Slides will be on the webpage in a few seconds.

up now :)

Announcements

PLEASE fill out course reviews before Sunday

I made moderate changes for this version; I'm going to make major changes before I teach this course again. Your feedback will help!

HW6 will be back soon

HW7 won't be back before the final ends, but we will release solutions by sometime Monday.

Fill out the poll everywhere for Activity Credit!
Go to pollev.com/cse417 and login with your UW identity

Longest Common Subsequence

Given two arrays A_1 and A_2 find the length of the longest subsequence that appears on both A_1 and A_2.

For example, if A_1 is a, b, c, d

And A_2 is a, c, b, a, d

The correct answer is 3 (corresponding to a, b, d or a, c, d).

Notice the subsequences are in the order of the original array.

- What’s one step?

- Or said differently, if you were going to try to write a recursive version, what would you consider checking?

r. call needs to know how many elements are left in each array

OPT[i][j] = length of the LCS of the first i elements of A_1, and first j elements of A_2

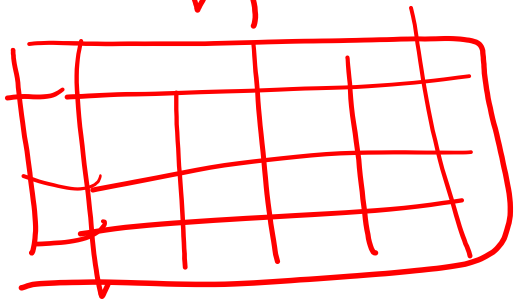
OPT(A_1.length, A_2.length).

$$
OPT(i,j) = \begin{cases} \max\{\, 1 + OPT(i-1, j-1) \text{ if } A_1[i] = A_2[j],\ OPT(i-1, j),\ OPT(i, j-1) \,\} \\ 0 \quad i = 0 \text{ or } j = 0 \end{cases}
$$

btn 1 and n

btn 1 and n

A_1, A_2 are 1-indexed array



DP Practice

The sequence C = c1, . . . , ck is a non-adjacent subsequence of A = a1, . . . , an, if C can be formed by selecting non-adjacent elements of A, (in order). The non-adjacent LCS problem is given sequences A and B, find a maximum length sequence C which is a non-adjacent subsequence of both A and B.

This problem can be solved with dynamic programming. Give a recurrence that is the basis for a dynamic programming algorithm. You should also give the appropriate base cases, and explain why your recurrence is correct.

longest common subsequence of A, B
must skip at least one element in each.

OPT(i, j) - length of longest common skipping subsequence

$$OPT(i,j) = \begin{cases} \max\{OPT(i-1,j),\ OPT(i,j-1),\ 1+OPT(i-2,j-2)\} & \text{if } i,j > 0 \\ 0 & \text{if } i=0 \text{ or } j=0 \end{cases}$$

still 1-indexing arrays in this definition
Want OPT(A.length, B.length) as our final answer.

Maximum Subarray Sum

We saw an $O(n \log n)$ divide and conquer algorithm.

Can we do better with DP?

Given: Array $A[]$

Output: i, j such that $A[i] + A[i + 1] + \cdots + A[j]$ is maximized.

Is it enough to know OPT(j)?

Remember maximum Subarray Sum?

$$INCLUDE(i) = \begin{cases} \max\{A[i], A[i] + INLCUDE(i-1)\} & \text{if } i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$OPT(i) = \begin{cases} \max\{INCLUDE(i), OPT(i-1)\} & \text{if } i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

If we include i, the subarray must be either just i or also include $i - 1$.

Overall, we might or might not include i. If we don't include i, we only have access to elements $i - 1$ and before. If we do, we want $INCLUDE(i)$ by definition.

Remember Maximum Subarray Sum?

Long subarrays only: Describe and analyze an algorithm that finds a contiguous subarray of A of length at least X (i.e. including at least X elements) that has the largest sum.

You may assume $X \leq n$.

What do I need from my recursive calls (what do they need to know?

- I need them to include element i-1 (o/w not contiguous)
- I need them to know the number of elements included (otherwise don't know if we have at least X).

→ OPT(i, k)
max sum of a contiguous subarray among 1, ..., i and containing exactly k
= max { OPT(i-1, k), Inc(i, k) }

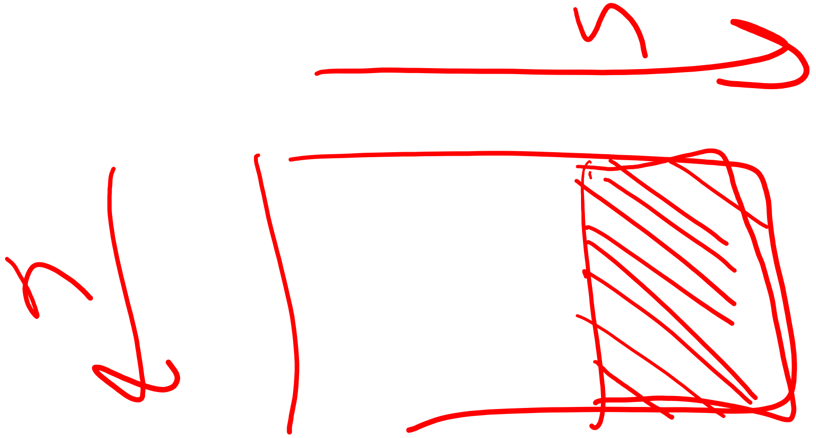
Include(i, k)
max sum of a contiguous subarray among 1, ..., i
including element i
and containing exactly k elements of the array.

