Dynamic Programming

Data Structures and Algorithms

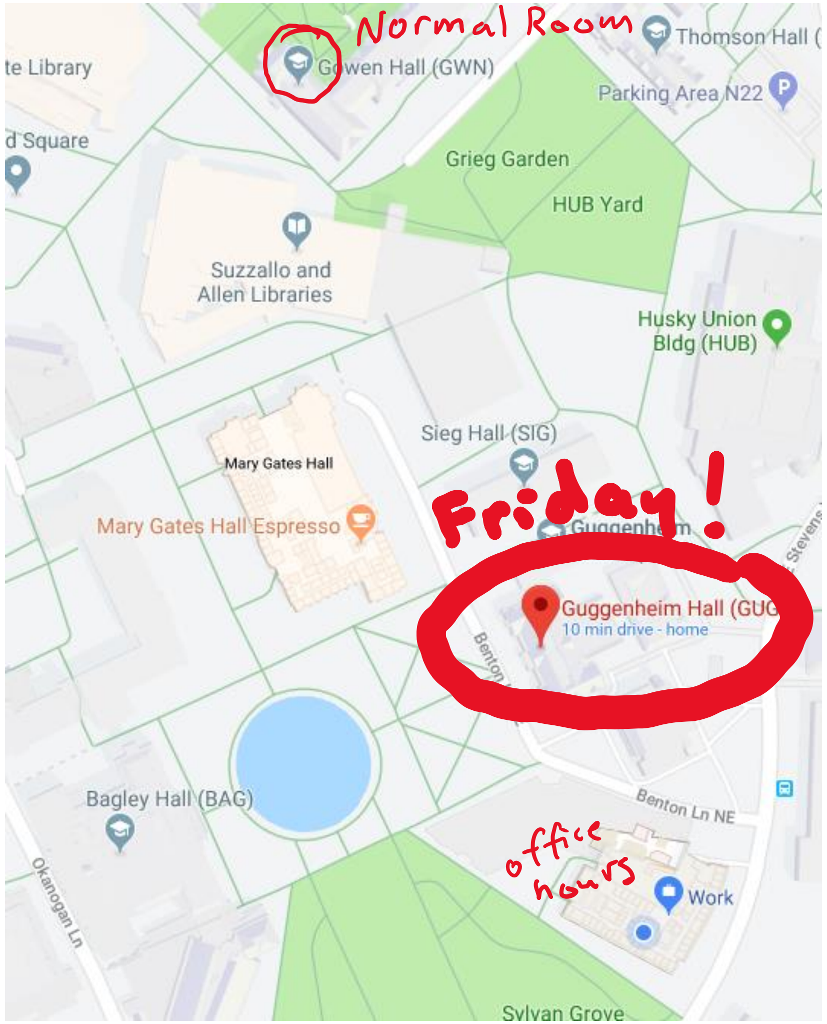
Announcements

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 denomitations w[0], w[1], w[2], … 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 OPT(W) denote the optimal number of coins to use to make change for W cents.

Step 2: "Characterize the Optimum"

define it as a recurrence relation

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 OPT(W) uses w[i], then $\boxed{\text{OPT(W)}}$ = OPT($\boxed{\text{W – w[i]}}$) + 1

or overall: OPT(W) = min{ OPT(W – w[1]) + 1, OPT(W – w[2]) + 1, … OPT(W – w[m]) + 1}

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

OPT(0) = 0

We also know that it is impossible to make negative change

OPT(n) = infinity for n < 0

OPT(5)

nickels

5-1

OPT(4)+1

OPT(5-5)

$\boxed{\text{OPT(0) + 1}}$

OPT(5-10)

OPT(3)

OP(3-5) = OPT(-2)

Step 3: Order the Subproblems

We have characterized our optimum solution:

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

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

look at what subproblems the recursive case depends on…; future values depends on smaller values;
so we should fill from the smaller W values.

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 (W – w[j]) >= 0 && OPT[W – w[j]] + 1 < best:
        best = OPT[W – w[j]] + 1
    OPT[i] = best
  return OPT[W]
```

Base case – since index < 0, used a conditional instead

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 (W – w[j]) >= 0 &&  OPT[W – w[j]] + 1 < best:
                best = OPT[W – w[j]] + 1
                bestCoin = j
        OPT[i] = best
        coins[i] = bestCoin
    return coins
```

O(1) / O(W)

W

n

O(1)

O(nW)

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:

which coins can we use

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

e.g. if w[0] = pennies, w[1] = nickels, w[2] = dimes, and w[3] = quarters,

OPT (12, 2)

can use coins denominations at w[0] thru w[k-1]

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

Characterizing the Optimum

37

dimes

i = # of dimes

For our base cases, we know that there is only one way to make 0 cents (no coins):

OPT(0, k) = 1 for all k

OPT(37, 3) = OPT(37, 2) +
OPT(27, 2) +
OPT(17, 2) +
OPT(7, 2)

There are 0 ways to make change with 0 coins (for non-zero amounts of change):

OPT(W,0) = 0 for all W != 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 W / w[k-1] 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

