Design and Analysis of Algorithms

Dynamic Programming

Reference:
CLRS Chapter 15

Topics:

- **Elements of DP Algorithms**
- **Longest Common Subsequence**

Elements of DP Algorithms

- **Optimal substructure(principle of optimality)**
  - **An optimal solution to the problem contains within its optimal solutions to subproblems.**
- **Optimal substructure varies across problem domains in two ways:**
  - **How many subproblems are used in an optimal solution to the original problem**
  - **How many choices we have in determining which subproblem(s) to use in an optimal solution.**

Elements of DP Algorithms

- In assembly-line scheduling
  - We had $\Theta(n)$ subproblems overall and only two choices to examine for each, yielding a $\Theta(n)$ running time.

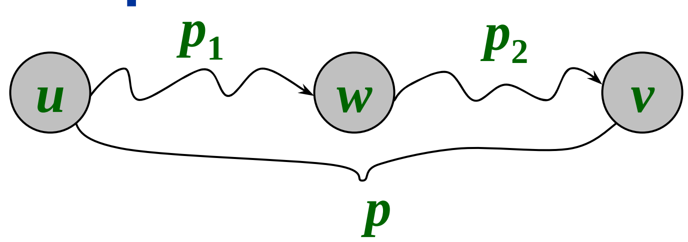
$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j\text{-}1] + a_{1,j},\ f_2[j\text{-}1] + t_{2,j\text{-}1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

  - $f^* = \min(f_1[n] + x_1,\ f_2[n] + x_2)$
- For matrix-chain multiplication
  - There were $\Theta(n^2)$ subproblems overall, and in each we had at most $n - 1$ choices, giving an $O(n^3)$ running time.

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{ m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

Optimal substructure

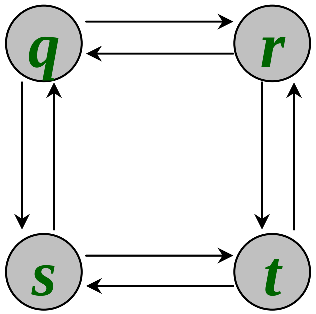
- **Optimal substructure does not apply to all optimization problems.**
- **Example. Given a directed graph $G = (V, E)$, and a pair of vertices u and v:**
  - **Shortest path find a path $u \sim> v$ with the fewest edges.**
  - **Longest path find a path $u \sim> v$ with the most edges.**
- **Shortest path has optimal substructure.**



Proof. If there's a shorter path from u to w, call it p'_1, then p'_1p_2 contains a shorter path than p_1p_2 from u to v, which contradicts the assumption.

Optimal substructure

- Longest path does not have optimal substructure.



- Longest path from q to t: $q \to r \to t$.
- Longest path from r to t: $r \to q \to s \to t$, which is not contained in the longest path from q to t.
- Difference between shortest and longest path:
  - Shortest path has independent subproblem (solution to one problem does not depend on the other).
  - If $(p_1 = uw)(p_2 = wv)$ is a shortest path, then p_1 and p_2 cannot share any vertex other than w.

Recursive Matrix Chain

- **To illustrate the overlapping-subproblem property, consider the CMM problem recursive algorithm.**

