Dynamic Programming
Announcements

Friday is a guest lecture in GUG 220
- Kendra Yourtee will give insider information on tech interviews
- Will not be covered on the final – will be very useful for jobs though!
- Don’t go to Gowen Hall on Friday – we won’t be there!

Final Homework will be posted tonight!
- Short (2 question) FINAL REVIEW
- Due Wed. before final
Goals for Today

3 examples of dynamic programming – the details of the first two are not important – it is the strategy that I want you to focus on

Learning goal 1: Be able to state the steps of designing a dynamic program

Learning goal 2: Be able to implement the Floyd-Warshall all-shortest-paths algorithm.

Learning goal 3: Given a description of a problem and how it is broken into subproblems, be able to write a dynamic program to solve the problem.
Coin Changing Problem (1)

THIS IS A VERY COMMON INTERVIEW QUESTION!

Problem: I have an unlimited set of coins of denominations \( w[0], w[1], w[2], \ldots \) I need to make change for \( W \) cents. How can I do this using the minimum number of coins?

Example: I have pennies \( w[0] = 1 \), nickels \( w[1] = 5 \), dimes \( w[2] = 10 \), and quarters \( w[3] = 25 \), and I need to make change for 37 cents.

I could use 37 pennies (37 coins), 3 dimes + 1 nickels + 2 pennies (6 coins), but the optimal solution is 1 quarter + 1 dime + 2 pennies (5 coins).

We want an algorithm to efficiently compute the best solution for any problem instance.
Step 1: Find the subproblems

What are our subproblems? How do we use them to compute a larger solution?

One way to make the problem “smaller” is to reduce the number of cents we are making change for.

Let $\text{OPT}(W)$ denote the optimal number of coins to use to make change for $W$ cents.
Step 2: “Characterize the Optimum”

What recurrence relation describes our optimum solution? What are the base cases?

Break the problem into cases. Any non-zero amount will use at least one coin, so we can cover all of our cases by:

1) use at least one penny
2) use at least one nickel
Etc.

i) use at least one of \( w[i] \)

So in the \( i \)'th case, if \( \text{OPT}(W) \) uses \( w[i] \), then

\[
\text{OPT}(W) = \text{OPT}(W - w[i]) + 1
\]

or overall:

\[
\text{OPT}(W) = \min\{ \text{OPT}(W - w[1]) + 1, \text{OPT}(W - w[2]) + 1, \ldots \text{OPT}(W - w[m]) + 1 \}
\]

For our base cases, we know that it takes 0 coins to make change for 0 cents:

\( \text{OPT}(0) = 0 \)

We also know that it is impossible to make negative change

\( \text{OPT}(n) = \text{infinity} \) for \( n < 0 \)
Step 3: Order the Subproblems

We have characterized our optimum solution:

\[
OPT(W) = \begin{cases} 
\infty & \text{if } W < 0 \\
0 & \text{if } W = 0 \\
\min_i OPT(W - w[i]) + 1 & \text{otherwise}
\end{cases}
\]

What order do we solve these in?

Notice that the recursive case depends only on smaller values of \( W \).
Therefore we can solve from smallest to largest: from 1 to \( W \).
Step 4: Write the algorithm

change(W, w[]): // w[] has length n

OPT = new array[W + 1]

OPT[0] = 0 // Base case

for i = 1 to W:

best = infinity

for j = 0 to n:

if (i - w[j]) >= 0 && OPT[i - w[j]] + 1 < best:

best = OPT[i - w[j]] + 1

OPT[i] = best

return OPT[W]
Which coins did we use?

This algorithm only tells us how many coins we need to use, not which coins they were.

Each time we found $OPT(k)$, we made a choice about which coin we were adding (see why)?
- The coin we “removed” to find the best subproblem in the top-down view is a coin “added” when viewed bottom-up.

**Idea:** Use a second array to keep track of which coins we are adding!
Step 5: Tracking Coins

change(W, w[]): // w[] has length n

OPT = new array[W + 1]
coins = new array[W+1]
coins[0] = -1
OPT[0] = 0
for i = 1 to W:
    best = infinity
    bestCoin = -1
    for j = 0 to n:
        if (i – w[j]) >= 0 && OPT[i – w[j]] + 1 < best:
            best = OPT[i – w[j]] + 1
            bestCoin = j
    OPT[i] = best
    coins[i] = bestCoin
return coins
Coin changing problem (2)

Same setup: How many different ways are there of making change? (Counting problem)

This time we’ll need both size variables – the amount of change to make, and the coins available:

$$\text{OPT}(W, k) := \text{The number of ways to make change for } W, \text{ using only the first } k \text{ coin types}$$

e.g. if \( w[0] = \text{pennies}, w[1] = \text{nickels}, w[2] = \text{dimes}, \) and \( w[3] = \text{quarters}, \)

$$\text{OPT}(12, 2) = 3; \text{ the number of ways to make 12 cents using only pennies and nickels}$$
Characterizing the Optimum

For our base cases, we know that there is only one way to make 0 cents (no coins):
\[ \text{OPT}(0, k) = 1 \text{ for all } k \]

There are 0 ways to make change with 0 coins (for non-zero amounts of change):
\[ \text{OPT}(W, 0) = 0 \text{ for all } W \neq 0 \]

Recursive Case: If we are making change with the first \( k \) coin types, we can use the \( k' \)th type of coin \( 0 \) times, 1 time, 2 times, ..., up to \( \lfloor W / w_{k-1} \rfloor \) times (remember the \( k' \)th coin is \( w_{k-1} \)).