If we use k - 1 coins,
w[k - 1] is the denomination of the first
coin we DONT use. So select i
of these new coins, and find the number
of ways to make change for W - i * denomination
using the first k - 1 coins (a subproblem we already
solved); sum all these up

denom.

$$\mathrm{OPT}(\mathrm{W},\mathrm{k}) = \sum_{i=0}^{\left\lfloor \frac{W}{w[k-1]} \right\rfloor} OPT(W - i \cdot \overbrace{w[k-1]}^{\text{denom.}}, \underbrace{k-1}_{\text{which coins can we use?}})$$

Ordering the Subproblems

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

	W = 0	1	2	…				
k = 0					0	0	0	0
1								
2								
…	1				OPT(W,k)			
	1							
	1							

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

Subproblems depended on only have smaller W and k values

only need to solve problems in the the row above and to the left
solve left to right, then top down guarentees this

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)
```

OPT[w, 0] = 1 for all w

loop over # of coins n

loop over # of cents

O(w)

$O(m W^2)$

# of coin types

All Shortest Paths

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

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

How to break this into smaller problems?

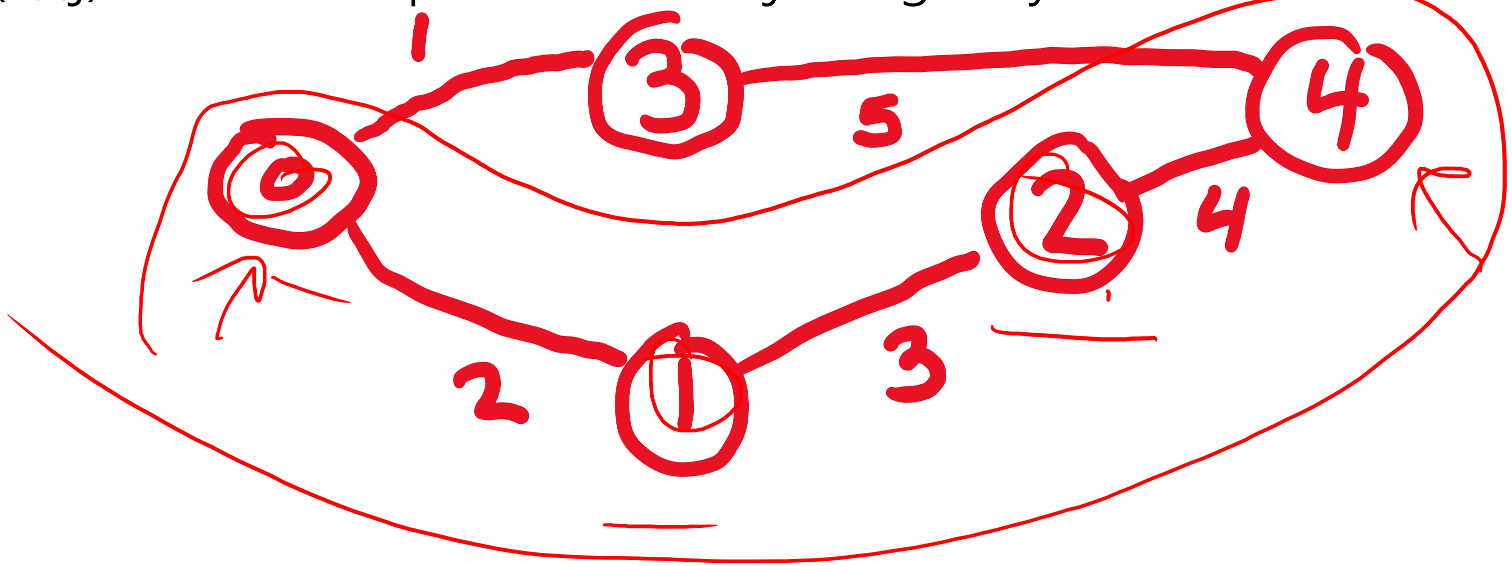
Borrow a trick from the last example: introduce a restriction:

generating artificial constrictions (VERY HELPFUL)

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

Characterizing OPT

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



K = 3

0, 1, 2

1st 3 vertices

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

using only 1st 3 vertices = 0, 1, 2 as intermediate nodes

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

Characterizing OPT

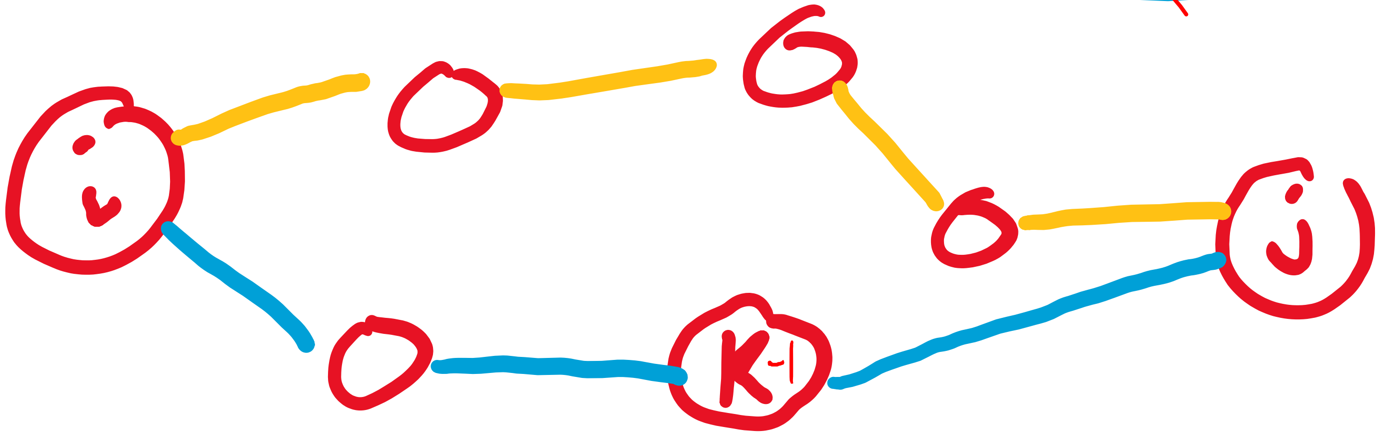
