# Lecture 18: Dynamic Programming II

## Lecture Overview

- Thinking back over DP
- Iterating
- Edit distance
- Knapsack
- Pseudopolynomial Time

## Review:

* Steps to dynamic programming

  (a) define subproblems (and make sure they work recursively) count # subproblems

  (b) guess (part of solution) count # choices

  (c) relate subproblem solutions to current solution compute time/subproblem

  (d) recurse + memoize time = time/subproblem · # subproblems
  OR build DP table bottom-up
  (*i.e.*, iterate checking subproblems in acyclic/topological order)

  (e) solve original problem: = a subproblem
  OR by combining subproblem solutions $\implies$ extra time

* useful problems for strings/sequences $x$:

  suffixes $x[i:]$, prefixes $x[:i]$ } $\Theta(|x|)$ $\leftarrow$ cheaper $\implies$ use if possible

  substrings $x[i:j]$ } $\Theta(x^2)$

For the things we'll see today, try to look at the underlying ideas and techniques involved. The actual algorithms, while interesting in and of themselves, are not the object of this lecture. Last lecture, we discussed a number of interesting algorithms, and we touched upon the idea of using iteration (rather than recursion and memoization) for DP, and while we discussed complexity, it was generally a very simple computation.

## Edit Distance

Used for DNA comparison, diff, CVS/SVN/..., spellchecking (typos), plagiarism detection, etc.
Given two strings $x$ & $y$, what is the cheapest possible sequence of character edits (insert c, delete c, replace c $\to$ c') to transform $x$ into $y$?

- cost of edit depends only on characters $c, c'$. This may be different for each character combination, etc.
- for example in DNA, C $\to$ G common mutation $\implies$ low cost , also used in CRISPR
- cost of sequence = sum of costs of edits
- Under what circumstances (insert, delete, and replace costs) is minimum edit distance equivalent to finding longest common subsequence? Note that a subsequence is sequential but not necessarily contiguous.

Till now, the previous DP problems we've done have been on one string or object. We now have two strings to deal with, which means that we'll have to address them simultaneously.

### Subproblems for multiple strings/sequences

- combine suffix/prefix/substring subproblems
- multiply state spaces
- still polynomial for $O(1)$ strings

### Edit Distance DP

(1) subproblems: $c(i, j) =$ edit-distance$(x[i:], y[j:])$ for $0 \le i < |x|, 0 \le j < |y|$
$\implies \Theta(|x| \cdot |y|)$ subproblems

(2) guess what we need to do to turn $x$ into $y$, (3 choices):

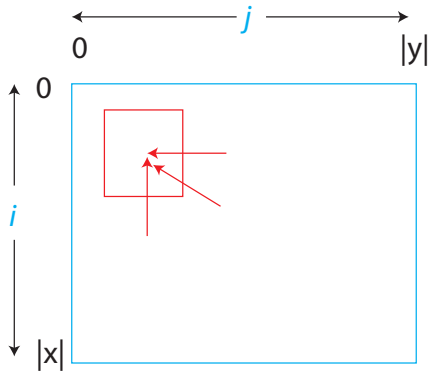
- $x[i]$ deleted
- $y[j]$ inserted
- $x[i]$ replaced by $y[j]$

(3) recurrence: $c(i, j) =$ MIN of:

- cost(delete $x[i]$) $+ c(i + 1, j)$ if $i < |x|$,
- cost(insert $y[j]$) $+ c(i, j + 1)$ if $j < |y|$,
- cost(replace $x[i] \to y[j]$) $+ c(i + 1, j + 1)$ if $i < |x| \& j < |y|$

base case: $c(|x|, |y|) = 0$
$\implies \Theta(1)$ time per subproblem

(4) topological order: DAG in 2D table (the three red arrows are the "i+1,j", "i,j+1", and "i+1,j+1" cases from the recurrence relation above; we just use the ):




- bottom-up OR right to left
- only need to keep last 2 rows/columns
  $\implies$ linear space
- total time $= \Theta(|x| \cdot |y|)$

(5) original problem: $c(0, 0)$

Now that we have a basic idea about the problem, and we have seen its recurrence relation as well as an iterative form (and how to move between the two), I highly recommend you go home and try to actually write out iterative versions of some of the DP algorithms you know. You'll find that in some cases they are more complex, but sometimes they are a lot simpler and often use a lot less space. In this case, we were always dealing with suffixes, so it sufficed to go from right to left, etc., but you might find many cases with more complicated breakdowns.

## Knapsack:

Now, we shall look at a very standard problem, but one which has a very interesting recurrence, and thus, a very interesting runtime.

