

Module 4

Dynamic Programming

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Introduction to Dynamic Programming

- Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping sub problems
- “Programming” here means “planning”
- Main idea:
 - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
 - solve smaller instances once
 - record solutions in a table
 - extract solution to the initial instance from that table
 - Dynamic programming can be interpreted as a special variety of space-and-time tradeoff (store the results of smaller instances and solve a larger instance more quickly rather than repeatedly solving the smaller instances more than once).
- Example: Fibonacci series 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
- $F(n) = F(n-1) + F(n-2)$, for $n > 1$. $F(0)=0$; $F(1) = 1$
 - $F(6) = F(5) + F(4)$.
 - $F(5) = F(4) + F(3)$. Note that we do not solve $F(4)$ twice. We find $F(4)$ only once and use that to compute $F(5)$ and $F(6)$.

Computing a binomial coefficient

Binomial coefficients are coefficients of the binomial formula:

$$(a + b)^n = C(n,0)a^n b^0 + \dots + C(n,k)a^{n-k}b^k + \dots + C(n,n)a^0 b^n$$

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$



$$C(n,0) = 1, \quad C(n,n) = 1 \text{ for } n \geq 0$$

Value of $C(n,k)$ can be computed by filling a table:

	0	1	2	...	$k-1$	k
0	1					
1	1	1				
.						
.						
.						
$n-1$					$C(n-1,k-1)$	$C(n-1,k)$
n						$C(n,k)$

$${}^n C_k = \frac{n!}{k! * (n-k)!}$$

Computing $C(12,5)$

		k 					
		0	1	2	3	4	5
n 	0	1					
	1	1	1				
	2	1	2	1			
	3	1	3	3	1		
	4	1	4	6	4	1	
	5	1	5	10	10	5	1
	6	1	6	15	20	15	6
	7	1	7	21	35	35	21
	8	1	8	28	56	70	56
	9	1	9	36	84	126	126
	10	1	10	45	120	210	252
	11	1	11	55	165	330	462
	12	1	12	66	220	495	792

Computing $C(n, k)$: pseudocode and analysis

ALGORITHM *Binomial*(n, k)

//Computes $C(n, k)$ by the dynamic programming algorithm

//Input: A pair of nonnegative integers $n \geq k \geq 0$

//Output: The value of $C(n, k)$

for $i \leftarrow 0$ **to** n **do**

for $j \leftarrow 0$ **to** $\min(i, k)$ **do**

if $j = 0$ **or** $j = i$

$C[i, j] \leftarrow 1$

else $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$

return $C[n, k]$

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$

Coin-Collecting Problem

- **Problem Statement**: Several coins are placed in cells of an $n \times m$ board, no more than one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location. When the robot visits a cell with a coin, it always picks up that coin. Design an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.
- **Solution**: Let $F(i, j)$ be the largest number of coins the robot can collect and bring to the cell (i, j) in the i th row and j th column of the board. It can reach this cell either from the adjacent cell $(i-1, j)$ above it or from the adjacent cell $(i, j-1)$ to the left of it.
- The largest numbers of coins that can be brought to these cells are $F(i-1, j)$ and $F(i, j-1)$ respectively. Of course, there are no adjacent cells to the left of the first column and above the first row. For such cells, we assume there are 0 neighbors.
- Hence, the largest number of coins the robot can bring to cell (i, j) is the maximum of the two numbers $F(i-1, j)$ and $F(i, j-1)$, plus the one possible coin at cell (i, j) itself.

Coin-Collecting Problem

Recurrence

$$F(i, j) = \max\{F(i - 1, j), F(i, j - 1)\} + c_{ij} \quad \text{for } 1 \leq i \leq n, \quad 1 \leq j \leq m$$

$$F(0, j) = 0 \quad \text{for } 1 \leq j \leq m \quad \text{and} \quad F(i, 0) = 0 \quad \text{for } 1 \leq i \leq n,$$

where $c_{ij} = 1$ if there is a coin in cell (i, j) and $c_{ij} = 0$ otherwise.

