

Dynamic Programming

CS 4231, Fall 2020

Mihalis Yannakakis

Dynamic Programming

- General method that applies to a class of problems, often a class of optimization problems (but not only optimization)
- Problem reduced to smaller (possibly overlapping) problems
- Main principles:
 1. Memoization: Do not solve the same problem instance repeatedly: Solve it once and record the result to reuse it if needed
Bottom-up (iterative) version: Problems are solved from smaller to larger and solutions tabulated
Top-down (recursive) version: Before initiating recursive call, check if solution was already computed previously.

Example: Fibonacci numbers

- $F_0 = 0, F_1 = 1$
 - $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$
 - Sequence: 0,1,1,2,3,5,8,13,21,...
-
- Recursive algorithm $\text{FIB}(n)$
 - if $n = 0$ then return 0
 - else if $n=1$ then return 1
 - else return $\text{FIB}(n-1) + \text{FIB}(n-2)$

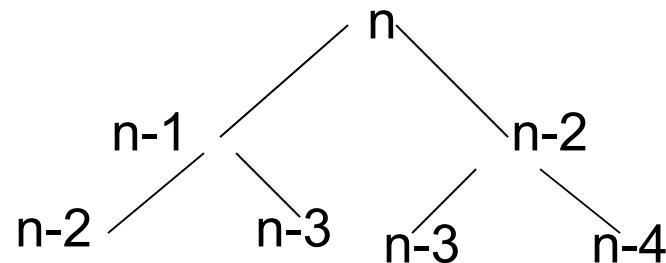
Complexity: $T(n) = T(n-1)+T(n-2)+O(1)$

Grows at least as fast as the Fibonacci numbers

$$F_n \approx \varphi^n / \sqrt{5}, \text{ where } \varphi = (1 + \sqrt{5})/2 = \text{golden ratio}$$

Fibonacci numbers ctd.

- **FIB** is called only on n distinct arguments, but it is called repeatedly with the same arguments



Tabulate the result, so no need to evaluate it again on the same argument \Rightarrow complexity $O(n)$

```
F[0]=0; F[1]=1;  
for i=2 to n do F[i]=F[i-1]+F[i-2]  
Return F[n]
```

(Actually, in this case we don't need an array: only need to remember the last two values)

Dynamic Programming for Optimization Problems

Main principles:

- Principle of Optimality (Optimal Substructure):

Problem can be reduced to a set of smaller subproblems;

Optimal solution for whole involves optimal solutions for subproblems

- Memoization:

Subproblems are solved from smaller to larger and solutions tabulated

Rod Cutting Problem

- Given rod of length n ,
- prices p_i , $i=1,\dots,n$, for rods of length i
- Find the optimal way to cut the rod of length n into smaller rods that maximizes the total revenue
- i.e., determine i_1, i_2, \dots, i_k such that $i_1 + i_2 + \dots + i_k = n$ and $p_{i_1} + p_{i_2} + \dots + p_{i_k}$ is maximized.

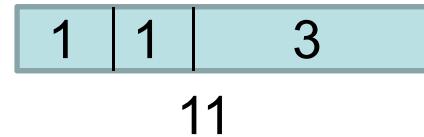
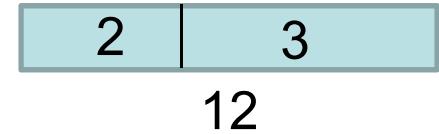
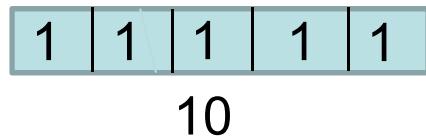
Example

- length i : 1 2 3 4 5
- price p_i : 2 5 7 9 11

- Rod of length 5



- Some possible cuttings and their revenue



Recurrence

- If first piece has length i , then remainder of length $n-i$ should be cut in an optimal way that maximizes revenue
(Principle of optimality)
- Let $r_j = \max$ revenue achievable from rod of length $j=0, 1, \dots, n$
- $r_n = \max\{ p_i + r_{n-i} \mid i = 1, \dots, n \}$
- $r_0 = 0$