RECURSIVE MATRIX-CHAIN MULTIPLICATION

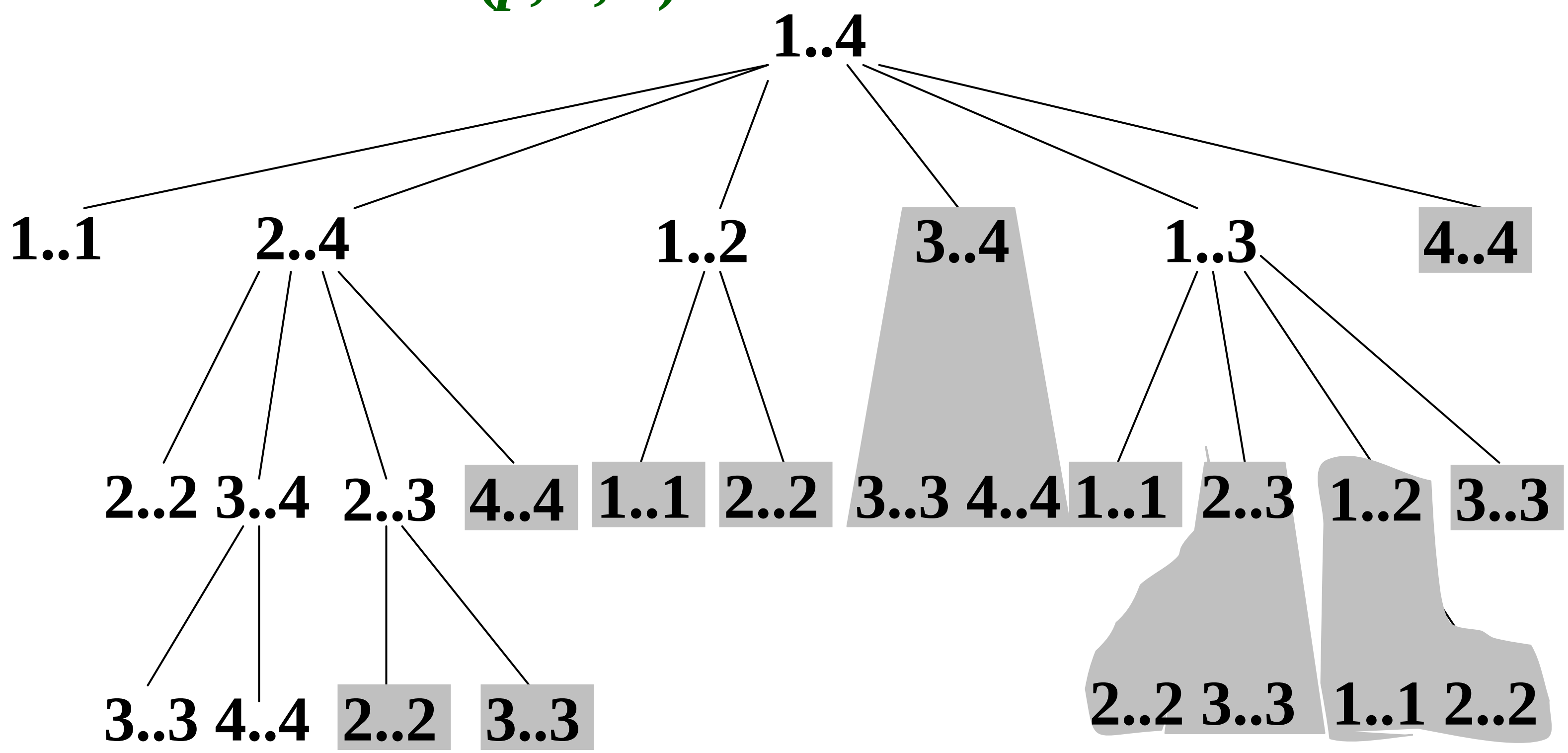
```
RECURSIVE-MATRIX-CHAIN(p,i,j)
if i = j
  then return 0
m[i,i] ← ∞
for k ← i to j-1
     do q ← RECURSIVE-MATRIX-CHAIN(p,i,k)
            + RECURSIVE-MATRIX-CHAIN(p,k+1,j)
            + p_{i-1} p_k p_j
        if q < m[i,j]
           then m[i,j] ← q
return m[i,j]
```

$$T(n) = \sum_{1 \le k < n} (T(k) + T(n-k) + 1) + 1 = \Omega(2^n)$$

see p_{346}

Elements of DP Algorithms

- **Overlapping subproblems**
  - **The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN(p, 1, 4).**