ALGORITHM *RobotCoinCollection*($C[1..n, 1..m]$)

//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at $(1, 1)$
//and moving right and down from upper left to down right corner

//Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0

//for cells with and without a coin, respectively

//Output: Largest number of coins the robot can bring to cell (n, m)

$F[1, 1] \leftarrow C[1, 1]$; for $j \leftarrow 2$ to m do $F[1, j] \leftarrow F[1, j - 1] + C[1, j]$

for $i \leftarrow 2$ to n do

$F[i, 1] \leftarrow F[i - 1, 1] + C[i, 1]$

 for $j \leftarrow 2$ to m do

$F[i, j] \leftarrow \max(F[i - 1, j], F[i, j - 1]) + C[i, j]$

return $F[n, m]$

Time Complexity: $\Theta(nm)$ Space Complexity: $\Theta(nm)$

Coin-Collecting Problem

- Tracing back the optimal path:
- It is possible to trace the computations backwards to get an optimal path.
- If $F(i-1, j) > F(i, j-1)$, an optimal path to cell (i, j) must come down from the adjacent cell above it;
- If $F(i-1, j) < F(i, j-1)$, an optimal path to cell (i, j) must come from the adjacent cell on the left;
- If $F(i-1, j) = F(i, j-1)$, it can reach cell (i, j) from either direction. Ties can be ignored by giving preference to coming from the adjacent cell above.
- If there is only one choice, i.e., either $F(i-1, j)$ or $F(i, j-1)$ are not available, use the other available choice.
- The optimal path can be obtained in $\Theta(n+m)$ time.

Coin-Collecting Problem: Ex-1

	1	2	3	4	5	6
1					5	
2		4		3		
3				2		7
4			8			2
5	9				6	














	1	2	3	4	5	6
1	0	0	0	0	5	5
2	0	4	4	7	7	7
3	0	4	4	9	9	16
4	0	4	12	12	12	18
5	9	9	12	12	18	18

Coin-Collecting Problem: Ex-1 (1)

	1	2	3	4	5	6
1	0	0	0	0	5	5
2	0	4	4	7	7	7
3	0	4	4	9	9	16
4	0	4	12	12	12	18
5	9	9	12	12	18	18

	1	2	3	4	5	6
1					5	
2		4		3		
3				2		7
4			8			2
5	9				6	

Coin-Collecting Problem: Ex-2

	1	2	3	4	5	6
1		7 				 4
2			 5	 3		
3		 8				 2
4	 4		 6		 1	
5	 9			 5		
6		 3			 7	

	1	2	3	4	5	6
1	0	7	7	7	7	11
2	0	7	12	15	15	15
3	0	15	15	15	15	17
4	4	15	21	21	22	22
5	13	15	21	26	26	26
6	13	18	21	26	33	33

Coin-Collecting Problem: Ex-2 (1)

	1	2	3	4	5	6
1	0	7	7	7	7	11
2	0	7	12	15	15	15
3	0	15	15	15	15	17
4	4	15	21	21	22	22
5	13	15	21	26	26	26
6	13	18	21	26	33	33

	1	2	3	4	5	6
1		7				4
2			5	3		
3		8				2
4	4		6		1	
5	9			5		
6		3			7	

Coin-Row Problem

- **Problem Statement:** There is a row of n coins whose values are some positive integers c_1, c_2, \dots, c_n , not necessarily distinct. The objective is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.
- **Recurrence:**
 - $F(n) = \text{Max}\{c_n + F(n-2), F(n-1)\}$ for $n > 1$
 - $F(0) = 0; F(1) = c_1$.

$\Theta(n)$ complexity for both time and space

ALGORITHM *CoinRow*($C[1..n]$)

```
//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array  $C[1..n]$  of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up
 $F[0] \leftarrow 0; F[1] \leftarrow C[1]$ 
for  $i \leftarrow 2$  to  $n$  do
     $F[i] \leftarrow \max(C[i] + F[i - 2], F[i - 1])$ 
return  $F[n]$ 
```

Example 1 for Coin-Row Problem

Solve for the coin-row problem for the instance {7, 9, 10, 9, 3, 5, 2}

Index	0	1	2	3	4	5	6	7
C		7	9	10	9	3	5	2
F	0	7	9	17	18	20	23	23
History		C1	C2	C3	C4	C5	C6	-
		-	F[0]	F[1]	F[2]	F[3]	F[4]	F[6]

F[0]	0			
F[1]	C1 = 7			
F[2]	Max(C2 + F[0], F[1]) = Max(9 + 0, 7) = 9			
F[3]	Max(C3 + F[1], F[2]) = Max(10 + 7, 9) = 17			
F[4]	Max(C4 + F[2], F[3]) = Max(9 + 9, 17) = 18			
F[5]	Max(C5 + F[3], F[4]) = Max(3 + 17, 18) = 20			
F[6]	Max(C6 + F[4], F[5]) = Max(5 + 18, 20) = 23			
F[7]	Max(C7 + F[5], F[6]) = Max(2 + 20, 23) = 23			

Dynamic Programming

Coins: C2, C4, C6; Value = 23

Greedy

C3, C1, C6 (in the dec. order of their values and non-overlap with neighboring coins);

Value = 22

Index	0	1	2	3	4	5	6	7
C		7	9	10	9	3	5	2
F	0	7	9	17	18	20	23	23
History		C1	C2	C3	C4	C5	C6	-
		-	F[0]	F[1]	F[2]	F[3]	F[4]	F[6]