Recursive Algorithm

- CUT-ROD(p,n)

if n=0 then return 0

else { q= -∞

 for i=1 to n do q =max(q, p[i] + CUT-ROD(p,n-i)) }

return q

Total number of calls:

$$f(n) = 1 + \sum_{j=0}^{n-1} f(j)$$
$$\Rightarrow f(n) = 2^n$$

Only n+1 different arguments \Rightarrow Many duplicate calls

Dynamic Programming algorithm

- Compute in array $r[0\dots n]$ the optimal revenues r_j according to the recurrence

$r[0]=0$

for $j=1$ to n do

{ $q = -\infty$

 for $i = 1$ to j do

$q = \max(q, p[i] + r[j-i])$

$r[j]=q$

}

return $r[n]$

Time Complexity: $\Theta(n^2)$

Reconstructing an optimal solution

- Record also in an array $s[1\dots n]$ the best choice for the length of the first piece for each $j=1,\dots,n$

$r[0]=0$

for $j=1$ to n do

{ $q = -\infty$

 for $i = 1$ to j do

 if ($q < p[i] + r[j-i]$) then { $q = p[i] + r[j-i]$; $s[j]=i$ }

$r[j]=q$

}

$j=n$

while $j>0$ do

{ print $s[j]$; $j=j-s[j]$ }

Top-Down DP Algorithm

- Like recursive algorithm but memoize in array $r[0\dots n]$ the optimal revenues r_j and only make recursive call if value is not available
- **MEMOIZED-CUT-ROD(p,n)**
for $i=0$ to n do $r[i] = -\infty$ [Initialization]
return M-CUT-ROD(p,n)
- **M-CUT-ROD(p,n)**
if $n=0$ then return 0
else { $q = -\infty$
 for $i=1$ to n do
 { if ($r[n-i] = -\infty$) $r[n-i] = M-CUT-ROD(p,n-i)$
 $q = \max(q, p[i] + r[n-i])$
 }
 }
return q

Time Complexity: $\Theta(n^2)$

0-1 Knapsack Problem

- n items, with given integer weights w_i , values v_i , $i=1,\dots,n$
- Knapsack with weight capacity W
- **Problem:** Choose a subset of items that fits in the knapsack and has maximum value
i.e, choose a subset $S \subseteq \{1,\dots,n\}$ that maximizes $\sum_{i \in S} v_i$
subject to $\sum_{i \in S} w_i \leq W$

Example

Item	Weight	Value	Knapsack capacity: 13
1	4	25	
2	6	30	
3	2	10	
4	5	27	
5	7	35	

Some feasible solutions, and their value

$\{1,2,3\}$: 65, $\{1,3,4\}$: 62, $\{1,3,5\}$: 70, $\{2,3,4\}$: 67,
 $\{2,5\}$: 65, $\{4,5\}$: 62,

Optimal solution: $\{1,3,5\}$, value 70

Reduction to smaller subproblems

- Should we take the n-th item?
 - If we take it, then, we have capacity $W-w_n$ left and can pick any subset from the first $n-1$ items → should pick the best
 - If we don't take it, we have capacity W for the first $n-1$ items
- Which of the two is better?

Let $M(b,i)$ = maximum value we can get with knapsack of capacity b , using a subset of the first i items only

$$M(b,i) = \max\{ M(b-w_i, i-1)+v_i, M(b,i-1) \} \text{ if } b \geq w_i \text{ else } = M(b,i-1)$$
$$M(b,0) = 0 \text{ for all } b$$

Recursive Algorithm

$\text{KNAP}(w, v, b, i)$

[max value that can be obtained for capacity b from first i items only]

if $i=0$ or $b=0$ then return 0

else if $b < w_i$ then return $\text{KNAP}(w, v, b, i-1)$

else return $\max(\text{KNAP}(w, v, b-w_i, i-1) + v_i, \text{KNAP}(w, v, b, i-1))$

- Main call: $\text{KNAP}(w, v, W, n)$

Time Complexity of Recursive algorithm

- A call for i items may generate two recursive calls with $i-1$ items.
- $T(i) = 2T(i-1) + O(1)$
- $\Rightarrow T(n) = \Theta(2^n)$
- But: if $W \ll 2^n$, then many of these recursive calls solve the same problem: at most nW different arguments.