Techniques for DP Algorithms

- **Reconstructing an optimal solution**
  - **We often store which choice we made in each subproblem in a table.**
    - **In assembly-line scheduling, we store in $l_i[j]$ the station preceding $S_{i,j}$. Reconstructing the predecessor stations takes only O(1) time per station, even without the $l_i[j]$.**
    - **For matrix-chain multiplication, we maintain the $s[i, j]$ table, after filling in the table $m[i, j]$ which contains optimal subproblems costs. The table $s[i, j]$ saves us a significant amount of work when reconstructing …. Why ?**

Top Down Recursive VS Bottom Up DP

- Bottom-up dynamic programming algorithm
  - More efficient
    - » regular pattern of table access can be exploited to reduce time or space
    - » take advantage of the overlapping-subproblems property.
- Top-down recursive algorithm
    - » repeatedly resolve each subproblem each time it appears in the recursion tree.
    - » recursion overhead
    - » Only work when the total number of subproblems in the recursion is small.

Memoization

- **A variation of dynamic programming**
  - **offers the efficiency of the usual dynamic programming approach while maintaining a top-down strategy.**
- **Ideas:**
  - **to memoization the natural, but inefficient , recursive algorithm.**
  - **A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem.**
  - **Initially contain a special value to indicate that the entry has yet to be filled in. when the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and then stored in the table.**

Memoized ver. of Recursive-Matrix-Chain

MEMOIZED MATRIX-CHAIN

```
MEMOIZED-MATRIX-CHAIN(p)
  n ← length[p]-1
  for i ← 1 to n
      do for j ← i to n
              do m[i,j] ← ∞
  return LOOKUP-CHAIN(p,1,n)
```

LOOKUP-CHAIN

```
LOOKUP-CHAIN(p,i,j)
  if m[i,j] < ∞
    then return m[i,j]
  if i=j
    then m[i,j] ← 0
    else for k ← i to j-1
                do q ← LOOKUP-CHAIN(p,i,k)
                        +LOOKUP-CHAIN(p,k+1,j)+ p_{i-1} p_k p_j
                    if q < m[i,j]
                      then m[i,j] ← q
  return m[i,j]
```

Top Down Memoi. VS Bottom Up DP

- Bottom-up dynamic programming
  - all subproblems must be solved
  - regular pattern of table access can be exploited to reduce time or space
- Top-down + memoization
  - solve only subproblems that are definitely required
  - recursion overhead
- Both methods solve the matrix-chain multiplication problem in $O(n^3)$ and take advantage of the overlapping-subproblems property.

Longest Common Subsequence

- In biological applications, we often want to compare the DNA of two (or more) different organisms.
- A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine, and thymine (A, G, C, T).
- Comparison of two DNA strings

  S_1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

  S_2=GTCGTTCGGAATGCCGTTGCTCTGTAAA

  S_3=GTCGTCGGAAGCCGCCGAA

Longest Common Subsequence

- **Similarity can be defined in different ways:**
  - **Two DNA strands are similar if one is a substring of the other.**
  - **Two strands are similar if the number of changes needed to turn one into the other is small.**
  - **There is a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively. The longer the strand S_3 we can find, the more similar S_1 and S_2 are. (we focus on this)**

Dynamic programming for LCS

- **Longest common subsequence(LCS)**
  - **Given two sequences $x[1..m]$ and $y[1..n]$, find a longest subsequence common to then both.**

"a" not "the"

x: A B C B D A B

y: B D C A B A

BCBA
= LCS(x, y)

functional notation, but not a function

Brute-force LCS algorithm

- Check every subsequence of $x[1..m]$ to see if it is also a subsequence of $y[1..n]$.