Example 2 for Coin-Row Problem

Solve for the coin-row problem for the instance {7, 3, 9, 10, 8, 6}

Index	0	1	2	3	4	5	6
C		7	3	9	10	8	6
F	0	7	7	16	17	24	24
History		C1	-	C3	C4	C5	-
		-	F[1]	F[1]	F[2]	F[3]	F[5]

F[0]	0			
F[1]	C1 = 7			
F[2]	Max(C2 + F[0], F[1]) = Max(3 + 0, 7) = 7			
F[3]	Max(C3 + F[1], F[2]) = Max(9 + 7, 7) = 16			
F[4]	Max(C4 + F[2], F[3]) = Max(10 + 7, 16) = 17			
F[5]	Max(C5 + F[3], F[4]) = Max(8 + 16, 17) = 24			
F[6]	Max(C6 + F[4], F[5]) = Max(6 + 17, 24) = 24			

Dynamic Programming

Coins: C1, C3, C5; Value = 24

Greedy

C4, C1, C6 (in the dec. order of their values and non-overlap with neighboring coins);

Value = 23

Index	0	1	2	3	4	5	6
C		7	3	9	10	8	6
F	0	7	7	16	17	24	24
History		C1	-	C3	C4	C5	-
		-	F[1]	F[1]	F[2]	F[3]	F[5]

Example 3 for Coin-Row Problem

Solve for the coin-row problem for the instance {5, 1, 2, 10, 6, 2}

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	17
History		C1	-	C3	C4	-	C6
		-	F[1]	F[1]	F[2]	F[4]	F[4]

F[0]	0
F[1]	C1
F[2]	Max (F[1], F[0]+C2) = Max(5, 0+1) = 5 = F[1]
F[3]	Max (F[2], F[1]+C3) = Max(5, 5+2) = 7 = F[1]+C3
F[4]	Max (F[3], F[2]+C4) = Max(7, 5+10) = 15 = F[2]+C4
F[5]	Max (F[4], F[3]+C5) = Max(15, 7+6) = 15 = F[4]
F[6]	Max (F[5], F[4]+C6) = Max(15, 15+2) = 17 = F[4]+C6

The coins to be picked up are: C1 = 5, C4 = 10 and C6 = 2. The maximum value of the sum obtained is 17. The same set of coins get picked up even if we follow a greedy strategy.

Longest Common Subsequence (LCS) Problem

LCS Problem: Overview

- The LCS problem is to find the longest subsequence common to all sequences in a set of sequences (often just two).
- Note that a subsequence is different from a substring in the sense that a subsequence need not be consecutive terms of the original sequence.
- An algorithm for the LCS problem could be used to find the longest common subsequence between the DNA strands of two organisms.
- For a given length of the two DNA strands, the longer the common subsequence, the more similar and closer (evolutionarily) are the two organisms.
- Example: $X = \text{ATGCAC}$ $Y = \text{CAGATCA}$
 - $\text{LCS}(X, Y) = \text{ATCA}$.

LCS Problem: Idea

- Let the two sequences to compare be X of length m and Y of length n. We want to find the $LCS(X[1\dots m], Y[1\dots n])$.
- If $X[m] = Y[n]$, then we can simply discard the last character (that is common) from both the sequences and find the LCS of $X[1\dots m-1]$ and $Y[1\dots n-1]$, such that

$$LCS(X[1\dots m], Y[1\dots n]) = LCS(X[1\dots m-1], Y[1\dots n-1]) + 1.$$

- If $X[m] \neq Y[n]$, then the longest common subsequence of the two sequences can be at most either $X[m]$ or $Y[n]$; but not both. Hence, we can say that:

$$LCS(X[1\dots m], Y[1\dots n]) = \text{Max} \{LCS(X[1\dots m-1], Y[1\dots n]), LCS(X[1\dots m], Y[1\dots n-1])\}$$

Dynamic Programming Formulation

Define: $LCS[i][j]$ = Length of the LCS of sequence $X[1\dots i]$ and $Y[1\dots j]$

