Dynamic Programming

Longest Common Subsequence

- ***Problem:*** Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$, find a common subsequence whose length is maximum.

springtime — printing

ncaa tournament — north carolina

basketball — krzyzewski

Subsequence need not be consecutive, but must be in order.

Other sequence questions

- ***Edit distance:*** Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$, what is the minimum number of deletions, insertions, and changes that you must do to change one to another?
- ***Protein sequence alignment:*** Given a score matrix on amino acid pairs, $s(a,b)$ for $a,b \in \{\Lambda\} \cup A$, and 2 amino acid sequences, $X = \langle x_1,\ldots,x_m \rangle \in A^m$ and $Y = \langle y_1,\ldots,y_n \rangle \in A^n$, find the alignment with lowest score…

More problems

Optimal BST: Given sequence $K = k_1 < k_2 < \cdots < k_n$ of n sorted keys, with a search probability p_i for each key k_i, build a binary search tree (BST) with minimum expected search cost.

Matrix chain multiplication: Given a sequence of matrices $A_1\ A_2\ \ldots\ A_n$, with A_i of dimension $m_i \times n_i$, insert parenthesis to minimize the total number of scalar multiplications.

Minimum convex decomposition of a polygon,

Hydrogen placement in protein structures, …

Dynamic Programming

- Dynamic Programming is an algorithm design technique for optimization problems: often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - » Subproblems may share subsubproblems,
  - » However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
- DP reduces computation by
  - » Solving subproblems in a bottom-up fashion.
  - » Storing solution to a subproblem the first time it is solved.
  - » Looking up the solution when subproblem is encountered again.
- Key: determine structure of optimal solutions

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

We'll study these with the help of examples.

Longest Common Subsequence

- ***Problem:*** Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$, find a common subsequence whose length is maximum.

springtime	ncaa tournament	basketball
printing	north carolina	snoeyink

Subsequence need not be consecutive, but must be in order.

Naïve Algorithm

- For every subsequence of X, check whether it's a subsequence of Y .
- Time: $\Theta(n2^m)$.
  - » 2^m subsequences of X to check.
  - » Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, for second, and so on.

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. 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}.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Notation:

prefix $X_i = \langle x_1,\ldots,x_i \rangle$ is the first i letters of X.

This says what any longest common subsequence must look like; do you believe it?

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. 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}.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Proof: (case 1: $x_m = y_n$)

Any sequence Z' that does not end in $x_m = y_n$ can be made longer by adding $x_m = y_n$ to the end. Therefore,

(1) longest common subsequence (LCS) Z must end in $x_m = y_n$.

(2) Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}, and

(3) there is no longer CS of X_{m-1} and Y_{n-1}, or Z would not be an LCS.

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k\text{-}1}$ is an LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of $X_{m\text{-}1}$ and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and $Y_{n\text{-}1}$.

Proof: (case 2: $x_m \neq y_n$, and $z_k \neq x_m$)

Since Z does not end in x_m,

(1) Z is a common subsequence of $X_{m\text{-}1}$ and Y, and

(2) there is no longer CS of $X_{m\text{-}1}$ and Y, or Z would not be an LCS.

Recursive Solution

- Define $c[i, j]$ = length of LCS of X_i and Y_j .
- We want $c[m,n]$.

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

This gives a recursive algorithm and solves the problem.
But does it solve it well?

Recursive Solution

$$c[\alpha,\beta]=\begin{cases}0 & \text{if } \alpha \text{ empty or } \beta \text{ empty},\\ c[\mathit{prefix}\alpha,\mathit{prefix}\beta]+1 & \text{if } \mathrm{end}(\alpha)=\mathrm{end}(\beta),\\ \max(c[\mathit{prefix}\alpha,\beta],c[\alpha,\mathit{prefix}\beta]) & \text{if } \mathrm{end}(\alpha)\neq\mathrm{end}(\beta).\end{cases}$$

c[springtime, printing]

c[springtim, printing] *c*[springtime, printin]

[springti, printing] [springtim, printin] ~~[springtim, printin]~~ [springtime, printi]

[springt, printing] [springti, printin] [springtim, printi] [springtime, print]

Recursive Solution

$$c[\alpha,\beta]=\begin{cases}0 & \text{if } \alpha \text{ empty or } \beta \text{ empty},\\ c[\mathit{prefix}\alpha,\mathit{prefix}\beta]+1 & \text{if } \mathrm{end}(\alpha)=\mathrm{end}(\beta),\\ \max(c[\mathit{prefix}\alpha,\beta],c[\alpha,\mathit{prefix}\beta]) & \text{if } \mathrm{end}(\alpha)\neq\mathrm{end}(\beta).\end{cases}$$

