**Dynamic Programming**

Some of the slides are courtesy of Charles Leiserson with small changes by Carola Wenk

---

**Example: Floyd Warshall Algorithm:**

**Computing all pairs shortest paths**

- Given $G(V, E)$, with weight $w(v_i, v_j)$ given on each of its edges (positive or negative), the output is a matrix $D[1..n, 1..n]$ such that (for every $i, j$)
  
  $D[i, j]$ is the length of the shortest path from $v_i$ to $v_j$

- How to find the shortest paths (and not only their costs) will be discussed in the homeworks.
  (analogous to Dijkstra)

- Assume no negative cycles exist in $G(V, E)$.

- In the homework: Finding such cycles.

---

**Assume** $V = \{v_1, v_2, \ldots, v_n\}$

**Def** $P_{k}(i, j)$ is the shortest path $v_i$ to $v_j$ avoiding any vertex from $\{v_{k+1}, \ldots, v_n\}$ as intermediate vertex.

**Example:** $P_{1}(i, j)$ could not go through any vertex of $V$.

**Def** $D_{k}[i, j]$ is its length of $P_{k}(i, j)$

So if the edge $(v_i, v_j)$ is in $G$ then

$P_{k}(i, j) = \{(v_i, v_j)\}$

$D_{k}[i, j] = w(v_i, v_j)$

If the edge $(v_i, v_j)$ is not in $E$, then $D_{k}(i, j) = +\infty$ (since any path connecting them must use a vertex from $V = \{v_1, v_n\}$
Def $P_{k}(i,j)$ is the shortest path from $v_{i}$ to $v_{j}$ avoiding any vertex from $\{v_{k+1} \ldots v_{n}\}$ as an intermediate vertex. (the sets $\{v_{k+1} \ldots v_{n}\}$ is forbidden)

Def $D_{k}[i,j]$ is its length of $P_{k}(i,j)$

Assume $D_{k-1}[i,j]$ has been computed ($1 < i, j < n$).

We now want to compute the matrix $D_{k}[i,j]$.

Now we could (but don’t have to) go through $v_{k}$ along the shortest path $v_{i} \rightarrow v_{j}$.

Two option:
1. Going through $v_{k}$ is longer, and we better stick to $P_{k-1}(i,j)$.
2. Use $P_{k-1}(i,k)$, the shortest path $v_{i} \rightarrow v_{k}$ to reach $v_{k}$, and continue $P_{k-1}(k,j)$ along to $v_{j}$.

Conclusion: $D_{k}[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$

**Floyd Warshall-Pairs Shortest Paths**

Computing $D_{k}[i,j]$ for every $i,j,k$.

Algorithm AllPair($G$) for all vertex pairs $(i,j)$

Use $n$ tables $D_{0}, D_{1}, \ldots, D_{n}$. Each is an $n \times n$ table:

if $i = j$ then $D_{0}[i,j] \leftarrow 0$
else if $(v_{i}, v_{j})$ is an edge in $G$

$D_{0}[i,j] \leftarrow w(v_{i}, v_{j})$
else

$D_{0}[i,j] \leftarrow \infty$

for $k \leftarrow 1$ to $n$ do

for $i \leftarrow 1$ to $n$ do

for $j \leftarrow 1$ to $n$ do

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

return $D_{n}$

**Floyd’s algorithm: example**
Floyd Warshall-Pairs Shortest Paths
Computing $D_{i,j}$ for every $i,j,k$.

Algorithm AllPair($G$) for all vertex pairs $(i,j)$
Use $n$ tables $D_0, D_1, \ldots, D_n$. Each is an $n \times n$
if $i = j$ then $D_0[i,i] \leftarrow 0$
else if $(v_i,v_j)$ is an edge in $G$
$D_0[i,j] \leftarrow w(v_i,v_j)$
else
$D_0[i,j] \leftarrow +\infty$
for $k$ from 1 to $n$
do
for $i$ from 1 to $n$
do
for $j$ from 1 to $n$
do
$D_k[i,j] = \min \{ D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j] \}$
return $D_n$

Running time $O(n^3)$
Space ???

