up) then
    row := row-1;
  else if (D[row,col] = left) then
    col := col-1;
reverse lcs;
return lcs;
```



		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7