- Keep track of $c[\alpha,\beta]$ in a table of *nm* entries:
  - top/down
  - bottom/up

		p	r	i	n	t	i	n	g
s									
p									
r									
i									
n									
g									
t									
i									
m									
e									

Computing the length of an 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
```

$b[i, j\,]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.

$c[m,n]$ contains the length of an LCS of X and Y.

Time: $O(mn)$

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
    elseif b[i, j ] = "↑"
         then PRINT-LCS(b, X, i–1, j)
else PRINT-LCS(b, X, i, j–1)
```

- Initial call is PRINT-LCS (*b*, *X*,*m*, *n*).
- When *b*[*i*, *j*] = ↖, we have extended LCS by one character. So LCS = entries with ↖ in them.
- Time: $O(m+n)$

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

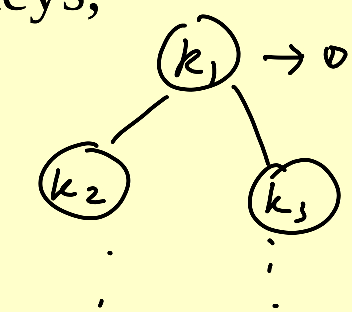
We'll study these with the help of examples.

Optimal Binary Search Trees

- Problem
  - » Given sequence $K = k_1 < k_2 < \cdots < k_n$ of n sorted keys, with a search probability p_i for each key k_i.
  - » Want to build a binary search tree (BST) with minimum expected search cost.
  - » Actual cost = # of items examined.
  - » For key k_i, cost = $\text{depth}_T(k_i)+1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T.

(k_i)

x



$$E[\text{cost}(T)] = \sum_{k\in K} pr(k) \times (d_T(k)+1) = \sum_{k\in K} pr(k)\, d_T(k) + 1$$

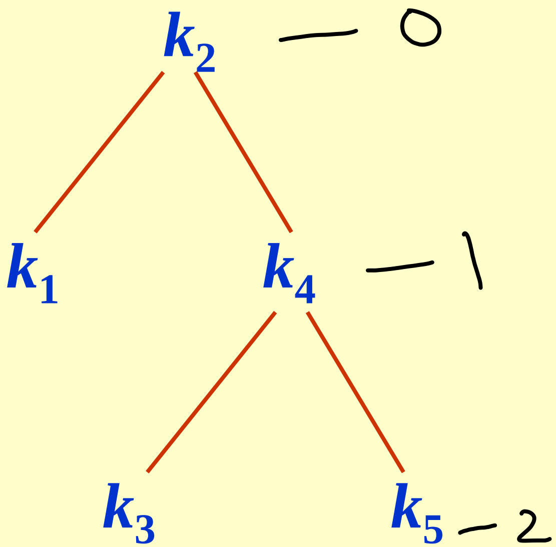
Expected Search Cost

$$
\begin{aligned}
E[\text{search cost in } T] \\
&= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i \\
&= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i \\
&= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i \qquad (15.16)
\end{aligned}
$$

Sum of probabilities is 1.

Example

- Consider 5 keys with these search probabilities:
  $p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$.

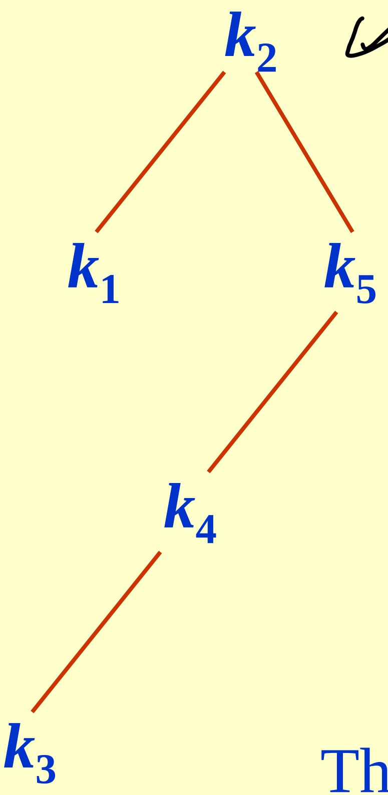


i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i)\cdot p_i$
1	1	0.25
2	0	0
3	2	0.1
4	1	0.2
5	2	0.6
		1.15

Therefore, E[search cost] = 2.15.

Example

- $p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$.



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	3	0.15
4	2	0.4
5	1	0.3
		1.10

Therefore, E[search cost] = 2.10.

This tree turns out to be optimal for this set of keys.