Dynamic Programming:
Example 2: Longest Common Subsequence

We look at sequences of characters (strings)
e.g. $x = \text{"ABCA"}$

Def: A subsequence of $x$ is a sequence obtained from $x$ by possibly deleting some of its characters (but without changing their order)
Examples: “ABC”, “ACA”, “AA”, “BCA"

Def: A prefix of $x$, denoted $x[1..m]$, is the sequence of the first $m$
characters of $x$
Examples: $x[1..4] = \text{"ABCA"}$ $x[1..3] = \text{"ABC"}$ $x[1..2] = \text{"AB"}$ $x[1..1] = \text{"A"}$ $x[1..0] = \text{"\"}"

Example 1: Longest Common Subsequence (LCS)
- Given two sequences $x[1..m]$ and $y[1..n]$, find a longest
  subsequence common to them both.

  “a” not “the”

$x$: A B C B D A B
$y$: B D C A B A

BCBA = LCS($x$, $y$)

Different phrasing: Find a set of a maximum number of segments, such that
- Each segment connects a character of $x$ to an identical character of $y$.
- Each character is used at most once.
- Segments do not intersect.
Brute-force LCS algorithm
Checking every subsequence of \( x \) whether it is also a subsequence of \( y \).

Analysis
• Checking = \( \Theta(m+n) \) time per subsequence.
• \( 2^m \) subsequences of \( x \)

Worst-case running time = \( \Theta((m+n)2^n) \)
= exponential time.

Towards a better algorithm

Simplification:
1. Look at the length of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence \( s \) by \( |s| \).

Strategy: Consider prefixes of \( x \) and \( y \).
• Define \( c[i, j] = |\text{LCS}(x[1..i], y[1..j])| \).
• Then, \( c[m, n] = |\text{LCS}(x, y)| \).

Recursive formulation

Theorem.
\[
c[i, j] = \begin{cases} 
  c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\
  \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.}
\end{cases}
\]

Proof: It is impossible that \( x[i] \) is matched to an element in \( y[1..j-1] \) and in addition \( y[j] \) is matched to an element in \( x[1..i-1] \).
Recursive formulation-cont

Case (I): \( x[i] = y[j] \).  Claim: \( c[i, j] = c[i-1, j-1] + 1 \).

Proof:

We claim that there is a max matching that matches \( x[i] \) to \( y[j] \).

Indeed, if \( x[i] \) is matched to \( y[k] \) (for \( k < j \)) then \( y[j] \) is unmatched (otherwise we have two crossing segments). Hence we can obtain another matching of the same cardinality by match \( x[i] \) to \( y[j] \).

This implies that we can match \( x[1..i] \) to \( y[1..j] \), and add the match \( (x[i], y[j]) \). So \( c[i, j] = c[i-1, j-1] + 1 \).

---

Recursive formulation-cont

Case (II): \( x[i] \neq y[j] \). Claim: \( c[i, j] = \max\{c[i-1, j], c[i, j-1]\} \).

Recall - in LCS(\( x[1..i], y[1..j] \)) it cannot be that both \( x[i] \) and \( y[j] \) are both matched.

If \( x[i] \) is unmatched then \( LCS(x[1..i], y[1..j]) = LCS(x[1..i-1], y[1..j]) \)
If \( y[j] \) is unmatched then \( LCS(x[1..i], y[1..j]) = LCS(x[1..i], y[1..j-1]) \)

So \( c[i, j] = \max\{c[i-1, j], c[i, j-1]\} \).

---

Dynamic-programming hallmark #1

**Optimal substructure**

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If \( z = LCS(x, y) \), then any prefix of \( z \) is an LCS of a prefix of \( x \) and a prefix of \( y \).
**Recursive algorithm for LCS**

LCS($x, y, i, j$)

- if ($i = 0$ or $j = 0$) return 0
- if $x[i] = y[j]$ then return $\text{LCS}(x, y, i-1, j-1) + 1$
- else return $\max \{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \}$

To call the function LCS($x, y, m, n$)

**Worst-case**: $x[i] \neq y[j]$, for all $i,j$ in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

---

**Recursion tree**

$m = 3, n = 4$:

```
  m+n
  /   \
3---1
  |   |
 2---2
```

Height = $m + n$ ⇒ work potentially $2^{m+n}$ exponential. but we’re solving subproblems already solved!

---

**Dynamic-programming hallmark #2**

*Overlapping subproblems*

A recursive solution contains a "small" number of distinct subproblems repeated many times.

