CSE 417
Dynamic Programming (pt 5)
Multiple Inputs
Reminders

> HW5 due Wednesday
Dynamic Programming Review

- Apply the steps...
  1. Describe solution in terms of solution to any sub-problems
  2. Determine all the sub-problems you’ll need to apply this recursively
  3. Solve every sub-problem (once only) in an appropriate order

- Key question:
  1. Can you solve the problem by combining solutions from sub-problems?

- Count sub-problems to determine running time
  - total is number of sub-problems times time per sub-problem

optimal substructure: (small) set of solutions, constructed from solutions to sub-problems that is guaranteed to include the optimal one
Review From Previous Lectures

> Previously:
  – Find opt substructure by considering how the opt solution could use the last input.
  – Given clever choice of sub-problems, find opt substructure by considering new options

> Tree Structure:
  – Sub-problems are left and subtrees
  – opt value = min cost of tree over choices of root:
  – Problems:
    > optimal binary search trees
    > matrix chain multiplication
    > optimal polygon triangulation (HW5)
Outline for Today

- Multiple Inputs Generally
- Longest Common Subsequence
- Edit Distance
- Pattern Matching
Multiple Inputs

> Have mainly looked at problems whose input is a list of items

> Now, we will look at problems with multiple lists of inputs...

> Can still use the same heuristic to find optimal sub-substructure: consider how the optimal solution might use the last element
Multiple Inputs

> Can still use the same heuristic to find optimal sub-substructure: consider how the optimal solution might use the last element

> Difference is that there are multiple last elements
  – the last one from each list

> To use the heuristic, consider how opt uses any of the last elements...
  – could think about just one or all of them simultaneously
  – one approach may work better than the others
Multiple Inputs: Knapsack

> We have seen a similar example already: the Knapsack problem

> Inputs are:
  - list of items, 1 .. n
  - price limit W

> Solved every sub-problem of the form 1 .. j and V with j ≤ n and V ≤ W
  - total of n(W+1) sub-problem

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>
Outline for Today

- Multiple Inputs Generally
- Longest Common Subsequence
- Edit Distance
- Pattern Matching
Longest Common Subsequence

> **Definition:** A subsequence of a list $a_1, ..., a_n$ is a list $c_1, ..., c_k$, where each $c_i$ is from the first list and they appear in the *same order*.

> Note that the indices need **not be contiguous:**
  - sub-sequences not ranges / sub-arrays

> E.g., if $A = [3, 8, -5, 0, 23, 4]$, then $B = [8, 0, 23]$ is a subsequence (not a subarray)
  $C = [3, 8, 5]$ is **not** a subsequence (no 5 in A)
  $D = [3, 4, 8]$ is **not** a subsequence (8 before 4 in A)
Longest Common Subsequence

> **Problem**: Given two lists, $a_1, ..., a_n$ and $b_1, ..., b_m$, find the longest subsequences of the two lists that are identical.
  
  - subsequence $a_{i_1}, a_{i_2}, ..., a_{i_k}$ of $a_1, a_2, ..., a_n$ and subsequence $b_{i_1}, b_{i_2}, ..., b_{i_k}$ of $b_1, b_2, ..., b_m$ with
    
    $a_{i_1} = b_{i_1}, a_{i_2} = b_{i_2}, ..., a_{i_k} = b_{i_k}$

> **Example**:

  - $A = [1, 2, 1, 5, 4, 3]$
  - $B = [2, 1, 3, 2, 1, 4]$
  - $[1, 2, 1, 4]$ is the longest common subsequence
Longest Common Subsequence

> Brute force would take $\Omega(4^{\min(n,m)})$ time
  - try all $\geq 2^{\min(n,m)}$ subsets of $a_1, ..., a_n$ with length at most $\min(n,m)$
  - try all $\geq 2^{\min(n,m)}$ subsets of $b_1, ..., b_m$ with length at most $\min(n,m)$
  - return the longest match found
Longest Common Subsequence

> Brute force would take $O(4^{\min(n,m)})$ time

> Apply dynamic programming...

> Q: How does the opt solution use the last elements ($a_n$ and $b_m$)?
  - could use just $a_n$, just $b_m$, both, or neither
Apply dynamic programming...

Q: How does the opt solution use the last elements \((a_n, b_m)\)?