Dynamic Programming Algorithm

```
for b=0 to W do M(b,0) =0  
for i=1 to n do  
    for b=0 to W do  
        if b≥wi and M(b-wi, i-1)+vi > M(b,i-1)  
            then M(b,i) = M(b-wi, i-1)+vi  
        else M(b,i) = M(b,i-1)  
Return M(W,n)
```

Running time: $O(nW)$

Reasonable if W is “small”.

Not a polynomial-time algorithm if weights given in binary.

Pseudopolynomial algorithm: polynomial if numbers are given in unary notation

Recovering an optimal solution

- Record which case generates $M(b,i)$ for every b,i

for $b=0$ to W do $M(b,0) = 0$

for $i=1$ to n do

 for $b=0$ to W do

 if $b \geq w_i$ and $M(b-w_i, i-1) + v_i > M(b, i-1)$

 then { $M(b,i) = M(b-w_i, i-1) + v_i$; $s(b,i)=1$ }

 else { $M(b,i) = M(b,i-1)$; $s(b,i) = 0$ }

Return $M(W,n)$ and s

Optimal Solution

$b=W$; $S = \emptyset$

for $i=n$ down to 1 do

 if $s(b,i)=1$ then { $S = S \cup \{i\}$; $b=b-w_i$ }

return S

Matrix Chain Multiplication

- Given a chain of n matrices A_1, A_2, \dots, A_n of dimensions $[p_0 \times p_1], [p_1 \times p_2], \dots, [p_{n-1} \times p_n]$ we want to multiply them using standard pairwise matrix multiplications
- Standard multiplication of a pxq times a qxr matrix has cost (#scalar multiplications) pqr (# additions also $\leq pqr$)
- Many ways to parenthesize the chain
- Different ways may have drastically different costs

Example

A_1, A_2, A_3 of dimensions $2 \times 100, 100 \times 2, 2 \times 100$

Solution 1: $(A_1 \times A_2) \times A_3$

Cost: $(2 \cdot 100 \cdot 2) + (2 \cdot 2 \cdot 100) = 800$

Solution 2: $A_1 \times (A_2 \times A_3)$

Cost: $(100 \cdot 2 \cdot 100) + (2 \cdot 100 \cdot 100) = 40,000$

Matrix Chain Multiplication Problem

- **Input:** Dimensions of the matrices A_1, A_2, \dots, A_n
i.e. $n+1$ numbers $p_0, p_1, p_2, \dots, p_{n-1}, p_n$
- **Output:** A parenthesization of the product of the matrices that minimizes the cost (#scalar multiplications)
- Note: # parenthesizations is exponential in n
- Relation with #binary trees with n leaves
Catalan number: $C(n-1) = \frac{1}{n} \binom{2n-2}{n-1} = \Omega(4^n/n^{3/2})$
- We cannot afford to try them all

Property1: Principle of Optimality (Optimal Substructure)

- If the last multiplication in the optimal solution is
$$(A_1 \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_n)$$
then both factors $(A_1 \times \cdots \times A_k)$ and $(A_{k+1} \times \cdots \times A_n)$ computed optimally

Recurrence relation for minimum cost of multiplying matrices A_i through A_j

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

Minimum overall cost: $m[1, n]$

Recursive Matrix Chain

RMC(p,i,j)

If $i=j$ then return 0

else { $m = \infty$

 for $k=i$ to $j-1$ do

 { $q = RMC(p,i,k) + RMC(p,k+1,j) + p_{i-1}p_kp_j$

 if $q < m$ then $m=q$

 }

}

Return m

Main: $RMC(p, 1, n)$

Recurrence relation for complexity of recursive algorithm

$$T(1) = \Theta(1)$$

$$T(n) = \sum_{k=1}^{n-1} (T(k) + T(n-k) + \Theta(1)) = 2 \sum_{k=1}^{n-1} T(k) + \Theta(n)$$

$$\Rightarrow T(n) = \Omega(3^n)$$

Too much!