---

The number of distinct LCS subproblems for two strings of lengths $m$ and $n$ is only $mn$. 
Memoization algorithm

**Memoization:** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS($x, y$)

for $i=1$ to $m$
for $j=1$ to $n$
if ($x[i] = y[j]$)
   then $c[i,j] ← c[i-1,j-1] + 1$
else $c[i,j] ← \max\{ c[i-1,j], c[i,j-1] \}$

Time = $\Theta(mn) = \text{constant work per table entry}$.
Space = $\Theta(mn)$.

---

LCS: Dynamic-programming algorithm

LCS($X,Y$)="BCBA"

Y=A B C B D A B
X=B D C A B A

Reconstruction $z=LCS(x,y)$

**IDEA:** Compute the table bottom-up. Fill $z$ backward.

Observation: $c[i,j]=c[i-1,j]$ and $c[i,j]=c[i,j-1]$

**Proof Sketch:** We use a longer prefix, so there are more chars to be matched.

LCS Reconstruction:

Set $i=m, j=n, k=c[i,j]$
While(k>0)  
if ($c[i,j]=c[i-1,j]$ and $c[i,j]=c[i,j-1]$)  
   $z[k] = x[i]$;  
   $i--; j--; k--$  
else if ($c[i,j]=c[i-1,j]$) or ($c[i,j]=c[i,j-1]$)  
   if ($c[i,j]=c[i,j-1]$)  
      $j--$;  
   else $i--;$  
}
Reconstructing $z = \text{LCS}(X,Y)$

Another idea – While filling $c[i,j]$, add arrows to each cell $c[i,j]$ specifying which neighboring cell $c[i,j]$ it got its value.

- $c[i,j]$ flag = “” if $c[i,j] = c[i-1,j-1] + 1$
- $c[i,j]$ flag = “↑” if $c[i,j] = c[i-1,j]$
- $c[i,j]$ flag = “←” if $c[i,j] = c[i,j-1]$

Example 3: Edit distance

Given strings $x,y$, the edit distance $ed(x,y)$ between $x$ and $y$ is defined as the minimum number of operations that we need to perform on $x$, in order to obtain $y$.


Examples:

- $ed("aaba", "aaba") = 0$
- $ed("aaa", "aaba") = 1$
- $ed("aaba", "abaa") = 1$
- $ed("baaa", "") = 4$
- $ed("baaa", "aabb") = 2$

Example 3’: “Priced” Edit distance $ed(x,y)$

Assume also given

- $\text{InsCost}$ - the cost of a single insertion into $x$.
- $\text{DelCost}$ - the cost of a single deletion from $x$, and
- $\text{RepCost}$ - the cost of replacing one character of $x$ by a different character.

Definition: Given strings $x,y$, the edit distance $ed(x,y)$ between $x$ and $y$ is the cheapest sequence of operations, starting on $x$ and ending at $y$.

Problem: Compute $ed(x,y)$, and compute the sequence of operations.
**Thm:**

Let $c[i, j] = ed(x[1..i], y[1..j])$

Assume $c[i-1, j-1], c[i-1, j], c[i, j]$ are already computed.

If $x[i] = y[j]$ then $c[i, j] = c[i-1, j-1]$  
Else if $x[i] != y[j]$

$c[i, j] = \min$

$c[i-1, j-1] + \text{RepCost}$, //convert $x[1..i-1] \Rightarrow y[1..j-1]$, and replace $y[j]$ by $x[i]$

$c[i-1, j] + \text{DelCost}$, //delete $x[i]$ and convert $x[1..i-1] \Rightarrow y[1..j]$

$c[i, j-1] + \text{InsCost}$ //convert $x[1..i] \Rightarrow y[1..j-1]$, and insert $y[j]$

\}

---

**Algorithm**

**Memoization:** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```
ed(x, y)
for i=0 to m  c[i, 0] = i * DelCost
for j=0 to n  c[0, j] = j * InsCost
for i=1 to m
  for j=1 to n
    if (x[i] = y[j])
      then $c[i, j] \leftarrow c[i-1, j-1]$
    else $c[i, j] \leftarrow \min$
      $c[i-1, j] + \text{DelCost}$,
      $c[i, j-1] + \text{RepCost}$,
      $c[i-1, j-1] + \text{InsCost}$
```

Time = $O(mn)$ = constant work per table entry. Space = $O(mn)$. 