- uses neither: same as opt on \(a_1, ..., a_{n-1} \) and \(b_1, ..., b_{m-1}\)
- uses only \(a_n\): same as opt on \(a_1, ..., a_n \) and \(b_1, ..., b_{m-1}\)
- uses only \(b_m\): same as opt on \(a_1, ..., a_{n-1} \) and \(b_1, ..., b_m\)
- uses both...
  > then we must have \(a_n = b_m\)
  > rest must be opt on \(a_1, ..., a_{n-1} \) and \(b_1, ..., b_{m-1}\)
  > opt value = 1 + opt value on \(a_1, ..., a_{n-1} \) and \(b_1, ..., b_{m-1}\)
    - each common subsequence on \(a_1, ..., a_{n-1} \) and \(b_1, ..., b_{m-1}\)
      becomes 1 longer by adding \(a_n\) and \(b_m\), so opt must use longest of those
Longest common subsequence

> Apply dynamic programming...

  1. Can find opt value for 1, ..., n (a) and 1, ..., m (b) using
     (i) opt value for 1, ..., n-1 and 1, ..., m
     (ii) opt value for 1, ..., n and 1, ..., m-1
     (iii) opt value for 1, ..., n-1 and 1, ..., m-1

  2. Need opt values sub-problems on 1, ..., i (a) and 1, ..., j (b) with i ≤ n and j ≤ m

> (n+1)(m+1) problem to solve
  - let i or j be zero (empty prefixes)
**Longest common subsequence**

- Apply dynamic programming...
  1. Can find opt value for 1, ..., n (a) and 1, ..., m (b) using prefixes of each.
  2. Need opt values sub-problems on 1, ..., i (a) and 1, ..., j (b) with i ≤ n and j ≤ m
  3. Solve each of these starting with i=0 or j=0
     - opt value = 0 if i = 0 or j = 0
     - opt value for 1, ..., i and 1, ..., j =
       - max( opt value for 1, ..., i-1 and 1, ..., j, opt value for 1, ..., i and 1, ..., j-1, (opt value for 1, ..., i-1 and 1, ..., j-1) + (1 if a_i = b_j else 0))

- O(1) per table entry, so O(nm) time all together
Outline for Today

- Multiple Inputs Generally
- Longest Common Subsequence
- Edit Distance
- Pattern Matching
Edit Distance

> **Problem:** Given two lists, $a_1, ..., a_n$ and $b_1, ..., b_m$, find the minimum cost way to transform $a$ into $b$ using three operations:

1. Change element $v$ to element $w$ at cost $\alpha_{v,w}$
2. Insert element $v$ at cost $\beta_v$
3. Delete element $v$ at cost $\delta_v$
Edit Distance Example

Edit distance between these two strings (DNA):

Mismatch at all the blue locations
Cost of those mismatches is $\alpha_{C,T} + \alpha_{G,T} + \alpha_{A,G} + 2\alpha_{A,C}$
Edit Distance Example

Edit distance between these two strings (DNA):

> Alternatively:
  - insert “C” at the beginning (top “−”)
  - delete “C” in the middle (across from bottom “−”)
  - cost is $\beta_C + \delta_C + \alpha_{A,C}$
Edit Distance Applications

> Computational biology ("sequence alignment")
  – measures similarity between DNA (or RNA or proteins)
  – cost of insert / delete / change based on likelihood of mutations

> Spell checkers
  – cost of insert / delete / change based on likelihood of those mistakes

> Diff tool

> Speech recognition
Edit Distance Applications

> Longest common subsequence:
  – insertion and deletion cost 1, changes costs $\infty$
  – for any common subsequence of length $k$, can first into second by:
    > deleting $n - k$ other elements from $a$
    > inserting $m - k$ other elements into $b$

> Example:
  – $A = [1, 2, 1, 5, 4, 3]$
  – $B = [2, 1, 3, 2, 1, 4]$
  – delete 5 & 3 from $A$ to get $[1, 2, 1, 4]$ (common subsequence)
  – insert 2 & 3 to this to get $B$
**Edit Distance Applications**

> Longest common subsequence:
  > – insertion and deletion cost 1, changes costs \( \infty \)
  > – for any common subsequence of length \( k \), can first into second by:
    > deleting \( n - k \) other elements from \( a \)
    > inserting \( m - k \) other elements into \( b \)
    > total cost is \( n + m - 2k \)
  > – since \( n + m \) is constant, minimizing \( n + m - 2k \) is maximizing over \( k \)

> Edit distance generalizes longest common subsequence:
  > – another example of robustness to problem changes
  > – also suggests previous solution will work here too...
**Edit Distance**

- Apply dynamic programming...