- Analysis
  - Checking = $O(n)$ time per subsequence.
  - 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
  - Worst-case running time = $O(n2^m)$
    = 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 $c[i, j] = |LCS(x[1..i]), y[1..j])|$.**
  - **Then, $c[m, n] = |LCS(x, y)|$.**

Optimal substructure

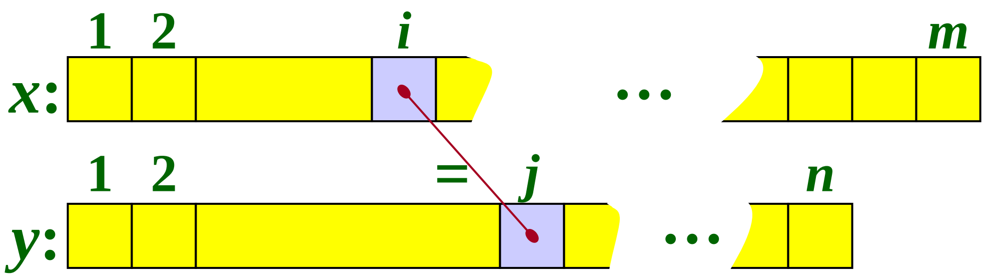
- Notation. $X_i = \langle x_1, \ldots, x_i \rangle$.
- Theorem. Let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$.
  - If $x_m = y_n$
    - » then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
  - If $x_m \neq y_n$
    - » then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y
  - If $x_m \neq y_n$
    - » then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}
- Proof.

Recursive formulation

- **Theorem.**

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

Proof. Case $x[i] = y[j]$:



Let $z[1..k] = \text{LCS}(x[1..i]), y[1..j])$, where $c[i, j] = k$. Then $z[k] = x[i])$, or else z could be extended. Thus, $z[1..k\text{-}1]$ = is CS of $x[1..i\text{-}1]$ and $y[1..j\text{-}1]$.

Proof(continued)

- **Claim:** $z[1..k\text{-}1] = \mathrm{LCS}(x[1..i\text{-}1]), y[1..j\text{-}1])$.

  Suppose w is a longer CS of $x[1..i\text{-}1])$ and $y[1..j\text{-}1]$, that is, $|w| > k-1$. Then, cut and paste: $w||z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1..i])$ and $y[1..j]$ with $|w||z[k]| > k$. Contradiction, proving the claim.

Thus, $c[i-1, j-1] = k\text{-}1$, which implies that $c[i, j] = c[i-1, j-1] + 1$.

Other cases are similar. □

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

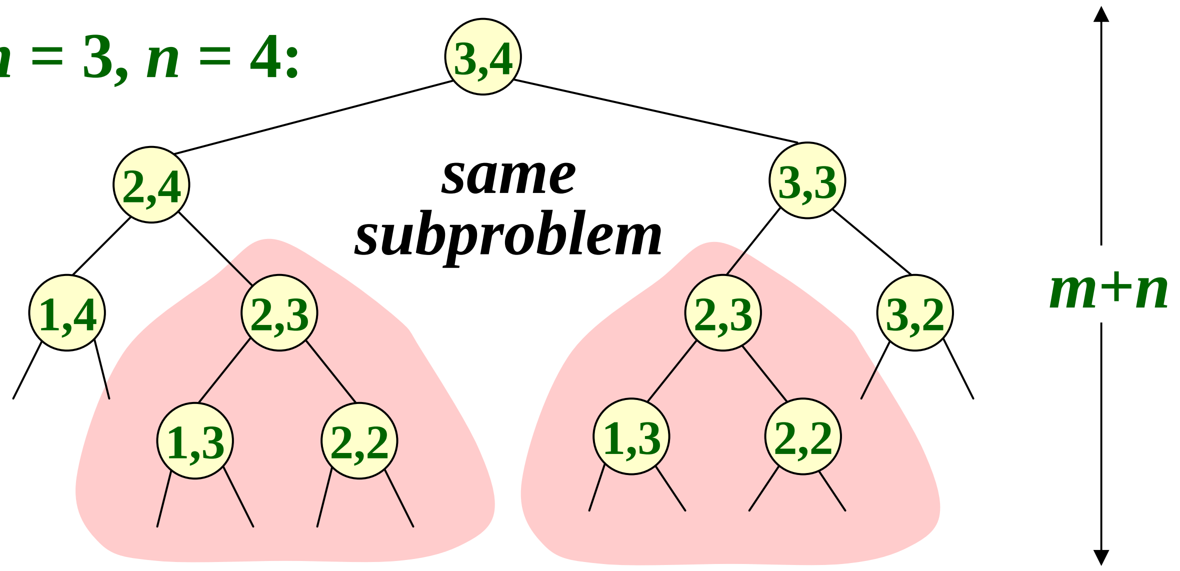
RECURSIVE for LCS

```
LCS(x,y,i,j)
  if x[i] = y[j]
     then c[i,j] ← LCS(x,y,i-1,j-1)+1
     else c[i,j] ← max{LCS(x,y,i-1,j),
                       LCS(x,y,i,j-1)}
```

