Dynamic Programming

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when “programming” meant “tabular method” (like linear programming). Doesn’t really refer to computer programming.
- Used for optimization problems:
  - Find a solution with the optimal value.
  - Minimization or maximization. (We’ll see both.)

Four-step method

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Assembly-line scheduling

A simple dynamic-programming example. Actually, solvable by a graph algorithm that we’ll see later in the course. But a good warm-up for dynamic programming.

[New in the second edition of the book.]
Automobile factory with two assembly lines.

- Each line has $n$ stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$.
- Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$.
- Entry times $e_1$ and $e_2$.
- Exit times $x_1$ and $x_2$.
- After going through a station, can either
  - stay on same line; no cost, or
  - transfer to other line; cost after $S_{i,j}$ is $t_{i,j}$. ($j = 1, \ldots, n - 1$. No $t_{i,n}$, because the assembly line is done after $S_{i,n}$.)

**Problem:** Given all these costs (time = cost), what stations should be chosen from line 1 and from line 2 for fastest way through factory?

Try all possibilities?

- Each candidate is fully specified by which stations from line 1 are included. Looking for a subset of line 1 stations.
- Line 1 has $n$ stations.
- $2^n$ subsets.
- Infeasible when $n$ is large.

**Structure of an optimal solution**

Think about fastest way from entry through $S_{1,j}$.

- If $j = 1$, easy: just determine how long it takes to get through $S_{1,1}$.
- If $j \geq 2$, have two choices of how to get to $S_{1,j}$:
  - Through $S_{1,j-1}$, then directly to $S_{1,j}$.
  - Through $S_{2,j-1}$, then transfer over to $S_{1,j}$.

Suppose fastest way is through $S_{1,j-1}$.
Key observation: We must have taken a fastest way from entry through $S_{1,j-1}$ in this solution. If there were a faster way through $S_{1,j-1}$, we would use it instead to come up with a faster way through $S_{1,j}$.

Now suppose a fastest way is through $S_{2,j-1}$. Again, we must have taken a fastest way through $S_{2,j-1}$. Otherwise use some faster way through $S_{2,j-1}$ to give a faster way through $S_{1,j}$.

Generally: An optimal solution to a problem (fastest way through $S_{i,j}$) contains within it an optimal solution to subproblems (fastest way through $S_{i-1,j}$ or $S_{i,j-1}$).

This is optimal substructure.

Use optimal substructure to construct optimal solution to problem from optimal solutions to subproblems.

Fastest way through $S_{1,j}$ is either
- fastest way through $S_{1,j-1}$ then directly through $S_{1,j}$, or
- fastest way through $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$.

Symmetrically:

Fastest way through $S_{2,j}$ is either
- fastest way through $S_{2,j-1}$ then directly through $S_{2,j}$, or
- fastest way through $S_{1,j-1}$, transfer from line 1 to line 2, then through $S_{2,j}$.

Therefore, to solve problems of finding a fastest way through $S_{1,j}$ and $S_{2,j}$, solve subproblems of finding a fastest way through $S_{i,j-1}$ and $S_{i,j-1}$.

Recursive solution

Let $f_i[j] =$ fastest time to get through $S_{i,j}$, $i = 1, 2$ and $j = 1, \ldots, n$.

Goal: fastest time to get all the way through $S_{i,j} = f^*$.

\[
f^* = \min (f_1[n] + x_1, f_2[n] + x_2)
\]

\[
f_1[1] = e_1 + a_{1,1}
\]

\[
f_2[1] = e_2 + a_{2,1}
\]

For $j = 2, \ldots, n$:

\[
f_1[j] = \min (f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})
\]

\[
f_2[j] = \min (f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j})
\]

$f_i[j]$ gives the value of an optimal solution. What if we want to construct an optimal solution?