**Q**: How does the opt solution *match* the last elements \((a_n \text{ and } b_m)\)?

- if \(a_n = b_m\): \(\text{opt value} = \text{opt on } a_1, ..., a_{n-1} \text{ and } b_1, ..., b_{m-1}\)
- if change: \(\text{opt value} = \alpha_v,w + \text{opt on } a_1, ..., a_{n-1} \text{ and } b_1, ..., b_{m-1}\)
- if insert \(b_m\): \(\text{opt value} = \beta_v + \text{opt on } a_1, ..., a_n \text{ and } b_1, ..., b_{m-1}\)
- if delete \(a_n\): \(\text{opt value} = \delta_v + \text{opt on } a_1, ..., a_{n-1} \text{ and } b_1, ..., b_m\)
Apply dynamic programming...

1. Can find opt value for 1, ..., n (a) and 1, ..., m (b) using prefixes of each.
2. Need opt values sub-problems on 1, ..., i (a) and 1, ..., j (b) with i ≤ n and j ≤ m
3. Solve each of these starting with i=0 or j=0
   > if i = 0, then opt value = \( \beta_{b1} + ... + \beta_{bj} \)
   > if j = 0, then opt value = \( \delta_{a1} + ... + \delta_{ai} \)
   > opt value for 1, ..., i and 1, ..., j =
     \[
     \max( \alpha_{v,w} + \text{opt value for } 1, ..., i-1 \text{ and } 1, ..., j-1, \\
     \beta_v + \text{opt value for } 1, ..., i \text{ and } 1, ..., j-1, \\
     \delta_v + \text{opt value for } 1, ..., i-1 \text{ and } 1, ..., j),
     \]
     where \( v = a_i \) and \( w = b_j \)

set \( \alpha_{v,w} = 0 \) when \( v = w \)
Edit Distance

> Running time is $O(nm)$ as before

> Very easy to implement
  – about 10 lines of code (see the textbook)

> Easily implemented in Excel
  – filling in a 2D table
  – each value is a minimum of 4 others
In computation biology, n and m could be very large...
  – with $n = m = 100k$, $nm = 10b$
  – running time is fine since modern machines perform billions of ops per sec
  – memory use of 10GB (assuming 1B per entry) is (just) possible

With $n = m = 1,000,000$ though:
  – running time is okay: 1000B operations in minutes
  – memory use of 1TB is not reasonable
    > could use disk space, but time would increase by factor of ~1k

More on this next time...
Outline for Today

- Multiple Inputs Generally
- Longest Common Subsequence
- Edit Distance
- Pattern Matching
Problem: Given a content string $a_1, \ldots, a_n$ and a pattern $p_1, \ldots, p_m$, find the longest substring of the content that matches the pattern according to the following rules:

- ‘?’ in the pattern matches any single character of content
- ‘*’ in the pattern matches any substring (including an empty one)
- any other letter in the pattern matches only the same letter of content
Pattern Matching Examples

> Content "abcba"
    Pattern "a*b"
    – longest match is prefix “abcb”

> Content "abcba"
    Pattern "b?b"
    – longest match is “bcb”

> Content "abcba"
    Pattern "b??a"
    – longest match is suffix “bcba”
Pattern Matching Applications

> Common feature of editors and IDEs

> Many also support regular expression matching
  – RE matching is part of most standard libraries
  – more on that later...
Pattern Matching

> Apply dynamic programming...

> Like max sub-array sum, it will be helpful to change the problem: find the longest match ending at $a_n$
  > apply DP to the original problem and you will find you need to solve these
  > but these are also sufficient to solve the whole problem
    > every match ends somewhere
    > longest over the longest ending at $a_1, \ldots, a_n$ is the longest overall
Pattern Matching

> Apply dynamic programming...
  – can consider how either $a_n$ or $p_m$ (or both) is used by the longest match
  – turns out to be easiest to think about how $p_m$ is used
    > in practice, just try all and see what works

> Q: How does the longest match use $p_m$?
> Depends on what $p_m$ is
  – $p_m$ is a letter
  – $p_m$ is a ‘?’
  – $p_m$ is a ‘*’
Pattern Matching

> Q: How does the longest match use $p_m$? Depends on what $p_m$ is...
  > $p_m$ is a ‘?’
  > if ‘?’ matches $a_m$, then $p_1, ..., p_{m-1}$ matches $a_1, ..., a_{n-1}$
  > longest match is the longest match of 1 .. n-1 with 1 .. m-1
Pattern Matching

> Q: How does the longest match use $p_m$? Depends on what $p_m$ is...

- $p_m$ is a '?'
  > if '?' matches $a_m$, then $p_1, \ldots, p_{m-1}$ matches $a_1, \ldots, a_{n-1}$
  > longest match is the longest match of 1 ..  $n-1$ with 1 ..  $m-1$

- $p_m$ is a '*'
  > if '*' matches $a_m$, then $a_1, \ldots, a_{n-1}$ either matches $p_1, \ldots, p_{m-1}$ or $p_1, \ldots, p_{m}$
Pattern Matching

> Q: How does the longest match use \( p_m \)? Depends on what \( p_m \) is...