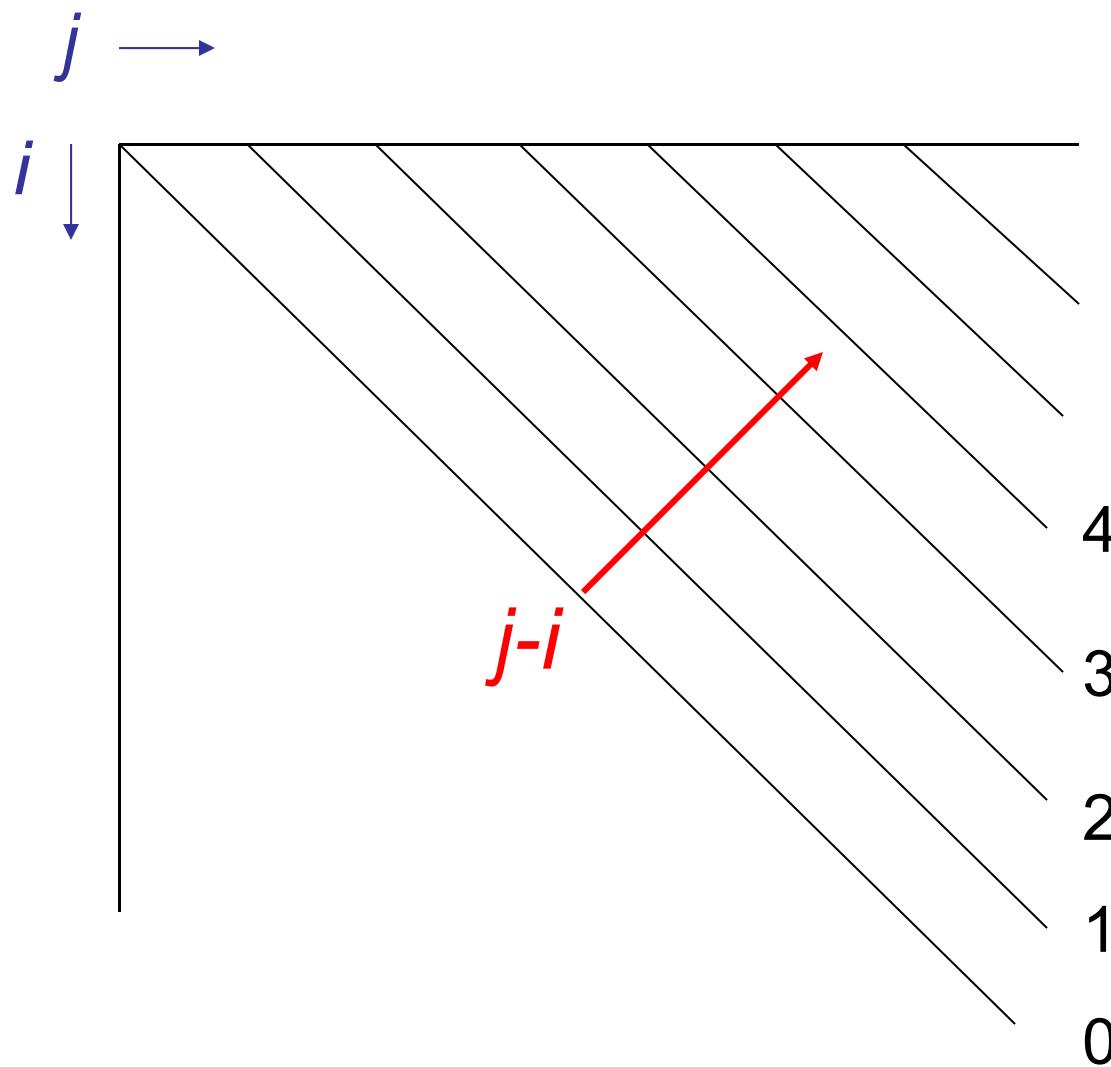
Same computations repeated over and over.