$$\text{Include}(i,k) = \begin{cases} \text{Include}(i-1, k-1) + A[i] & \text{if } k \geq 2 \\ A[i] & \text{if } k = 1 \end{cases}$$

lazy version

$$\max\{OPT(n,k)\}$$

$k \geq$



Optimized version:

Key idea: once we get to at least X elements, we no longer care about the exact number of elements in the subarray. So we won't keep track of it

INCLUDE(i,k): max sum of a subarray (among elements 1,..,i) that includes at least k elements (will have at least X-k elements to the right), and includes element A[i].

$$\text{INCLUDE}(i,k) = \begin{cases} \text{INCLUDE}(i-1,k-1)+A[k] & \text{if } k>1 \\ \max\{\text{INCLUDE}(i-1,1), A[i]\} & \text{if } k=1 \end{cases}$$

What's with the "(includes at least X-k elements to the right)"?
It's a trick to make sure the definition makes sense.

In a recursive version, the parameter k is "I need at least k more elements"
In a recursive version we don't know what recursive calls were made to get to us.
In an iterative version "for us to count this, the recursive version would have already done this"
So we are saying

What's our final answer?

OPT(i,k) is max subarray sum of elements 1,...,i that includes at least k elements
(and will have at least X-k added on the right)

$$OPT(i,k) = \max\begin{cases} \text{Include}(i,k), OPT(i-1,k) & \text{if } k>0 \\ 0 & \text{if } k=0 \end{cases}$$

Our final answer is
OPT(n,X).

We don't look at OPT(n,X-1). that might only have X-1 elements!

Memoization
two $n \times X+1$ arrays

Eval order
for(k from 0 to X)
for(i from 1 to n)
eval INC then OPT

The running time is $O(nX)$. That is a little faster than $O(n^2)$ from class.

We could also change what we did in class

If we make the base case of INCLUDE(i, 0)
be the maximum subarray sum of any length ending at i.
Which is a separate O(n) calculation to store them all.

It's less intuitive (to me at least) but works.

It's essentially what a recursive version would do

Reductions

1. Figure out what you're reducing from and to

The known NP-hard problem is the source, the new problem is the target.

2. Understand both your input types (are they both graphs? Is one a graph and the other a list of variables and constraints?)

3. Understand the "certificates" of each – what are you looking for?

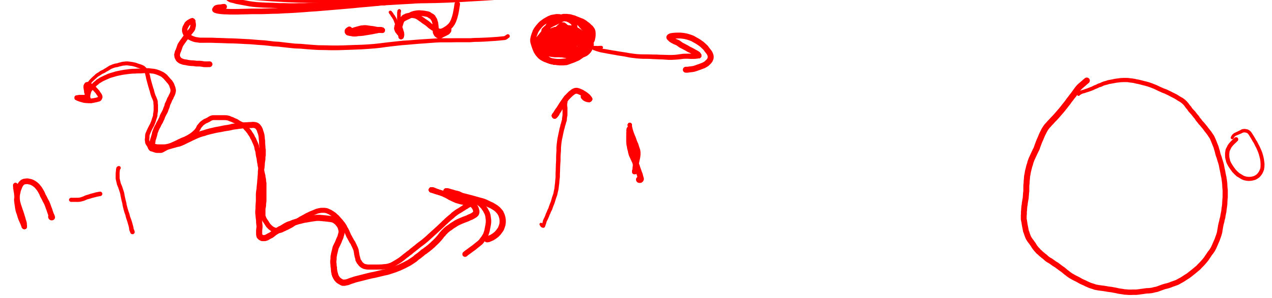
Actually designing the reduction

Your goal is to transform the certificate of the source problem into the certificate of the target problem

AND to not create any "false positive" certificates.

You are given a directed graph G = (V, E) with weights we on its edges e ∈ E. The weights can be negative or positive. The Zero-Weight-Cycle Problem is to decide if there is a simple cycle in G so that the sum of the edge weights on this cycle is exactly 0. Prove that the Zero-Weight-Cycle problem is NP-~~Complete~~ Hard. (Hint: Hamiltonian PATH) → known

Reduce from H-Path to ZWC



More full sketch

Given G, directed, unweighted graph (for Ham-Path)

Let H be a copy of G, Make every current edge of H weight 1.

Add a new vertex u.

For every vertex v of H (except u)

    add edge (v, u) of weight 1

    add edge (u, v) of weight $-n$.

b = ZeroWeightCycleSolver(H)

return b;

If G has a Ham Path then reduction says YES

We can follow Ham Path of G, follow added edge to u and back to the start of Ham Path. Total weight is

$$\underbrace{n-1}_{\text{n-1 edges to visit n vertices}} + \underbrace{1}_{\text{edge to u}} + \underbrace{(-n)}_{\text{edge back to start}} = 0.$$

If reduction says YES then Scheme found a 0 weight cycle.

There are no 0 weight edges in graph, so we need a cycle to have at least one negative edge. So we must use an edge leaving u. Because we don't repeat vertices, and only neg. edges leave u, rest of cycle must have wt. n and return to u. Only such paths visit every vertex once (otherwise not a cycle, or not enough weight). But that path is a Ham Path of G (plus the edge to u). So there is a HAM Path in G.

Thinking under pressure.

What is being CS academia/algorithm-researcher like? Do you just sit there staring at questions and reading books until you figure out an algorithm?

Chess moves are a problem that's beyond NP, but now we're able to develop AI that plays the game better than humans can. Does that mean that Non-NP = NP? What are the implications of that?

"Generalized Chess"