- \( p_m \) is a '?'
  - if '?' matches \( a_m \), then \( p_1, ..., p_{m-1} \) matches \( a_1, ..., a_{n-1} \)
  - longest match is the longest match of 1 .. \( n-1 \) with 1 .. \( m-1 \)

- \( p_m \) is a '*'
  - if '*' matches \( a_m \), then \( a_1, ..., a_{n-1} \) either matches \( p_1, ..., p_{m-1} \) or \( p_1, ..., p_m \')

```
abcba

|a*|
```
Pattern Matching

> Q: How does the longest match use $p_m$? Depends on what $p_m$ is...
  > $p_m$ is a '?'
  >   if '?' matches $a_m$, then $p_1, ..., p_{m-1}$ matches $a_1, ..., a_{n-1}$
  >   longest match is the longest match of 1 .. n-1 with 1 .. m-1
  > $p_m$ is a '*'
  >   if '*' matches $a_m$, then $a_1, ..., a_{n-1}$ either matches $p_1, ..., p_{m-1}$ or $p_1, ..., p_m$
  >   longest match is the longer of match of 1 .. n-1 with 1 .. m-1 and 1 .. n-1 with 1 .. m

actually, this has a problem...
it does not allow the '*' to match nothing
Pattern Matching

Q: How does the longest match use $p_m$? Depends on what $p_m$ is...

- $p_m$ is a '?'
  - if '?' matches $a_m$, then $p_1, \ldots, p_{m-1}$ matches $a_1, \ldots, a_{n-1}$
  - longest match is the longest match of $1 .. n-1$ with $1 .. m-1$

- $p_m$ is a '*'
  - then either $a_1, \ldots, a_{n-1}$ matches $p_1, \ldots, p_m$ or $a_1, \ldots, a_n$ matches $p_1, \ldots, p_{m-1}$
  - longest match is the longer of match of $1 .. n-1$ with $1 .. m$ and $1 .. n$ with $1 .. m-1$

either match $a_n$ with same pattern
or '*' matches nothing
(can still match multiple characters)
How does the longest match use $p_m$? Depends on what $p_m$ is...

- $p_m$ is a ‘?’
  > if ‘?’ matches $a_m$, then $p_1, ..., p_{m-1}$ matches $a_1, ..., a_{n-1}$
  > longest match is the longest match of 1 .. n-1 with 1 .. m-1

- $p_m$ is a ‘*’
  > then either $a_1, ..., a_{n-1}$ matches $p_1, ..., p_m$ or $a_1, ..., a_n$ matches $p_1, ..., p_{m-1}$
  > longest match is the longer of match of 1 .. n-1 with 1 .. m and 1 .. n with 1 .. m-1

- $p_m$ is a letter
  > if $p_m$ matches $a_m$, then
    longest match is the longest match of 1 .. n-1 with 1 .. m-1
  > if $p_m$ does not match $a_m$, then there is no match
Apply dynamic programming...

1. Can find longest match for 1 .. n (a) and 1 .. m (p) using prefixes of each
2. Need longest match on 1, ..., i (a) and 1, ..., j (p) with i ≤ n and j ≤ m
3. Solve each of these starting with i=0
   > longest match starts at i+1 if j=0
     - that indicates the range i+1 .. i, which is empty
   > longest match starts at infinity if i=0 (and j > 0)
     - that indicates no range
   > longest match for 1 .. i and 1 .. j (i > 0 and j > 0) starts at
     min of four cases on previous slide
     - (if/then’s are better written as code... still very short)
     - chose infinity for no range so min will never choose it if a match exists
Pattern Matching

> Apply dynamic programming...
> 1. Can find longest match for 1 .. n (a) and 1 .. m (p) using prefixes of each
> 2. Need longest match on 1, ..., i (a) and 1, ..., j (p) with i ≤ n and j ≤ m
> 3. Solve each of these starting with i=0

> (n+1)(m+1) entries in table, and O(1) time per entry,
so total running time is O(nm)
  – in practice, n >> m (say, m ≤ 100), so this is O(n)

> Only needs O(m) memory
  – only need column for i-1 to compute i, so just keep prev column
  – this is why we started with i=0 rather than j=0
Regular expressions greatly generalize these simple patterns

However, the matching algorithm is largely unchanged
- prefixes of the pattern are replaced with states of the NFSM
- for our simple patterns, this produces the same result because states of the equivalent NFSM are in 1-to-1 correspondence with prefixes
- for more general patterns, that is longer the case, so it becomes necessary to determine the NFSM states