- $l_i[j] =$ line # (1 or 2) whose station $j-1$ is used in fastest way through $S_{i,j}$.
- In other words $S_{l_i[j],j-1}$ precedes $S_{i,j}$.
- Defined for $i = 1, 2$ and $j = 2, \ldots, n$.
- $l^* =$ line # whose station $n$ is used.
For example:

\[
\begin{array}{c|ccccc}
  j & 1 & 2 & 3 & 4 & 5 \\
  \hline
  f_1[l] & 9 & 18 & 20 & 24 & 32 \\
  f_2[l] & 12 & 16 & 22 & 25 & 30 \\
\end{array}
\quad \begin{array}{c|c}
  j & 2 & 3 & 4 & 5 \\
  \hline
  l_1[l] & 1 & 2 & 1 & 1 \\
  l_2[l] & 1 & 2 & 1 & 2 \\
\end{array}
\]

\[f^* = 35\quad l^* = 1\]

Go through optimal way given by \(l\) values. (Shaded path in earlier figure.)

**Compute an optimal solution**

Could just write a recursive algorithm based on above recurrences.

- Let \(r_i(j) = \#\) of references made to \(f_i[j]\).
- \(r_1(n) = r_2(n) = 1\).
- \(r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)\) for \(j = 1, \ldots, n-1\).

**Claim**

\(r_i(j) = 2^{n-j}\).

**Proof**  Induction on \(j\), down from \(n\).

**Basis:** \(j = n\). \(2^{n-j} = 2^0 = 1 = r_i(n)\).

**Inductive step:** Assume \(r_i(j+1) = 2^{n-(j+1)}\).

Then \(r_i(j) = r_i(j+1) + r_2(j+1)\)

\[
= 2^{n-(j+1)} + 2^{n-(j+1)} \\
= 2^{n-(j+1)+1} \\
= 2^{n-j}.
\]

**Therefore, \(f_1[1]\) alone is referenced \(2^{n-1}\) times!**

So top down isn’t a good way to compute \(f_i[j]\).

**Observation:** \(f_i[j]\) depends only on \(f_i[j-1]\) and \(f_2[j-1]\) (for \(j \geq 2\)).

So compute in order of increasing \(j\).
FASTEST-WAY \((a, t, e, x, n)\)

\[ f_1[1] \leftarrow e_1 + a_{1,1} \]
\[ f_2[1] \leftarrow e_2 + a_{2,1} \]

for \(j \leftarrow 2\) to \(n\)

\[ \text{do if } f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j} \]
\[ \text{then } f_1[j] \leftarrow f_1[j-1] + a_{1,j} \]
\[ l_1[j] \leftarrow 1 \]
\[ \text{else } f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j} \]
\[ l_1[j] \leftarrow 2 \]
\[ \text{if } f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j} \]
\[ \text{then } f_2[j] \leftarrow f_2[j-1] + a_{2,j} \]
\[ l_2[j] \leftarrow 2 \]
\[ \text{else } f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j} \]
\[ l_2[j] \leftarrow 1 \]

if \(f_1[n] + x_1 \leq f_2[n] + x_2\)
\[ \text{then } f^* = f_1[n] + x_1 \]
\[ l^* = 1 \]
\[ \text{else } f^* = f_2[n] + x_2 \]
\[ l^* = 2 \]

Go through example.

Constructing an optimal solution

PRINT-STATIONS \((l, n)\)

\[ i \leftarrow l^* \]

print “line” \(i\), “station” \(n\)

for \(j \leftarrow n\) down to 2

\[ \text{do } i \leftarrow l_1[j] \]

print “line” \(i\), “station” \(j - 1\)

Go through example.

Time = \(\Theta(n)\)

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 subsequence common to both whose length is longest. A subsequence doesn’t have to be consecutive, but it has to be in order.

[To come up with examples of longest common subsequences, search the dictionary for all words that contain the word you are looking for as a subsequence. On a UNIX system, for example, to find all the words with pine as a subsequence, use the command `grep ’.*p.*i.*n.*e.*’ dict`, where `dict` is your local dictionary. Then check if that word is actually a longest common subsequence. Working C code for finding a longest common subsequence of two strings appears at http://www.cs.dartmouth.edu/˜thc/code/lcs.c]
Examples: [The examples are of different types of trees.]

.springtime  .horseback
  pioneer  .snowflake

.maelstrom  .heroically
  becalm  .scholarly

Brute-force algorithm:
For every subsequence of X, check whether it’s a subsequence of Y.
Time: \(\Theta(n^2m)\).
- \(2^m\) subsequences of X to check.
- Each subsequence takes \(\Theta(n)\) time to check: scan Y for first letter, from there scan for second, and so on.