There is a better way.

Property 2: Many common subproblems → Memoization

- Only $O(n^2)$ subproblems $[i,j]$
- Solve each subproblem once and record the optimal cost in a table $m[i,j]$
- Compute the table in increasing order of the problem size = difference $j-i$
- Record for each pair i,j also the k that gave the optimal value, to trace back the optimal solution
- Time per pair i,j is $\Theta(j-i)$
- Total Complexity: $\Theta(n^3)$

Order of computation



Matrix Chain DP Algorithm

For $i=1$ to n do $m[i,i]=0$

For $d = 1$ to $n-1$ do /* $d=difference j-i */$

for $i=1$ to $n-d$ do

{ $j=i+d$

$m[i,j]=\infty$

for $k=i$ to $j-1$ do

{ $q = m[i,k]+m[k+1,j]+ p_{i-1}p_kp_j$

if $q < m[i,j]$ then { $m[i,j]=q$; $s[i,j]=k$ }

}

}

Return m and s

Complexity: $O(n^3)$

Recovering the optimal expression

Print-Opt(s,i,j)

If $i=j$ then print “ A_i ”

else { print “(“

 Print-Opt(s,i,s[i,j])

 print “ \times ”

 Print-Opt(s,s[i,j]+1,j)

 print “)”

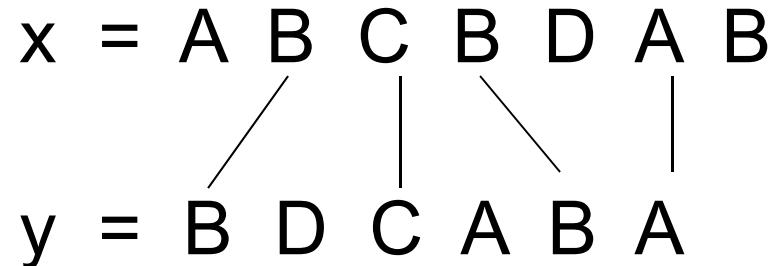
}

Main: Print-Opt(s,1,n)

Once we have computed s , this takes $O(n)$ time.

Longest Common Subsequence (LCS)

- Given two sequences $x[1\dots m]$, $y[1\dots n]$, find a longest common subsequence (gaps allowed)



- Applications: computational biology, diff
- Naive way: Take every subsequence of x check against $y \rightarrow$ Time $(2^m n)$

Optimal Substructure

- Subproblems i, j : Compute $\text{LCS}(x_1 \dots x_i, y_1 \dots y_j)$ and its length $c[i, j]$, for $i = 1, \dots, m$; $j = 1, \dots, n$
- Overall LCS has length $c[m, n]$
- Recurrence

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i, j - 1], c[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

