

Dynamic Programming

CIS 110, Spring 2012

University of Pennsylvania

Dynamic Programming

Dynamic programming records saves computation for reuse later.

- **Programming**: in the optimization sense (“Linear Programming”)
- **Dynamic**: “... it’s impossible to use [it] in a pejorative way.” (Richard Bellman)
- The name was designed to sound cool to RAND management and the US Department of Defense
- A more descriptive term is **look-up table**



Richard Bellman

Dynamic Programming

Dynamic programming records saves computation for reuse later.

- **Programming**: in the optimization sense (“Linear Programming”)
- **Dynamic**: “... it’s impossible to use [it] in a pejorative way.” (Richard Bellman)
- The name was designed to sound cool to RAND management and the US Department of Defense
- A more descriptive term is **look-up table**

When is dynamic programming useful?

- Exponential number of solutions
- Cost of solution is recursively computed
- Different solutions recursively compute same values



Richard Bellman

Dynamic Programming

Dynamic programming records saves computation for reuse later.

- **Programming**: in the optimization sense (“Linear Programming”)
- **Dynamic**: “... it’s impossible to use [it] in a pejorative way.” (Richard Bellman)
- The name was designed to sound cool to RAND management and the US Department of Defense
- A more descriptive term is **look-up table**

When is dynamic programming useful?

- Exponential number of solutions
- Cost of solution is recursively computed
- Different solutions recursively compute same values



Richard Bellman

The trick is exposing the common subproblems

Rod Cutting

Start with a rod of integer length n ...



... and cut it into several smaller pieces (of integer length).



Rod Cutting

Start with a rod of integer length n ...



... and cut it into several smaller pieces (of integer length).



Now suppose each length has a different value:



Rod Cutting

Start with a rod of integer length n ...



... and cut it into several smaller pieces (of integer length).



Now suppose each length has a different value:



How should we cut the rod into pieces?

- 2^{n-1} possibilities for a rod of length n

Rod Cutting

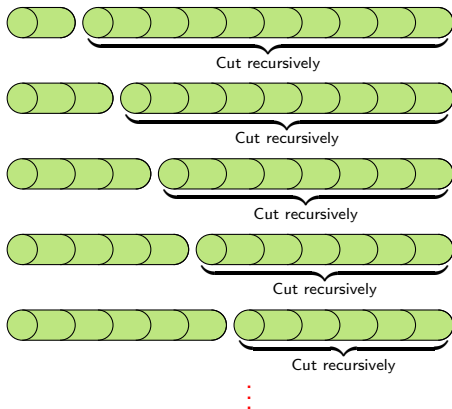
First rod-cutting strategy (brute-force):

- For every possible cut, compute the value of the left part plus the value of optimally cutting the right part. Take the best cut.

Rod Cutting

First rod-cutting strategy (brute-force):

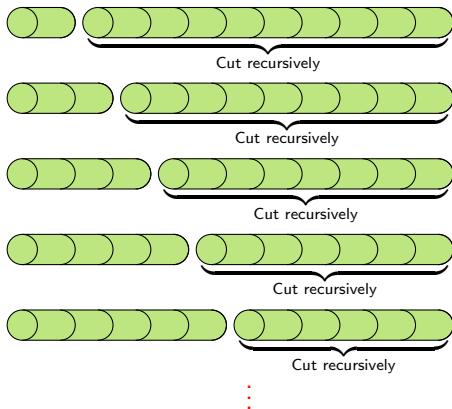
- For every possible cut, compute the value of the left part plus the value of optimally cutting the right part. Take the best cut.



Rod Cutting

First rod-cutting strategy (brute-force):

- For every possible cut, compute the value of the left part plus the value of optimally cutting the right part. Take the best cut.



- **Exponential number of recursive calls!**

Rod Cutting

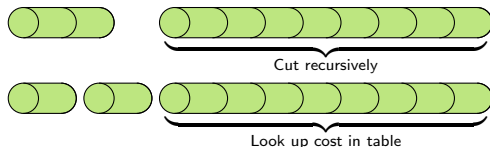
Second rod-cutting strategy (top-down):

- For every cut, compute value of left part and store it in a table
- Find value of optimal cut for right part in table
 - ▶ Compute it recursively if it doesn't exist yet

Rod Cutting

Second rod-cutting strategy (top-down):

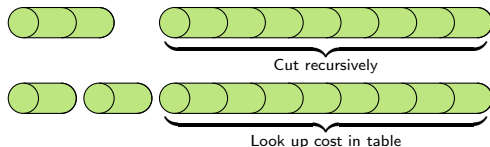
- For every cut, compute value of left part and store it in a table
- Find value of optimal cut for right part in table
 - ▶ Compute it recursively if it doesn't exist yet