Optimal substructure

Notation:
\(X_i = \text{prefix } \langle x_1, \ldots, x_i \rangle\)
\(Y_i = \text{prefix } \langle y_1, \ldots, y_i \rangle\)

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 \(z_k \neq x_m \Rightarrow Z\) is an LCS of \(X_{m-1}\) and \(Y\).
3. If \(x_m \neq y_n\), then \(z_k \neq y_n \Rightarrow Z\) is an LCS of \(X\) and \(Y_{n-1}\).

Proof
1. First show that \(z_k = x_m = y_n\). Suppose not. Then make a subsequence \(Z' = \langle z_1, \ldots, z_k, x_m \rangle\). It’s a common subsequence of X and Y and has length \(k + 1\) \(\Rightarrow Z'\) is a longer common subsequence than \(Z \Rightarrow Z\) being an LCS.
Now show \(Z_{k-1}\) is an LCS of \(X_{m-1}\) and \(Y_{n-1}\). Clearly, it’s a common subsequence. Now suppose there exists a common subsequence \(W\) of \(X_{m-1}\) and \(Y_{n-1}\) that’s longer than \(Z_{k-1} \Rightarrow \text{length of } W \geq k\). Make subsequence \(W'\) by appending \(x_m\) to \(W\). \(W'\) is common subsequence of \(X\) and \(Y\), has length \(\geq k + 1 \Rightarrow Z\) being an LCS.
2. If \(z_k \neq x_m\), then \(Z\) is a common subsequence of \(X_{m-1}\) and \(Y\). Suppose there exists a subsequence \(W\) of \(X_{m-1}\) and \(Y\) with length \(> k\). Then \(W\) is a common subsequence of \(X\) and \(Y \Rightarrow Z\) being an LCS.
3. Symmetric to 2. \( \blacksquare \) (theorem)

Therefore, an LCS of two sequences contains as a prefix an LCS of prefixes of the sequences.

**Recursive formulation**

Define \( c[i, j] = \text{length of LCS of } X_i \text{ and } Y_j \). We want \( c[m, n] \).

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

Again, we could write a recursive algorithm based on this formulation.

Try with bozo, bat.

- Lots of repeated subproblems.
- Instead of recomputing, store in a table.

**Compute length of optimal solution**

\( \text{LCS-LENGTH}(X, Y, m, n) \)

\[
\text{for } i \leftarrow 1 \text{ \textbf{to} } m \\
\quad \text{do } c[i, 0] \leftarrow 0 \\
\text{for } j \leftarrow 0 \text{ \textbf{to} } n \\
\quad \text{do } c[0, j] \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ \textbf{to} } m \\
\quad \text{for } j \leftarrow 1 \text{ \textbf{to} } n \\
\quad \text{do } \text{ if } x_i = y_j \\
\qquad \text{then } c[i, j] \leftarrow c[i - 1, j - 1] + 1 \\
\qquad \text{b}[i, j] \leftarrow \text{"\uparrow"} \\
\qquad \text{else if } c[i - 1, j] \geq c[i, j - 1] \\
\qquad \text{then } c[i, j] \leftarrow c[i, j - 1] \\
\qquad \text{b}[i, j] \leftarrow \text{"\<\"} \\
\qquad \text{else } c[i, j] \leftarrow c[i, j - 1] \\
\qquad \text{b}[i, j] \leftarrow \text{"\<\"} \\
\text{return } c \text{ and } b
\]
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).
• b[i, j] points to table entry whose subproblem we used in solving LCS of X_i and Y_j.
• When b[i, j] = “↖”, we have extended LCS by one character. So longest common subsequence = entries with “↖” in them.

Demonstration: show only c[i, j]:

```
 a m p u t a t i o n
 0—0—0 0 0 0 0 0 0 0 0 0
s 0 0 0 0 0 0 0 0 0 0 0
p 0 0 0 1—1—1 1 1 1 1 1
a 0 1 1 1 1 1 2 2 2 2
n 0 1 1 1 1 1 2 2 2 3
k 0 1 1 1 1 1 2 2 2 3
i 0 1 1 1 1 1 2 3 3 3
n 0 1 1 1 1 1 2 3 3 4
g 0 1 1 1 1 1 2 3 3 4
```

Time: Θ(mn)

Optimal binary search trees

[Also new in the second edition.]

