Dynamic Programming: Why?

Problem: duplicate computation, as seen for

- fibonacci, with equation

$$\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$$

- combinatorics, with equation

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Solution: compute solutions bottom-up

- $\mathrm{fib}(n)$: we have seen iterative algorithm
- $\binom{n}{k}$: Pascal's triangle

This can be generalized to numerous other problems.

Dynamic Programming: How?

General recipe: for a given problem, we find

- the space of relevant subproblems
- how subproblems depend on each other (no cycles)

Then a dynamic programming algorithm will

1. solve problems not depending on anything
2. solve problems only depending on problems solved in step 1
3. solve problems only depending on problems solved in step 1 or step 2
4. etc, etc
5. solve the original problem.

Assessment:

- $+$ each problem solved only once
- $-$ we may solve problems not needed
  (fix: top-down with memoization)

Giving Optimal Exact Change

Problem: for a given amount, give back

- coins that exactly add up to that amount
- but using as few as possible

Assumptions:

- unlimited supply of each denomination
- there are "pennies"

Easy "greedy" strategy:

1. give back as many quarters as possible
2. give back as many dimes as possible
3. give back as many nickels as possible
4. give back the rest in pennies.

What happens if nickels removed from circulation?

1. the above strategy no longer optimal
2. we will need to explore various options

Equations for Optimal Change

Problem: For amount A, using coins from denominations $d_1 \ldots d_n$, *find optimal exact change*.

Subproblems: for amount a with $a \in 0 \ldots A$, using coins from denominations $d_1 \ldots d_i$ with $i \in 0 \ldots n$,*find optimal exact change* where we use $c[i, a]$ for the number of coins needed. We have the equations:

- if $a = 0$ then $c[i, a] = 0$
- if $a > 0$ but $i = 0$ then $c[i, a] = \infty$
- if $a, i > 0$ but $d_i > a$ then $c[i, a] = c[i - 1, a]$

Otherwise, there are two options:

- use no coin from d_i; we then need $c[i - 1, a]$ coins
- use (at least) one coin from d_i; we then need $1 + c[i, a - d_i]$ coins

The algorithm should pick the smallest:
$c[i, a] = \min(c[i - 1, a], 1 + c[i, a - d_i])$

Dynamic Programming for Optimal Change

$$
\begin{aligned}
c[i,0] &= 0 \text{ when } 0 \le i \le n \\
c[0,a] &= \infty \text{ when } 0 < a \le A \\
c[i,a] &= c[i-1,a] \text{ when } d_i > a \\
c[i,a] &= \mathbf{min}(c[i-1,a],\ 1 + c[i,a-d_i]) \text{ when } d_i \le a
\end{aligned}
$$

Using these recurrences, $c[n,A]$ can be computed; to avoid duplicate computations we do it bottom-up:

```
for i ← 0 to n
    c[i,0] ← 0
    for a ← 1 to A
        if i = 0
            c[i,a] ← ∞
        else if d_i > a
            c[i,a] ← c[i−1,a]
        else
            c[i,a] ← min(c[i−1,a], 1 + c[i,a−d_i])
```

Space use is $\Theta(nA)$ and Running time is $\Theta(nA)$

Simulating Optimal Change

For $A = 8$ and $d_1 = 1; d_2 = 4; d_3 = 5$, we get

$i \backslash a$	0	1	2	3	4	5	6	7	8
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	1	2	3	4	5	6	7	8
2	0	1	2	3	1	2	3	4	2
3	0	1	2	3	1	1	2	3	2

We see that 2 coins suffice, but which?

- do we need 5-coins (from d_3)? since $c(3, 8)$ and $c(2, 8)$ are equal, no
- do we need 4-coins (from d_2)? since $c(2, 8)$ and $c(1, 8)$ are not equal, yes
- do we need further 4-coins? since $c(2, 4)$ and $c(1, 4)$ are not equal, yes and we are done since we now consider $c(2, 0)$.

Binary Knapsack Problem

We have n items (numbered $1..n$); each item i has a weight w_i and a value v_i (both positive). Our goal is to

- put as much value as possible into a knapsack
- while not exceeding its capacity W.

We shall consider the binary version: for each i we must

- either pick item i, letting $x_i = 1$
- or not pick item i, letting $x_i = 0$.

Then our goal is to maximize the value we carry:

$$\sum_{i=1}^{n} x_i v_i$$

while not exceeding capacity:

$$(\sum_{i=1}^{n} x_i w_i) \leq W$$

Solving the Binary Knapsack Problem

There is a simple solution: for each $S \subseteq \{1..n\}$ we

1. check if $\sum_{i \in S} w_i \leq W$
2. if so, compute $V_S = \sum_{i \in S} v_i$ and
    - if V_s is greater than the current maximum then let V_S be the new maximum

But as $\{1..n\}$ has 2^n subsets, this gives an exponential algorithm; can we do better?

- no one has found (and published) an algorithm always polynomial in n
- no one has found (and published) a proof that there does not exist such a polynomial algorithm.