Rod Cutting

Second rod-cutting strategy (top-down):

- For every cut, compute value of left part and store it in a table
- Find value of optimal cut for right part in table
 - ▶ Compute it recursively if it doesn't exist yet



- Reduces computation from $O(2^n)$ to $O(n^2)$ (Why?)
- Requires an array of length n to store intermediate computations

Rod Cutting

Third rod-cutting strategy (bottom-up):

- Compute the value of a rod of length 1. Store it.
- Compute the value of a rod of length 2. You can only cut it into rods of length 1. The value of a rod of length 1 is already computed, so there is no recursion.
- Compute the value of successively longer rods up to length n . The optimal values of shorter rods are always computed first so there is no recursion.

Sequence Matching

- Human genes are coded by four bases: Adenine (A), Thymine (T), Guanine (G), Cytosine (C)
- DNA undergoes mutations with each copy:
 - ▶ Substitutions: replace one base with another
 - ▶ Deletions: some bases are dropped

- Suppose we isolate a gene in a new organism:

A A C A G T T A C C

Predict function by comparing to genes in know, organism:

e.g. T A A G G T C A

- How similar are A A C A G T T A C C and T A A G G T C A?

Sequence Matching

How similar are A A C A G T T A C C and T A A G G T C A?

- How many mutations to change first sequence into second?
- How (un)likely is each mutation

Edit Distance: minimum cost to convert one string into another.

- Each change (mutation) has an associated cost:

Gap	2
Mismatch	1
Match	0

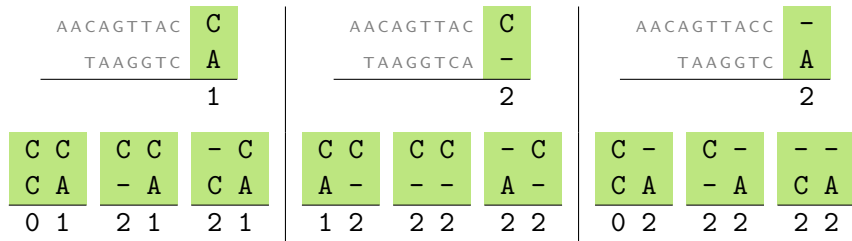
- Example matchings:

	A	A	C	A	G	T	T	A	C	C		
	T	A	-	A	G	G	T	C	A	-	-	
8	1	0	1	1	0	0	1	0	2	2		
	A	A	C	A	G	T	T	A	C	C		
	T	A	-	A	G	G	T	-	C	A		
	1	0	2	0	0	1	0	2	0	1	7	

Edit Distance

Brute-force recursive solution:

- Start at end of sequence and work backwards



- Recurse until we have all possible matches, then find minimum
- Three recursive calls per node $\Rightarrow O(3^n)$ matching cost
- We need to do better!

Edit Distance

Dynamic Programming recursive solution

- Consider a pair of characters in the middle:

```
A A C A G T T A C C
T A A G G T C A
```

- What is the cost of matching from this pair of Gs to the end?
 - ▶ Cost of matching Gs (0) + *lowest* cost of matching T T A C C to T C A.
 - ▶ Brute force solution computes *all possible* costs
- Idea: For each pair of characters, keep track of best match up to end

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A											2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Initialization:

- Cost of zero-length match (lower right) is zero
- Inserting a gap (move right or down in table) costs two

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A										1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A									3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A								4↘	3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A							6→	4↘	3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A						8→	6→	4↘	3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A					10→	8→	6→	4↘	3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A				12↘	10→	8→	6→	4↘	3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A			14→	12↘	10→	8→	6→	4↘	3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A		16↘	14→	12↘	10→	8→	6→	4↘	3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T											16↓
A											14↓
A											12↓
G											10↓
G											8↓
T											6↓
C											4↓
A	18↘	16↘	14→	12↘	10→	8→	6→	4↘	3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T	7↘	6↘	6↘	7↓	9↘	8↘	9↘	11↓	13↘	14↓	16↓
A	8↘	6↘	5↘	5↘	7↘	8↘	8↘	9↘	11↘	12↓	14↓
A	10↘	8↘	6→	4↘	5↘	6↘	7↘	7↘	9↘	10↓	12↓
G	12↘	10↘	8↘	6↘	4↘	4↘	5↘	6↘	7↘	8↓	10↓
G	13→	11→	9→	7→	5↘	4↘	3↘	4↘	5↘	6↓	8↓
T	15↘	13↘	11→	9→	7→	5↘	3↘	2↘	3↘	4↓	6↓
C	16→	14→	12↘	11↘	9↘	7↘	5↘	3→	1↘	2↘	4↓
A	18↘	16↘	14→	12↘	10→	8→	6→	4↘	3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Iteration:

- Work back from lower right
- Cost of cell (i, j) is

$$C(i, j) = \min(C(i + i, j) + 2, C(i, j + 1) + 2, C(i + 1, j + 1) + \delta)$$

where $\delta = 1$ if the i 'th character of string A and j 'th character of string B are identical.