Thus, $LCS[i][0] = 0$ for all i

$LCS[0][j] = 0$ for all j

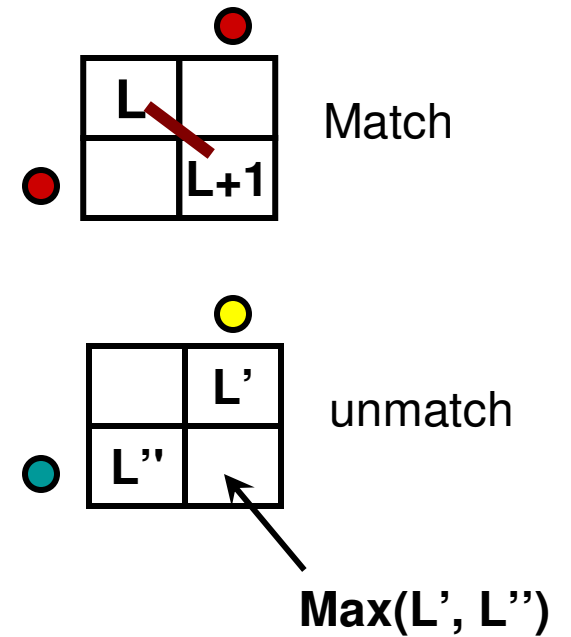
The goal is to find $LCS[m][n]$

$$LCS[i][j] = \begin{cases} LCS[i-1][j-1] + 1 & X[i] = Y[j] \\ \text{Max}\{LCS[i][j-1], LCS[i-1][j]\} & X[i] \neq Y[j] \end{cases}$$

X = ATGACTATAA
 Y = GACTAATA

LCS Example 1 (1)

		G	A	C	T	A	A	T	A
A	0	0	1	1	1	1	1	1	1
T	0	0	0	1	2	2	2	2	2
G	0	1	1	1	2	2	2	2	2
A	0	1	2	2	2	3	3	3	3
C	0	1	2	3	3	3	3	3	3
T	0	1	2	3	4	4	4	4	4
A	0	1	2	3	4	5	5	5	5
T	0	1	2	3	4	5	5	6	6
A	0	1	2	3	4	5	6	6	7
A	0	1	2	3	4	5	6	6	7



X = ATGACTATAA
 Y = GACTAATA

LCS Example 1 (2)

		G	A	C	T	A	A	T	A
A	0	0	1	1	1	1	1	1	1
T	0	0	0	1	2	2	2	2	2
G	0	1	1	1	2	2	2	2	2
A	0	1	2	2	2	3	3	3	3
C	0	1	2	3	3	3	3	3	3
T	0	1	2	3	4	4	4	4	4
A	0	1	2	3	4	5	5	5	5
T	0	1	2	3	4	5	5	6	6
A	0	1	2	3	4	5	6	6	7
A	0	1	2	3	4	5	6	6	7

A T G A C T - A T A A
 - - G A C T A A T - A

LCS: G A C T A T A

Ties are broken by going up

X = TGACTAC
Y = ACTGATGC

LCS Example 2 (1)

		T	G	A	C	T	A	C
		0	0	0	0	0	0	0
A		0	0	0	1	1	1	1
C		0	0	0	1	2	2	2
T		0	1	1	1	2	3	3
G		0	1	2	2	2	3	3
A		0	1	2	3	3	3	4
T		0	1	2	3	3	4	4
G		0	1	2	3	3	4	4
C		0	1	2	3	4	4	4
		0	1	2	3	4	4	5

X = TGACTAC
 Y = ACTGATGC

LCS Example 2 (2)

	T	G	A	C	T	A	C
A	0	0	0	1	1	1	1
C	0	0	1	2	2	2	2
T	0	1	1	2	3	3	3
G	0	1	2	2	3	3	3
A	0	1	2	3	3	4	4
T	0	1	2	3	4	4	4
G	0	1	2	3	4	4	4
C	0	1	2	3	4	4	5

T G A C T - A - - C
 - - A C T G A T G C

LCS: A C T A C

LCS Example 3 (1)

	C	A	A	G	T	A	C	G
	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1
C	0	1	1	1	1	1	2	2
T	0	1	1	1	2	2	2	2
G	0	1	1	1	2	2	2	3
G	0	1	1	1	2	2	2	3
A	0	1	2	2	2	3	3	3
G	0	1	2	2	3	3	3	4
C	0	1	2	2	3	3	4	4
A	0	1	2	3	3	4	4	4
T	0	1	2	3	4	4	4	4

X = CAAGTACG
Y = ACTGGAGCAT

LCS Example 3 (2)

	C	A	A	G	T	A	C	G
A	0	0	1	1	1	1	1	1
C	0	1	1	1	1	1	2	2
T	0	1	1	1	2	2	2	2
G	0	1	1	2	2	2	2	3
G	0	1	1	2	2	2	2	3
A	0	1	2	2	2	3	3	3
G	0	1	2	2	3	3	3	4
C	0	1	2	2	3	3	4	4
A	0	1	2	3	3	3	4	4
T	0	1	2	3	3	4	4	4

X = CAAGTACG
Y = ACTGGAGCAT

C A A G - T - - A C G - - -
- - A - C T G G A - G C A T

LCS: A T A G