The remainder of the money needs to be made up of the other coin types, so we have
\[
\text{OPT}(W, k) = \sum_{i=0}^{\lfloor W / w_{k-1} \rfloor} \text{OPT}(W - i \cdot w_{k-1}, k - 1)
\]
Ordering the Subproblems

We now have 2 variables, W and k, so our array will be 2D:

\[
\text{OPT}(W,k) = \sum_{i=0}^{\left\lfloor \frac{W}{w[k-1]} \right\rfloor} \text{OPT}(W - i \cdot w[k-1], k-1)
\]

Subproblems depended on only have smaller W and k values.
Algorithm

changeCounting(W, w[]): // w[] has length m

  OPT[][] = new int[][]
  OPT[0][k] = 0 for all k
  for k = 1…m:
    for n = 1…W:
      numWays = 0
      for i = 0… W / w[k-1]:
        numWays += OPT[n – i*w[k-1], k-1]
      OPT[n, k] = numWays
  return OPT(W, m)

O(m W^2)

# of coin types

loop over # of coins

loop over # of cents

O(w)

CSE 373 SU 18 – BEN JONES
All Shortest Paths

Given a graph G, find the length of the shortest path between every pair of vertices.

Looks like $\text{OPT}(i,j) := \text{length of shortest path from } v_i \text{ to } v_j$

How to break this into smaller problems?

Borrow a trick from the last example: introduce a restriction:

$\text{OPT}(k; i,j) := \text{length of shortest path from } v_i \text{ to } v_j \text{ using only the first } k \text{ vertices as intermediate nodes } (v_0, v_1, v_2, \ldots, v_{k-1})$
Characterizing OPT

OPT(k,i,j) := shortest path from i to j using only first k vertices in between

What is OPT(3, 0, 4) for this graph? What path does it correspond to?

OPT(3, 0, 4) = 9 since we can’t use 3 as an intermediate node.
Characterizing OPT

\[ \text{OPT}(k,i,j) := \text{shortest path from } i \text{ to } j \text{ using only first } k \text{ vertices in between} \]

Observation: \( \text{OPT}(k,i,j) \) either uses the \( k^{th} \) vertex, or it doesn’t:

\[
\text{OPT}(k,i,j) = \min \left\{ \begin{array}{l}
\text{OPT}(k-1, i, j), \\
\text{OPT}(k-1, i, k) + \text{OPT}(k-1, k, j)
\end{array} \right. 
\]
Characterizing OPT

\[ \text{OPT}(k, i, j) = \min \{ \text{OPT}(k-1, i, j), \text{OPT}(k-1, i, k) + \text{OPT}(k-1, k, j) \} \]

Base cases?

The path from a vertex to itself has length 0:

\[ \text{OPT}(k, i, i) = 0 \]

A path with no intermediate vertices is only possible if the edge i->j exists:

\[ \text{OPT}(0, i, j) = w_{ij} \text{ if i->j exists, otherwise } \infty \]
Ordering the Subproblems

\[ OPT(k, i, j) = \begin{cases} 
0 & \text{if } i = j \\
\min\{OPT(k - 1, i, j), OPT(k - 1, i, k) + OPT(k - 1, k, j)\} & \text{otherwise} \\
w_{ij} & \text{if } k = 0 
\end{cases} \]

What order should we use?

Q: Which subproblems do we depend on in the recursive case?

A: Lower values of k, and the same values of \(i\) and \(j\).

So if we order our subproblems in increasing order of k, we will always have the subproblems we need solved!

OPTIMIZATION: Since we only use one lower k value, we can re-use the same array for each iteration of \(k\).
Floyd-Warshall Algorithm

shortestPaths(G):

let $d[][]$ be a $|V| \times |V|$ matrix

$d[i][j] = w(i,j)$ or infinity if no edge ($w(i,i) = 0$ for all $i$)

for $k = 0 \ldots |V| - 1$ :

for $i = 0 \ldots |V| - 1$:

for $j = 0 \ldots |V| - 1$:

if $(d[i][j] + d[k][j] < d[i][j])$:

$d[i][j] = d[i][k] + d[k][j]$

return $d$
Example

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Distances

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Path Reconstruction

shortestPaths(G):

let $d[][]$ be a $|V| \times |V|$ matrix

let $\text{path}[][]$ be a $|V| \times |V|$ matrix initialized to -1s

d[i][j] = w(i,j) or infinity if no edge (w(i,i) = 0 for all i)

for k=0 ... |V| - 1:

for i = 0 ... |V| - 1:

for j = 0 ... |V| - 1:

if (d[i][j] + d[k][j] < d[i][j]):

\[ d[i][j] = d[i][k] + d[k][j] \]

\[ \text{path}[i][j] = k \]

return d