Dynamic Programming

Idea: For each pair of characters, keep track of best match to end

	A	A	C	A	G	T	T	A	C	C	-
T	7↘	6↘	6↘	7↓	9↘	8↘	9↘	11↓	13↘	14↓	16↓
A	8↘	6↘	5↘	5↘	7↘	8↘	8↘	9↘	11↘	12↓	14↓
A	10↘	8↘	6→	4↘	5↘	6↘	7↘	7↘	9↘	10↓	12↓
G	12↘	10↘	8↘	6↘	4↘	4↘	5↘	6↘	7↘	8↓	10↓
G	13→	11→	9→	7→	5↘	4↘	3↘	4↘	5↘	6↓	8↓
T	15↘	13↘	11→	9→	7→	5↘	3↘	2↘	3↘	4↓	6↓
C	16→	14→	12↘	11↘	9↘	7↘	5↘	3→	1↘	2↘	4↓
A	18↘	16↘	14→	12↘	10→	8→	6→	4↘	3↘	1↘	2↓
-	20→	18→	16→	14→	12→	10→	8→	6→	4→	2→	0

Recovering the best alignment:

- Final cost is in cell (0,0)
- Follow arrows to reconstruct string
- → aligns letter in the current *column* with a gap
- ↓ aligns letter in the current *row* with a gap
- ↘ matches letters in current row and column with each other
- Total running time: $O(mn)$!

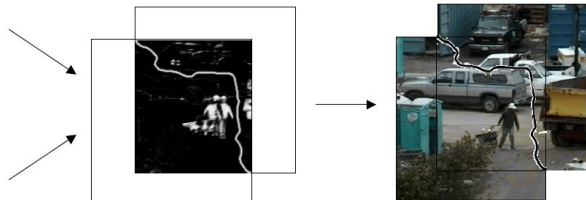
Some More Examples

- Image Compositing

- ▶ Given a set of overlapping images, what is the best way to stitch them?
- ▶ Cut the images along an “invisible” seam, and splice them together.
- ▶ The optimal seam can be found through dynamic programming.
 - ▶ Even better: shortest path



Davis98



Some More Examples

- Matrix parenthesization
 - ▶ Need to multiply a sequence of rectangular matrices
 - ▶ Which matrices should be multiplied first to minimize the number of operations

$$\left((a_1 \ a_2 \ a_3) \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \right) (c_1 \ c_2 \ c_3) = (a_1 b_1 + a_2 b_2 + a_3 b_3) (c_1 \ c_2 \ c_3)$$

$$(a_1 \ a_2 \ a_3) \left(\begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} (c_1 \ c_2 \ c_3) \right) = (a_1 \ a_2 \ a_3) \begin{pmatrix} b_1 c_1 & b_1 c_2 & b_1 c_3 \\ b_2 c_1 & b_2 c_2 & b_2 c_3 \\ b_3 c_1 & b_3 c_2 & b_3 c_3 \end{pmatrix}$$

Some More Examples

- Matrix parenthesization

- ▶ Need to multiply a sequence of rectangular matrices
- ▶ Which matrices should be multiplied first to minimize the number of operations

$$\left((a_1 \ a_2 \ a_3) \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \right) (c_1 \ c_2 \ c_3) = (a_1 b_1 + a_2 b_2 + a_3 b_3) (c_1 \ c_2 \ c_3)$$

$$(a_1 \ a_2 \ a_3) \left(\begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} (c_1 \ c_2 \ c_3) \right) = (a_1 \ a_2 \ a_3) \begin{pmatrix} b_1 c_1 & b_1 c_2 & b_1 c_3 \\ b_2 c_1 & b_2 c_2 & b_2 c_3 \\ b_3 c_1 & b_3 c_2 & b_3 c_3 \end{pmatrix}$$

- Seam carving (see demo)

- ▶ Shrink an image by finding one row or column of pixels to remove
- ▶ The seam doesn't have to be straight—it can wiggle
- ▶ Use dynamic programming to find the best set of pixels to remove