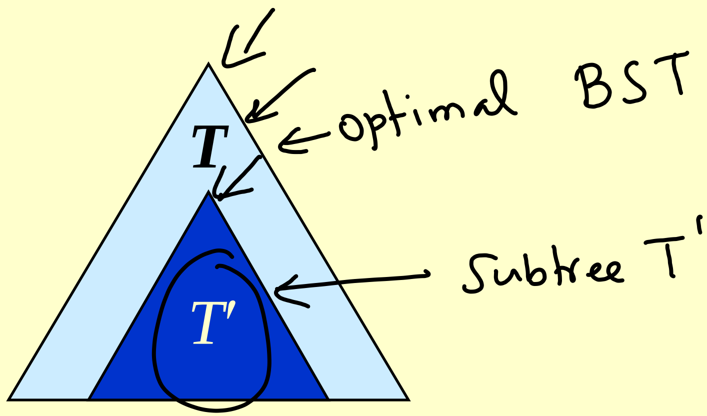
Example

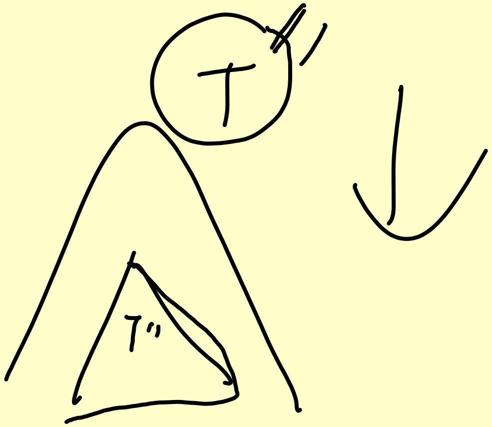
$(k_1 \; k_2 \; k_3 \; k_{r-1}) \; k_r \; (- \quad k_{n-1} \quad k_n)$

- **Observations:**
  - » Optimal BST may not have smallest height.
  - » Optimal BST may not have highest-probability key at root.
- Build by exhaustive checking?
  - » Construct each *n*-node BST.
  - » For each,
    assign keys and compute expected search cost.
  - » But there are $\Omega(4^n/n^{3/2})$ different BSTs with *n* nodes.

Optimal Substructure

- Any subtree of a BST contains keys in a contiguous range $k_i, \ldots, k_j$ for some $1 \le i \le j \le n$.



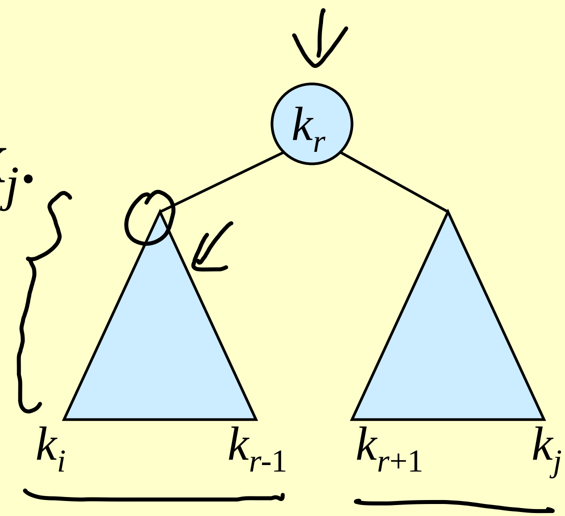


For T to be optimal, T′ must also be optimal

- If T is an optimal BST and
  T contains subtree T' with keys $k_i, \ldots, k_j$,
  then T' must be an optimal BST for keys $k_i, \ldots, k_j$.
- **Proof:** Cut and paste.

Optimal Substructure

- One of the keys in $k_i, \ldots, k_j$, say k_r, where $i \leq r \leq j$, must be the root of an optimal subtree for these keys.
- Left subtree of k_r contains $k_i, \ldots, k_{r-1}$.
- Right subtree of k_r contains $k_r+1, \ldots, k_j$.



- To find an optimal BST:
  - » Examine all candidate roots k_r , for $i \leq r \leq j$
  - » Determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$

Recursive Solution

- Find optimal BST for $k_i,\ldots,k_j$, where $i \geq 1$, $j \leq n$, $j \geq i-1$. When $j = i-1$, the tree is empty.

  j

  $(i, i-1)$

- Define $e[i, j\,]$ = expected search cost of optimal BST for $k_i,\ldots,k_j$.

- If $j = i-1$, then $e[i, j\,] = 0$.
- If $j \geq i$,
  - » Select a root k_r, for some $i \leq r \leq j$.
  - » Recursively make an optimal BSTs
    - for $k_i,\ldots,k_{r-1}$ as the left subtree, and
    - for $k_{r+1},\ldots,k_j$ as the right subtree.