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

Recursive tree

m = 3, *n* = 4:



- Lots of repeated subproblems.
- Height = $m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved.
- Instead of re-computing, store in a table.

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 m and n is only mn.

Dynamic programming algorithm

COMPUTING the LENGTH of LCS

```
LCS-LENGTH(X,Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i,0] ← 0
for j ← 0 to n
    do c[0,j] ← 0
for i ← 1 to m
    do for j ← 1 to n
           do if x[i] = y[j]
                  then c[i,j] ← c[i-1,j-1]+1
                       b[i,j] ←"↖"
                  else if c[i-1,j] ≥ c[i,j-1]
                          then c[i,j] ← c[i-1,j]
                               b[i,j] ← "↑"
                          else c[i,j] ← c[i,j-1]
                               b[i,j] ← "←"
return c and b
```

Computing the length of an LCS

- The sequences are X = <*A*, *B*, *C*, *B*, *D*, *A*, *B*> and Y = <*B*, *D*, *C*, *A*, *B*, *A*>

Compute $c[i, j]$ row by row for $i = 1..m$, $j = 1..n$.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space = $\Theta(mn)$.

i \ j		0	1	2	3	4	5	6
	y_j		*B*	*D*	*C*	*A*	*B*	*A*
0	x_i	0	0	0	0	0	0	0
1	*A*	0	↑0	↑0	↑0	↖1	←1	↖1
2	*B*	0	↖1	←1	←1	↑1	↖2	←2
3	*C*	0	↑1	↑1	↖2	←2	↑2	↑2
4	*B*	0	↖1	↑1	↑2	↑2	↖3	←3
5	*D*	0	↑1	↖2	↑2	↑2	↑3	↑3
6	*A*	0	↑1	↑2	↑2	↖3	↑3	↖4
7	*B*	0	↖1	↑2	↑2	↑3	↖4	↑4

Constructing an LCS

CONSTRUCTING an LCS

```
PRINT-LCS(b,X,i,j)
if i = 0 or j = 0
  then return
if b[i,j] = "↖"
  then PRINT-LCS(b,X,i-1,j-1)
    print x[i]
else if b[i,j] = "↑"
   then PRINT-LCS(b,X,i-1,j)
else PRINT-LCS(b,X,i,j-1)
```

- The procedure takes time $O(m + n)$.

Memoization algorithm

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

COMPUTING the LENGTH of LCS

```
MEMOI-LCS(X,Y,i,j)
if c[i,j] = NIL
  then if x[i] = y[j]
          then c[i,j] ← MEMOI-LCS(X,Y,i-1,j-1)+1
          else c[i,j] ← max{MEMOI-LCS(X,Y,i-1,j),
                            MEMOI-LCS(X,Y,i,j-1)}
return c[i,j]
```

same as before

Memoization algorithm

- Assuming that initially $\forall i, j$: $c[i, j]$ = NIL, to get the length of LCS of X and Y, MEMOI-LCS(X, Y, length[X], length[Y]) should be called.

COMPUTING the LENGTH of LCS

```
MEMOIZED-LCS(X,Y)
  for i ← 1 to m
     do for j ← i to n
           do c[i,j] ← NIL
  return MEMOI-LCS(X,Y,m,n)
```

- Time = $\Theta(mn)$ = constant work per table entry.
- Space = $\Theta(mn)$.