Dynamic Programming

- Dynamic Programming is an algorithmic technique
- It's somewhat related to divide-and-conquer and greedy alg.
- In divide-and-conquer, each subproblem occurs only once, so solution can be efficient using recursion.
- In greedy alg. (will see later) the subproblems are determined by taking a step that is also part of the overall solution.

  e.g. Making optimal change using {1¢, 5¢, 10¢, 25¢}

- So we use dynamic Programming when a divide-and-conquer approach would cause repeated solution of the same subproblem, and no locally optimal step is possible that leads to globally optimal solution.

We will go over the ideas in D.P by considering specific examples:

Longest Common Subsequence LCS.

Given two strings $x[1..m]$, $y[1..n]$, find a longest sequence that is common to both

x: A B C B D A B

y: B D C A B A

BCBA is contained in both and is LCS.

Applications: • Computational Biology. DNA, RNA, or protein sequences, identify similarities.

• Unix command "diff" compares lines of files.

To appreciate an efficient solution to this problem, consider a Brute_force approach.

For every subsequence of x, check if it's a subsequence of y

What's the running time of this approach? There are 2^m subsequences of x to check; each take $\Theta(n)$ time (scan y for first element, scan from there for next element, ...)

So running time is $\Theta(2^m n)$ [if $m > n$, we can optimize by exchanging the roles of x and y]

Approach: 1) focus on the length of LCS

2) Extend alg. to find LCS itself

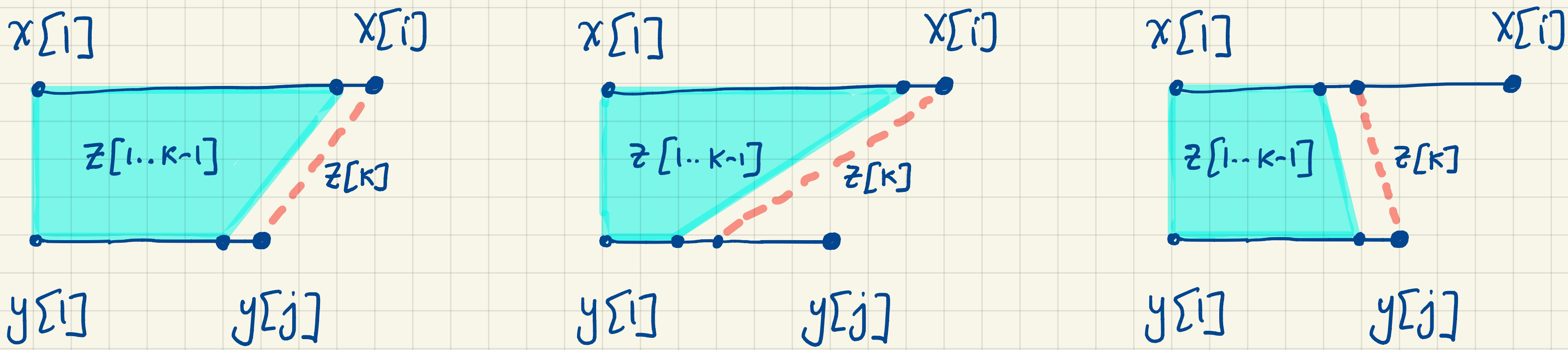
Typical D.P.

- Define $c[i,j]$ = length of LCS of $x[1..i]$ and $y[1..j]$
- Then $c[m,n]$ = length of LCS of x and y
- Claim:

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j] \quad (\text{case 1}) \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

Proof of Case 1: Let $z[1..k]$ be LCS of $x[1..i]$ and $y[1..j]$, $c[i,j] = k$.

We have 3 scenarios:



otherwise $z[1..k]$ can be extended by making $z[k+1] = x[i] = y[j]$

In all cases $z[1..k-1]$ is CS of $x[1..i-1]$ and $y[1..j-1]$

We can show that $z[1..k-1]$ is LCS of $x[1..i-1]$ and $y[1..j-1]$

Proof: [Cut-and-Paste argument]

Suppose not, then $\exists$ $w[1..k']$ that is LCS of above and $k' > k-1$. Cut z paste w.