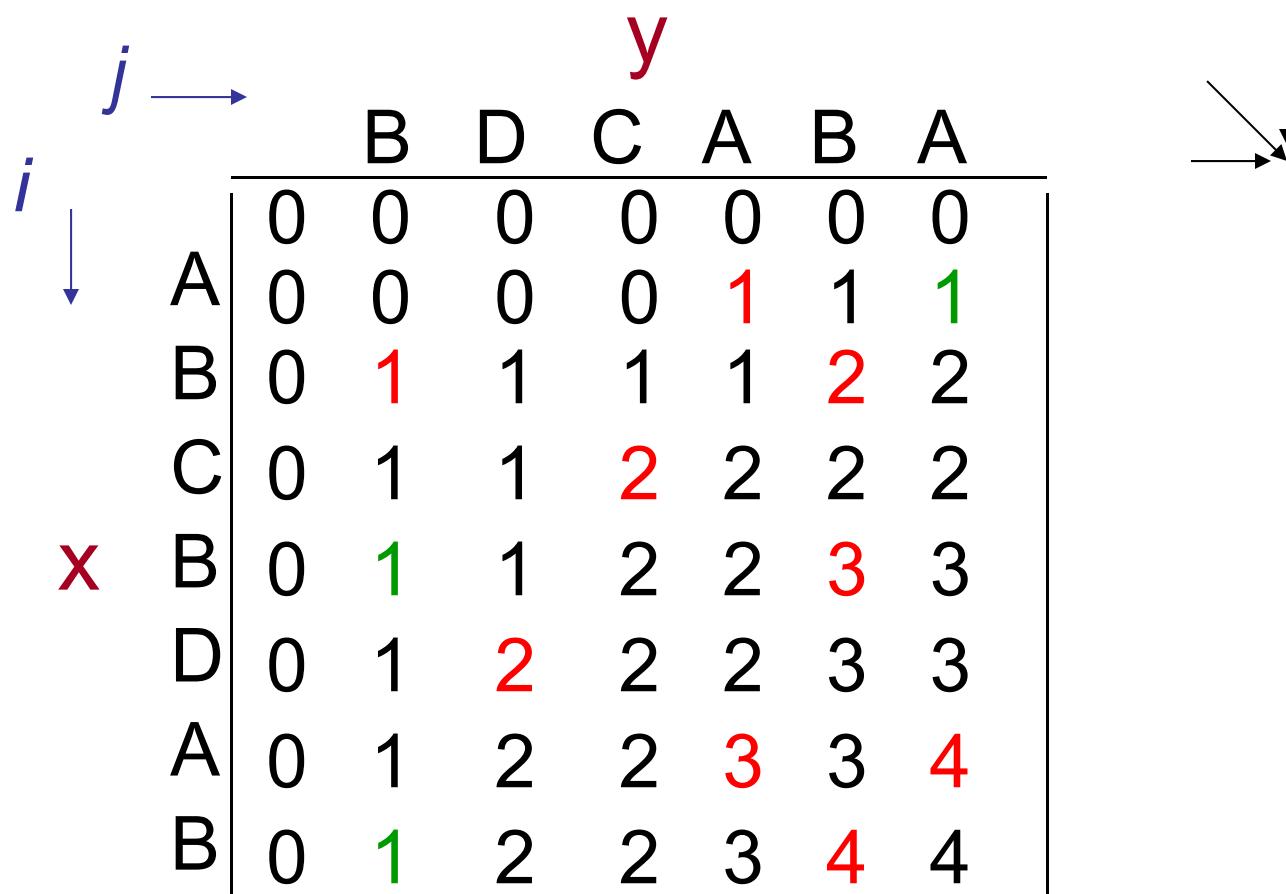
Proof: Look at LCS $z_1 z_2 \dots z_k$ of $(x_1 \dots x_i)$ and $(y_1 \dots y_j)$

Case 1: $x_i = y_j \Rightarrow z_k = x_i$ and $z_1 \dots z_{k-1} = \text{LCS}(x_1 \dots x_{i-1}, y_1 \dots y_{j-1})$

Case 2: $x_i \neq y_j, z_k \neq x_i \Rightarrow z_1 \dots z_k = \text{LCS}(x_1 \dots x_{i-1}, y_1 \dots y_j)$

Case 3: $x_i \neq y_j, z_k \neq y_j \Rightarrow z_1 \dots z_k = \text{LCS}(x_1 \dots x_i, y_1 \dots y_{j-1})$

Example



		y					
		B	D	C	A	B	A
		0	0	0	0	0	0
A		0	0	0	0	1	1
B		0	1	1	1	1	2
C		0	1	1	2	2	2
X		0	1	1	2	2	3
B		0	1	2	2	3	3
D		0	1	2	2	3	3
A		0	1	2	2	3	4
B		0	1	2	2	3	4

Order of computation

- Any order that is consistent with dependencies of recurrence: left to right, up to down, diagonally