• Given sequence K = ⟨k_1, k_2, ..., k_n⟩ of n distinct keys, sorted (k_1 < k_2 < \ldots < k_n).
• Want to build a binary search tree from the keys.
• For k_i, have probability p_i that a search is for k_i.
• Want 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 $T$.

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

$$= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i \quad \text{(since probabilities sum to 1)} \quad (*)$$

[Similar to optimal BST problem in the book, but simplified here: we assume that all searches are successful. Book has probabilities of searches between keys in tree.]

<table>
<thead>
<tr>
<th>$i$</th>
<th>$1$</th>
<th>$2$</th>
<th>$3$</th>
<th>$4$</th>
<th>$5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>.25</td>
<td>.2</td>
<td>.05</td>
<td>.2</td>
<td>.3</td>
</tr>
</tbody>
</table>

Example:

```
  k_2
 /   \
k_1   k_4
   /    /
  k_3   k_5
```

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{depth}_T(k_i)$</th>
<th>$\text{depth}_T(k_i) \cdot p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td>1</td>
<td>.25</td>
</tr>
<tr>
<td>$2$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$3$</td>
<td>2</td>
<td>.1</td>
</tr>
<tr>
<td>$4$</td>
<td>1</td>
<td>.2</td>
</tr>
<tr>
<td>$5$</td>
<td>2</td>
<td>.6</td>
</tr>
</tbody>
</table>

Therefore, $E[\text{search cost}] = 2.15$. 
Therefore, $E[\text{search cost}] = 2.10$, which turns out to be optimal.

**Observations:**

- Optimal BST might not have smallest height.
- Optimal BST might not have highest-probability key at root.

Build by exhaustive checking?

- Construct each $n$-node BST.
- For each, put in keys.
- Then compute expected search cost.
- But there are $\Omega(4^n/n^{3/2})$ different BSTs with $n$ nodes.

**Optimal substructure**

Consider any subtree of a BST. It contains keys in a contiguous range $k_i, \ldots, k_j$ for some $1 \leq i \leq j \leq n$.

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.

Use optimal substructure to construct an optimal solution to the problem from optimal solutions to subproblems:
• Given keys $k_i, \ldots, k_j$ (the problem).
• One of them, $k_r$, where $i \leq r \leq j$, must be the root.
• Left subtree of $k_r$ contains $k_i, \ldots, k_{r-1}$.
• Right subtree of $k_r$ contains $k_{r+1}, \ldots, k_j$.

\[
\begin{array}{c}
k_r \\
k_i \quad k_{r-1} \quad k_{r+1} \quad k_j
\end{array}
\]

• If
  • we examine all candidate roots $k_r$, for $i \leq r \leq j$, and
  • we determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$,

then we’re guaranteed to find an optimal BST for $k_i, \ldots, k_j$.

**Recursive solution**

Subproblem domain:
• 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.

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 $i \leq r \leq j$.
• Make an optimal BST with $k_i, \ldots, k_{r-1}$ as the left subtree.
• Make an optimal BST with $k_{r+1}, \ldots, k_j$ as the right subtree.
• Note: when $r = i$, left subtree is $k_i, \ldots, k_{i-1}$; when $r = j$, right subtree is $k_{j+1}, \ldots, k_j$.

When a subtree becomes a subtree of a node:
• Depth of every node in subtree goes up by 1.
• Expected search cost increases by

\[
w(i, j) = \sum_{l=i}^{j} p_l \quad \text{(refer to equation (\*))}.
\]

If $k_r$ is the root of an optimal BST for $k_i, \ldots, k_j$:

\[
e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j)).
\]
But \( w(i, j) = w(i, r - 1) + p_r + w(r + 1, j) \).
Therefore, \( e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j) \).
This equation assumes that we already know which key is \( k_r \).
We don’t.
Try all candidates, and pick the best one:
\[
e[i, j] = \begin{cases} 
0 & \text{if } j = i - 1, \\
\min_{i \leq r \leq j} \{e[i, r - 1] + e[r + 1, j] + w(i, j)\} & \text{if } i \leq j.
\end{cases}
\]
Could write a recursive algorithm . . .