Still, let's try to apply dynamic programming. For that purpose, assume that the weights are all integers.

Equations for Binary Knapsack

Problem: for capacity W, picking among items $1..n$, find maximum value.

Subproblems: for capacity w with $w \in 0..W$, picking among the items $1..i$ with $i \in 0..n$, find maximum value $V[i, w]$. We have the equations:

- if $i = 0$ then $V[i, w] = 0$
- if $w = 0$ then $V[i, w] = 0$
- if $i, w > 0$ but $w_i > w$ then $V[i, w] = V[i - 1, w]$

Otherwise, there are two options:

- do not pick item i; the value is then $V[i - 1, w]$.
- pick item i; the value is then $v_i + V[i - 1, w - w_i]$

The algorithm should pick the largest:
$V[i, w] = \max(V[i - 1, w], v_i + V[i - 1, w - w_i])$

Dynamic Programming for Binary Knapsack

$$
\begin{aligned}
V[0, w] &= 0 \text{ when } w \in 0..W \\
V[i, 0] &= 0 \text{ when } i \in 0..n \\
V[i, w] &= V[i-1, w] \text{ when } w_i > w \\
V[i, w] &= \mathbf{max}(V[i-1, w], v_i + V[i-1, w-w_i])
\end{aligned}
$$

Using these recurrences, $V[n, W]$ can be computed; to avoid duplicate computations we do it bottom-up:

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \textbf{ to } n \\
\quad \textbf{for } w \leftarrow 0 \textbf{ to } W \\
\quad\quad \textbf{if } i = 0 \texttt{ or } w = 0 \\
\quad\quad\quad V[i, w] \leftarrow 0 \\
\quad\quad \textbf{else if } w_i > w \\
\quad\quad\quad V[i, w] \leftarrow V[i-1, w] \\
\quad\quad \textbf{else} \\
\quad\quad\quad V[i, w] \leftarrow \mathbf{max}(V[i-1, w], \\
\quad\quad\quad\quad\quad\quad v_i + V[i-1, w-w_i])
\end{array}
$$

Assessment of Dynamic Programming

for $i \leftarrow 0$ **to** n
  for $w \leftarrow 0$ **to** W
    if $i = 0$ `or` $w = 0$
      $V[i, w] \leftarrow 0$
    else if $w_i > w$
      $V[i, w] \leftarrow V[i-1, w]$
    else
      $V[i, w] \leftarrow \mathbf{max}(V[i-1, w],$
        $v_i + V[i-1, w - w_i])$

We can retrieve solutions, backwards:

1. if $V[n, W] > V[n-1, W]$ we know item n is needed
2. if so, then if $V[n-1, W-w_n] > V[n-2, W-w_n]$ we know item $n-1$ is needed, ...

Space use is $\Theta(nW)$ and Running time is $\Theta(nW)$ which looks polynomial but is not as W may be unbounded.

All-Pair Shortest Path

Given a directed graph with

- n nodes, numbered $1..n$
- edges that have length

find, for each $i, j \in 1..n$, the path from i to j that is shortest (smallest sum of lengths, not fewest edges).

- we shall find the length of a shortest path, called $D(i,j)$ (but can then retrieve the path itself)
- we must require there are no negative cycles

Subproblems(Floyd-Warshall): for $i, j \in 1..n$, find $D_k(i,j)$, the length of the shortest path from i to j where all intermediate nodes belong to $1..k$ where $k \in 0..n$.

$$\begin{aligned}
D_0(i,i) &= 0 \\
D_0(i,j) &= d \text{ if there is an edge from } i \text{ to } j \text{ with length } d \\
D_0(i,j) &= \infty \text{ if there is no edge from } i \text{ to } j
\end{aligned}$$

Shortest Path: Key Recurrence

To find the recurrences for $D_k(i,j)$ when $k > 0$, observe that paths with intermediate nodes in $1..k$ either:

- do not have node k as intermediate node; a shortest such path has length
$$D_{k-1}(i,j)$$
- do have node k as intermediate node (but only once); such paths are composed by a path from i to k followed by a path from k to j, and hence a shortest such path has length
$$D_{k-1}(i,k) + D_{k-1}(k,j).$$

We conclude that we have the recurrence:

$$D_k(i,j) = \mathbf{min}(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j))$$

Dynamic Programming for Shortest Path

```
for k ← 1 to n
    for i ← 1 to n
        for j ← 1 to n
            v ← D_{k−1}[i,k] + D_{k−1}[k,j]
            if v < D_{k−1}[i,j]
                D_k[i,j] ← v
            else
                D_k[i,j] ← D_{k−1}[i,j]
```

Running time: $\Theta(n^3)$

- Could we let i run from n to 1? Yes!
- Could we let k run from n to 1? No!
- Could the loop for i be inside the loop for j? Yes!
- Could we let the loop for k be the inner loop? No!

Space Use: naively $\Theta(n^3)$ but an easy optimization gives $\Theta(n^2)$ since it suffices to have two copies of D, and actually one is enough since $D_k(i,k) = D_{k-1}(i,k)$, etc.