We have a backpack or knapsack of size $S$ you want to pack

- item $i$ has integer size $s_i$ & real value $v_i$
  You can replace size with weight or whatever "cost" you want; we just need to have two separate cost and value sets.

- goal: choose subset of items of maximum total value subject to total size $\leq S$. Like the previous problem, we'll find that we need to keep track of multiple things. (We had the two strings before; here we have the size $S$ and the values.)

## First Attempt:

1. We start with the items in some arbitrary order and we try to read it from left to right, hopefully adding items, and creating subproblems which are suffixes of previous problems.
2. ~~subproblem = value for suffix i:~~ WRONG
3. guessing = whether to include item $i$ $\implies$ # choices = 2
4. recurrence:
   - $DP[i] = \max(DP[i+1], v_i + DP[i+1] \text{ if } ~~s_i \leq S~~?!)$
   - not enough information to know whether item $i$ fits — how much space is left? GUESS!
     We have no way of representing the fact that we used up some of the remaining size (there's nothing about S in this formula!)

## Correct:

1. subproblem = value for suffix $i$:
   given knapsack of size $X$
   $\implies$ # subproblems = $O(nS)$ !
3. recurrence:
   - $DP[i, X] = \max(DP[i+1, X], v_i + DP[i+1, X - s_i] \text{ if } s_i \leq X)$
   - $DP[n, X] = 0$
     $\implies$ time per subproblem = $O(1)$
4. topological order: for $i$ in $n, \ldots, 0$: for $X$ in $0, \ldots S$

   total time = $O(nS)$
5. original problem = $DP[0, S]$

   (& use parent pointers to recover subset)

AMAZING: effectively trying all possible subsets! ... but is this actually fast?

### Polynomial time

Polynomial time = polynomial in input size

- here $\Theta(n)$ if number $S$ fits in a word
- $O(n \lg S)$ in general if the values, etc., are huge, you might need to have $O(n + n \log S + n \log max(v_i))$ or something, but certainly not $O(nS)$
- $S$ is exponential in $\lg S$ (not polynomial)

## Pseudopolynomial Time

Notice how our previous runtime depended not only on the size of the input, but on one of the input values. Imagine a sorting algorithm where you have a loop that runs $x_i$ times (where $x_i$ is one of the values inputted)!

A number of real life problems to possess this kind of structure. For example, think about a naive algorithm for testing whether a given number is prime. How do we do this? We simply check whether $n$ can be divided by $2, 3, \ldots, n$; of course, in reality, we would only check for $2, 3, \ldots, \sqrt{n}$. Now, while this is sublinear in $n$, is is exponential in the size of $n$ – we only need $\log n$ bits to write down $n$. (Think about the fact that we can trivially write down a number, say, 1000 digits long, but it might take a very long time to check whether it is prime; adding two such numbers, however, is linear time in the size of the input.) Since we generally measure complexity with respect to the size of the input, this primality checking algorithm would be exponential; we have decided to call such things pseudopolynomial. (Note that Primes is in P, thanks to Manindra Agrawal. Neeraj Kayal. Nitin Saxena for IIT-Kanpur, who managed to show this in 2002. The speed is something like $O((\log n)^6)$; they initially started out with 12, ignoring log factors outside.)

Pseudopolynomial time = polynomial in the problem size AND the numbers (here: $S$, $s_i$'s, $v_i$'s) in input. $\Theta(nS)$ is pseudopolynomial.

An NP-complete problem with a known pseudo-polynomial time algorithm is called weakly NP-complete. An NP-complete problem is called strongly NP-complete if it is proven that it cannot be solved by a pseudo-polynomial time algorithm (unless P=NP).

Note that quasipolynomial time is a very different concept. These are much better defined, mathematically speaking, and are algorithms that fall somewhere between exponential and polynomial time. The worst case runtime of a quasipolynomial algorithm is $2^{O((\log n)^c)}$ for some positive $c$. Notice that if $c = 1$, then we get polynomial time. These generally occur when we deal with simplified versions of NP-hard problems, but there are some very strange places where such things crop up.

One such case is a Steiner tree: Given an undirected graph $G = (E, V)$ with non-negative edge weights and a set of vertices $A \in V$, find a minimum weight tree which contains all vertives in $A$, but may optionally include other vertices in $V$. That is, $A \subset V_T \subset V$. This is particularly interesting given that if $A$ is size 2, this reduces to finding the shortest path between two vertices; if, at the other extreme, $A = V$, then it reduces to finding the minimum spanning tree – both problems have polynomial time algorithms! However, solving the Steiner tree problem has proved to be very difficult, and the best we have is a quasipolynomial algorithm. (Note that the decision variant of the Steiner tree problem in graphs is NP-complete, and was among Karp's original 21 NP-complete problems – which implies that our optimization variant is also NP-hard.)