We can then extend w by $w[k'+1] = x[i] = y[j]$.

obtaining a CS of $x[1..i]$ and $y[1..j]$ of length $> k$, a contradiction. This proves $c[i-1, j-1] = k-1$.

then $c[i,j] = c[i-1, j-1] + 1$

Equivalent formulation :

$$c[i,j] = \max \begin{cases} c[i-1,j-1] + w(i,j) \\ c[i,j-1] - \gamma \\ c[i-1,j] - \gamma \end{cases}$$

$$c[k,0] = c[0,k] = -k\gamma \qquad (k \geq 0)$$

where $w(i,j) = \begin{cases} 1 & x[i] = y[j] \\ -\infty & \text{otherwise.} \end{cases}$, and $\gamma = 0$ for LCS

But one can generalize the scoring scheme to obtain "similar" subsequences (Not exact matches) where gaps are penalized by γ.

Dynamic Programming : Hallmark #1

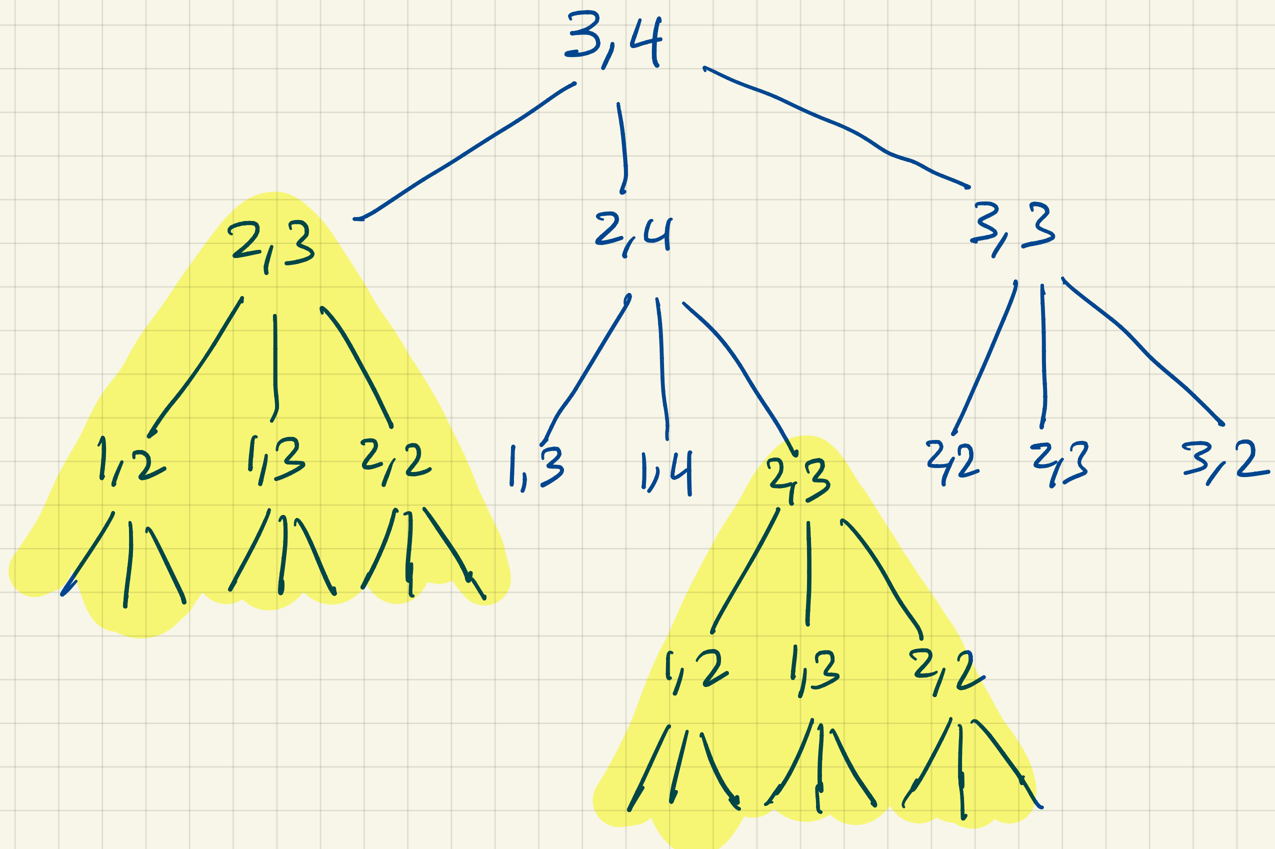
Optimal Substructure: An optimal solution to a problem, contains optimal solutions to subproblems.