Floyd-Warshall's Algorithm for Shortest Path

```
for i ← 1 to n
    for j ← 1 to n
        if i = j
            D[i,i] ← 0
        else if there is an edge from i to j
            D[i,j] ← L(i,j)
        else
            D[i,j] ← ∞
for k ← 1 to n
    for i ← 1 to n
        for j ← 1 to n
            v ← D[i,k] + D[k,j]
            if v < D[i,j]
                D[i,j] ← v
                // record that shortest path
                // from i to j goes thru k
```

Multiplying Chain of Matrices

We shall again look at matrix multiplication.

- previously: multiply two large square matrices
- now: multiply a chain of matrices of any size and shape.

Key Question: in which order to do the multiplications?

- commutativity does not hold; we cannot swap
- associativity holds, we can rearrange parentheses.

Thus we may compute ABC as

$$(AB)C \text{ or } A(BC)$$

and may compute $ABCD$ as one of

$$(AB)(CD),\ ((AB)C)D,\ (A(BC))D,\ A(B(CD)),\ A((BC)D)$$

While all approaches give the same result, their performances may vastly differ!

Counting Multiplications

Given matrices $A : p \times q$ and $B : q \times r$, AB is $p \times r$ with

$$(AB)_{ij} = \sum_{k=1}^{q} A_{ik} B_{kj} \text{ for } i \in 1..p, j \in 1..r$$

which requires pqr (integer) multiplications, the cost of the operation.

With C an $r \times s$ matrix, we can compute ABC in 2 ways:

1. $(AB)C$, with multiplications required: $pqr + prs$
2. $A(BC)$, with multiplications required: $qrs + pqs$

Example: if $p = r = 2$ and $q = s = 100$ then

1. the first way has cost $400 + 400 = 800$
2. the second way has cost $20,000 + 20,000 = 40,000$

Example: if $p = 3$, $q = 2$, $r = 6$, $s = 4$ then

1. the first way has cost $36 + 72 = 108$
2. the second way has cost $48 + 24 = 72$

While 1 was initially cheaper, 2 is better.

General Problem

Given dimensions $d_0..d_n$, find lowest cost of multiplying $A_1 \dots A_n$ where each A_i is a $d_{i-1} \times d_i$ matrix.

Subproblems: for i, j with $1 \le i \le j \le n$, find lowest cost $M[i,j]$ of multiplying $A_i \dots A_j$.

1. if $j = i$ then $M[i,j] = 0$
2. if $j = i + 1$ then $M[i,j] = d_{i-1} d_i d_{i+1}$.

To find the recurrence for $M[i,j]$ when $j > i$, observe that a multiplication order for $A_i \dots A_j$ will be of the form

$$(A_i \dots A_k)(A_{k+1} \dots A_j) \text{ where } i \le k < j$$

whose last multiplication is $d_{(i-1)} \times d_k$ with $d_k \times d_j$; thus

$$M[i,j] = \min_{k \in i \dots j-1}(M[i,k] + M[k+1,j] + d_{i-1} \cdot d_k \cdot d_j)$$

which when $j = i + 1$ gives recurrence 2.

Algorithm for Chained Matrix Multiplication

We must compute entries before we refer to them, as in:

for $i \leftarrow n$ **downto** 1
    $M[i,i] \leftarrow 0$
    for $j \leftarrow i+1$ **to** n
        // compute $M[i,j]$
        $M[i,j] \leftarrow \infty$
        for $k \leftarrow i$ **to** $j-1$
            $v \leftarrow M[i,k] + M[k+1,j] + d_{i-1} \cdot d_k \cdot d_j$
            if $v < M[i,j]$
                $M[i,j] \leftarrow v$
                // record: best way to multiply $A_i..A_j$
                // is to do $(A_i..A_k)(A_{k+1}..A_j)$

Space use is $\Theta(n^2)$ and Running time is

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\Theta(j-i+1) = \sum_{i=1}^{n}\Theta((n-i+1)^2) \in \Theta(n^3)$$

which is time for scheduling, not for the multiplications!

Shortest vs Longest Paths

Assume that π is a shortest path from A to C, with

$$\pi = A \xrightarrow{\pi_1} B \xrightarrow{\pi_2} C$$

Then a shortest path from A to B is π_1, for if π_1' is a shorter path we would have a shorter path from A to C:

$$\pi' = A \xrightarrow{\pi_1'} B \xrightarrow{\pi_2} C$$

An optimal solution for shortest path can be composed of optimal solutions for subproblems

Next assume that π is a longest simple (cycle-free) path from A to C, and that we can write

$$\pi = A \xrightarrow{\pi_1} B \xrightarrow{\pi_2} C$$

Is π_1 then the longest simple path from A to B?

- no, as easy counterexample
- previous proof fails: if π_1' is a longer simple path then $\pi_1'\pi_2$ is longer than $\pi_1\pi_2$ but may not be simple.

Thus solutions can not be immediately composed.