$j-i+1$

$j-r < j-i+1$

Recursive Solution

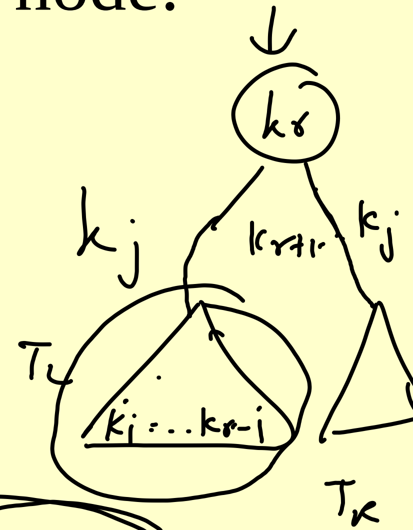
- When the OPT subtree becomes a subtree of a node:
  - » Depth of every node in OPT subtree goes up by 1.
  - » Expected search cost increases by

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

cumulative density

from (15.16)

k_i . . . k_j



- If k_r is the root of an optimal BST for $k_i,..,k_j$:
  - » $e[i, j\,] = p_r + (e[i, r{-}1] + w(i, r{-}1)) + (e[r{+}1, j] + w(r{+}1, j))$

$= e[i, r{-}1] + e[r{+}1, j] + w(i, j).$ (because $w(i, j) = w(i, r{-}1) + p_r + w(r + 1, j)$)

- But, we don't know k_r. Hence,

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

Computing an Optimal Solution

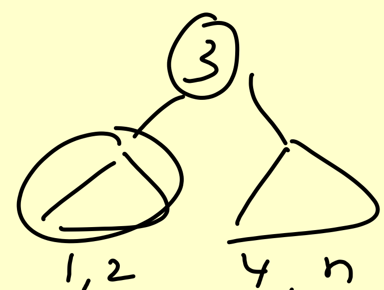
For each subproblem (i,j), store:

$e[1,0]$

- expected search cost in a table $e[1 \,..\, n+1 \,,\, 0 \,..\, n]$
  - » Will use only entries $e[i, j]$, where $j \geq i-1$.
- $\text{root}[i, j]$ = root of subtree with keys $k_i,..,k_j$, for $1 \leq i \leq j \leq n$.
- $\underline{w[1..n+1, 0..n]}$ = sum of probabilities
  - » $w[i, i-1] = 0$ for $1 \leq i \leq n$.
  - » $w[i, j] = w[i, j\text{-}1] + p_j$ for $1 \leq i \leq j \leq n$.

$j \geq i-1$

$i \,..\, n$



root $(1, n) = 3$

$r[1,2] = 1$

Pseudo-code

```
OPTIMAL-BST(p, q, n)
for i ← 1 to n + 1
    do e[i, i–1] ← 0
       w[i, i–1] ← 0
for l ← 1 to n
    do for i ← 1 to n–l + 1
        do j ←i + l–1
           e[i, j ]←∞
           w[i, j ] ← w[i, j–1] + p_j
           for r ←i to j
               do t ← e[i, r–1] + e[r + 1, j ] + w[i, j ]
                  if t < e[i, j ]
                     then e[i, j ] ← t
                          root[i, j ] ←r
return e and root
```

Consider all trees with *l* keys.

Fix the first key.

Fix the last key

Determine the root of the optimal (sub)tree

Time: $O(n^3)$

Elements of Dynamic Programming

- Optimal substructure
- Overlapping subproblems

Optimal Substructure

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste.
- Need to ensure that a wide enough range of choices and subproblems are considered.

Optimal Substructure

- Optimal substructure varies across problem domains:
  - » 1. *How many subproblems* are used in an optimal solution.
  - » 2. *How many choices* in determining which subproblem(s) to use.
- Informally, running time depends on (# of subproblems overall) × (# of choices).
- How many subproblems and choices do the examples considered contain?
- Dynamic programming uses optimal substructure **bottom up**.
  - » *First* find optimal solutions to subproblems.
  - » *Then* choose which to use in optimal solution to the problem.

Optimal Substucture

- Does optimal substructure apply to all optimization problems? No.
- Applies to determining the shortest path but NOT the longest simple path of an unweighted directed graph.
- Why?
  - » Shortest path has independent subproblems.
  - » Solution to one subproblem does not affect solution to another subproblem of the same problem.
  - » Subproblems are not independent in longest simple path.
    - Solution to one subproblem affects the solutions to other subproblems.
  - » Example:

Overlapping Subproblems

- The space of subproblems must be “small”.
- The total number of distinct subproblems is a polynomial in the input size.
  - » A recursive algorithm is exponential because it solves the same problems repeatedly.
  - » If divide-and-conquer is applicable, then each problem solved will be brand new.