Here, $c[i,j]$ is expressed in terms of $c[i-1,j-1]$, $c[i,j-1]$, and $c[i-1,j]$

Optimal Substructure $\Longrightarrow$ Recursive Solution.

```
LCS(x, y, i, j)
    if i=0 or j=0
        then return 0            ▷ base case
    return max( LCS(i-1, j-1) + S(i,j),
                LCS(i, j-1),
                LCS(i-1, j))
```

Recursive tree: $(m=3,\ n=4)$



Depth of any leaf $\geq m$ (assuming $m \leq n$)

Branches by 3 at each node, so amount of work $\Omega(3^m)$

Dynamic Programming: Hallmark #2

Overlapping Subproblems

- There are only few subproblems, here $\Theta(mn)$.
- Many recurring instances of each

  [unlike Divide-and-Conquer where problems are independent]

Solution: Memoization.

After computing solution to subproblem store it in "table" to avoid redoing work.

Conceptual Implementation

Top down approach

LCS(x, y, i, j)

if $i = 0$ or $j = 0$

then return 0

if $c[i,j] = NIL$

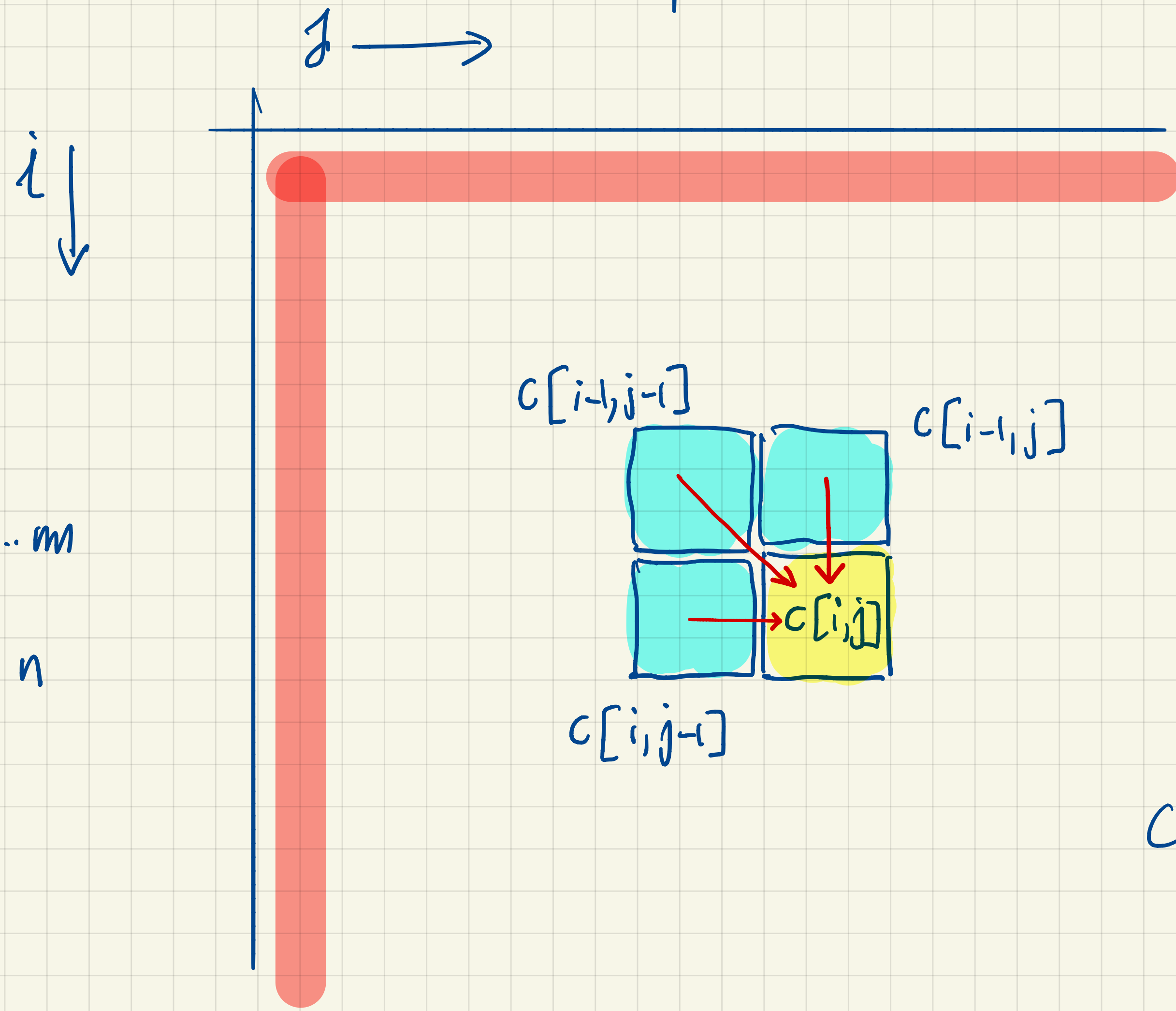
then $c[i,j] \leftarrow \max(\quad , \quad , \quad)$