OPT(k,i,j) := shortest path from i to j using only first k vertices in between

Observation: OPT(k,i,j) either uses the k'th vertex, or it doesn't:

min path doesnt use kth vertex

use kth vertex

OPT(k,i,j) = min { OPT(k-1, i, j) , OPT(k-1, i, k) + OPT(k-1, k, j) }



Characterizing OPT

OPT(k,i,j) = min {OPT(k-1, i, j) , OPT(k-1, i, k) + OPT(k-1, k, j) }

Base cases?

The path from a vertex to itself has length 0:

OPT(k, i, i) = 0

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

OPT(0, i, j) = w_{ij} if i->j exists, otherwise ∞

Ordering the Subproblems

$$OPT(k,i,j) = \begin{cases} 0 & if\ i = j \\ w_{ij}\ if\ k = 0 & (assume\ w_{ij}\ is\ \infty\ if\ no\ edge) \\ \min\{ OPT(k-1,i,j), OPT(k-1,i,k) + OPT(k-1,k,j) & otherwise \end{cases}$$

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 , K

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

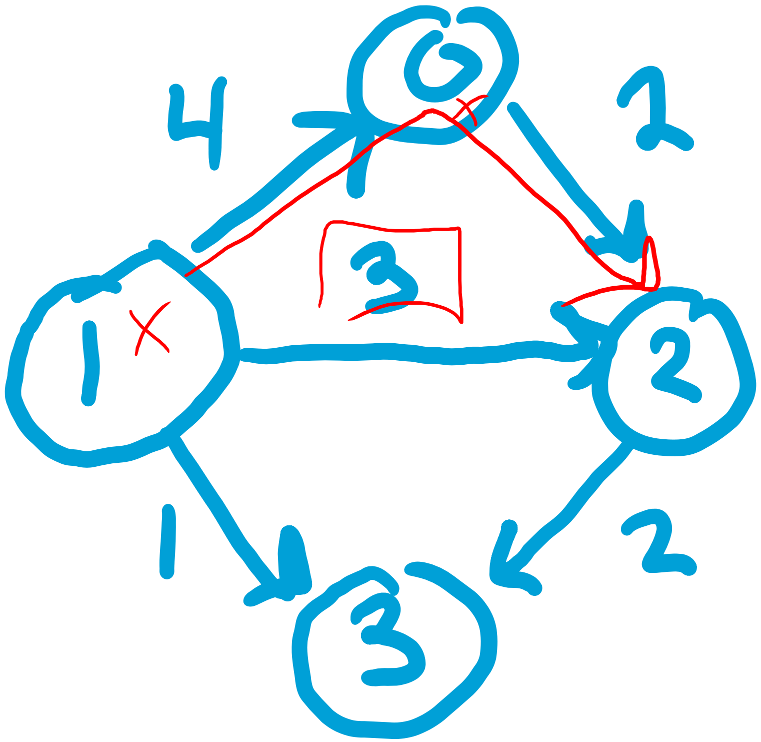
computes shortest length path between ALL vertices

```
shortestPaths(G):
    let d[][] be a |V|x|V| matrix
    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]
    return d
```

$|V|^3$

Example

step 1: Initialize main diagonal to 0's; for each entry initialize to weight of edge from i to j, or to infinity if no such edge exists.
step 2: increment k (first to 1, then 2, etc)
step 3: moving through each row i, is there an entry j such i —> k —> j is faster than i —> j (where k is the most recently available vertex; i.e. vertex labeled k-1) If so, update this entry.
step 4: repeat 2-4



Distances

k=1	j: 0	1	2	3
i: 0	0	∞	2	∞
1	4	0	3	1
2	∞	∞	0	2
3	∞	∞	∞	0

Path Reconstruction

```
shortestPaths(G):
    let d[][] be a |V|x|V| matrix
    let path[][] be a |V|x|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]
                    path[i][j] = k
    return d
```