return $c[i,j]$

else return $c[i,j]$

In practice, we use a **bottom up approach** -

start with small sub problems and work toward larger.



$i = 0 \ldots m$

$j = 0 \ldots n$

Time $\Theta(mn)$

Space $\Theta(mn)$

Compute table row by row

$j \rightarrow$

$i \downarrow$

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

$c[6,7] = 4$

Actual LSC are obtained by "backtracking", which is a typical aspect of Dynamic Programming.

Optimal BST

- Assume $k_1 < k_2 < \dots < k_n$ are keys with access prob. P_i, $i = 1..n$

- Construct a BST that minimizes

$$\sum_{i=1}^{n} [1 + d(k_i)] \, p_i$$

where $d(k_i)$ is the depth of k_i.

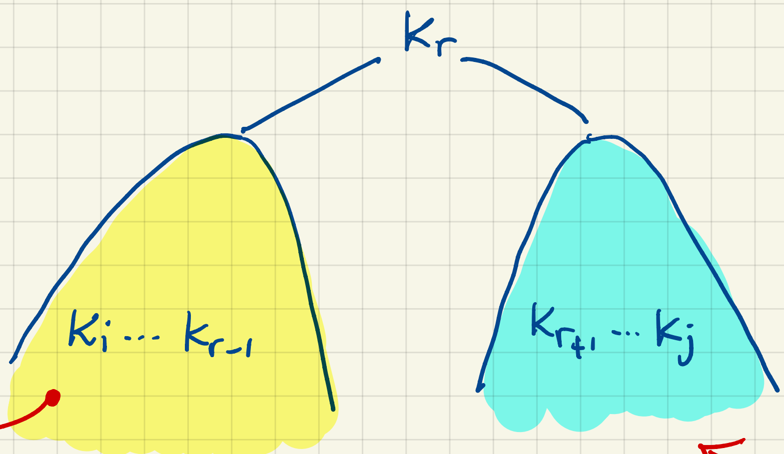
[simplification of book version]: all searches are always for $\{k_1, k_2, \dots, k_n\}$. So $\sum_{i=1}^{n} P_i = 1$.

Optimal Substructure:

Let $c[i,j]$ be the expected cost of an optimal tree containing $\{k_i, k_{i+1}, \dots, k_j\}$

If k_r $(i \leqslant r \leqslant j)$ is root of such tree, then

$$c[i,j] = c[i, r-1] + \sum_{\ell=i}^{r-1} p_\ell + p_r + c[r+1, j] + \sum_{\ell=r+1}^{j} p_\ell$$



must be optimal by cut-and-paste argument

depth of a node k_ℓ is $1+$ it's depth in subtree so add p_ℓ to cost.

Construct recursive solution :

$$c[i,j] = \begin{cases} 0 & j = i-1 \text{ (Empty)} \\ \min\limits_{i \le r \le j} c[i,r-1] + c[r+1,j] + w(i,j) & \text{otherwise} \end{cases}$$

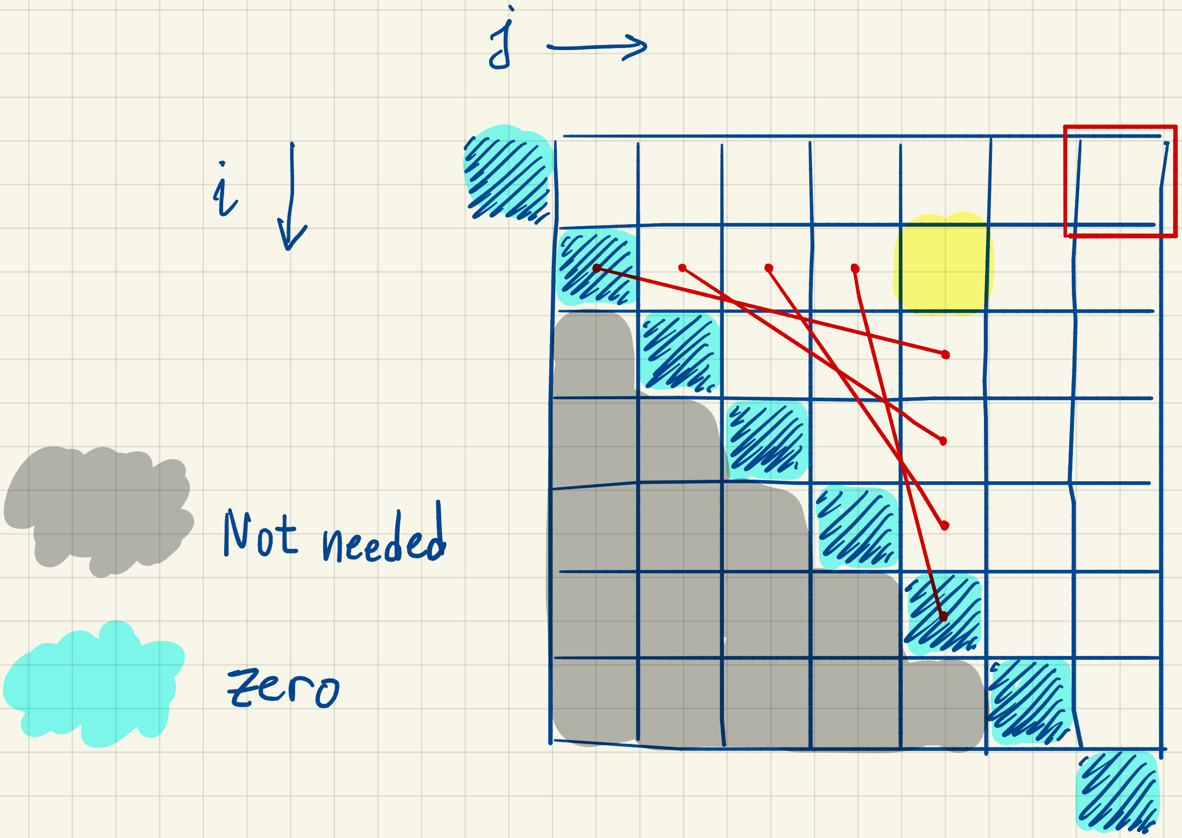
where $w(i,j) = \sum_{l=i}^{j} p_l$

If we use Memoization, and assuming all $w(i,j)$ available, each $c[i,j]$ requires $\Theta(n)$ time to compute, for a total of $\Theta(n^3)$.

Note: $w(i,j)$ can be computed in $O(n^3)$ time trivially, but using DP, it can be done in $\Theta(n^2)$ time.

$$w(i,j) = \begin{cases} 0 & j < i \\ w(i,j-1) + p_j & \text{otherwise} \end{cases}$$

What does the table for c[i,j] look like ?



c[2,5] will check :

c[2,1] + c[3,5]

c[2,2] + c[4,5]

c[2,3] + c[5,5]

c[2,4] + c[6,5]

Actual tree obtained by standard D.P. backtracking.