Computing an optimal solution

As “usual,” we’ll store the values in a table:

\[
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
e[1..n + 1, 0..n] \\
can store \\
e[n + 1, n] \quad e[1, 0]
\end{array}
\end{array}
\end{array}
\]

- Will use only entries \( e[i, j] \), where \( j \geq i - 1 \).
- Will also compute
  \[ \text{root}[i, j] = \text{root of subtree with keys } k_i, \ldots, k_j, \text{for } 1 \leq i \leq j \leq n. \]

One other table . . . don’t recompute \( w(i, j) \) from scratch every time we need it.
(Would take \( \Theta(j - i) \) additions.)
Instead:

- Table \( w[1..n + 1, 0..n] \)
- \( w[i, i - 1] = 0 \) for \( 1 \leq i \leq n \)
- \( w[i, j] = w[i, j - 1] + p_j \) for \( 1 \leq i \leq j \leq n \)

Can compute all \( \Theta(n^2) \) values in \( O(1) \) time each.

\textbf{OPTIMAL-BST}(p, q, n)

\begin{verbatim}
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
\end{verbatim}
First for loop initializes \( e, w \) entries for subtrees with 0 keys.

Main for loop:
- Iteration for \( l \) works on subtrees with \( l \) keys.
- Idea: compute in order of subtree sizes, smaller (1 key) to larger (\( n \) keys).

For example at beginning:

\[
\begin{array}{cccccc}
  & & & j & & \\
  & i & 0 & 1 & 2 & 3 & 4 & 5 \\
 0 & 1 & 0 & .25 & .65 & .8 & 1.25 & 2.10 \\
 1 & 2 & 0 & .2 & .3 & .75 & 1.35 & \\
 2 & 3 & 0 & 0.05 & 0.3 & .85 & \\
 3 & 4 & 0 & 0 & 0 & .7 & \\
 4 & 5 & 0 & & & .3 & \\
 5 & 6 & 0 & & & & \\
\end{array}
\]

\[
\begin{array}{cccccc}
  & & & j & & \\
  & i & 0 & 1 & 2 & 3 & 4 & 5 \\
 0 & 1 & 0 & .25 & .45 & .5 & .7 & 1.0 \\
 1 & 2 & 0 & .2 & .25 & .45 & .75 & \\
 2 & 3 & 0 & 0.05 & 0.25 & .55 & \\
 3 & 4 & 0 & 0.2 & 0.5 & \\
 4 & 5 & 0 & .3 & \\
 5 & 6 & 0 & & \\
\end{array}
\]

\[
\begin{array}{cccccc}
  & & & j & & \\
  & root & 1 & 2 & 3 & 4 & 5 \\
 1 & 1 & 1 & 1 & 2 & 2 \\
 2 & 2 & 2 & 2 & 4 & \\
 3 & 3 & 4 & 5 & \\
 4 & 4 & 5 & \\
 5 & 5 & \\
\end{array}
\]

**Time:** \( O(n^3) \): for loops nested 3 deep, each loop index takes on \( \leq n \) values. Can also show \( \Omega(n^3) \). Therefore, \( \Theta(n^3) \).

**Construct an optimal solution**

**CONSTRUCT-OPTIMAL-BST** (root)

\[
r \leftarrow \text{root}[1, n] \\
\text{print } \text{“}k\text{”, "is the root"} \\
\text{CONSTRUCT-OPT-SUBTREE}(1, r - 1, r, \text{“left"}, \text{root}) \\
\text{CONSTRUCT-OPT-SUBTREE}(r + 1, n, r, \text{“right"}, \text{root})
\]

**CONSTRUCT-OPT-SUBTREE** (i, j, dir, root)

\[
\text{if } i \leq j
\]

\[
\text{then } t \leftarrow \text{root}[i, j] \\
\text{print } \text{“}k\text{”, "is" } \text{dir} \text{ “child of } k\text{”} \\
\text{CONSTRUCT-OPT-SUBTREE}(i, t - 1, t, \text{“left"}, \text{root}) \\
\text{CONSTRUCT-OPT-SUBTREE}(t + 1, j, t, \text{“right"}, \text{root})
\]
Elements of dynamic programming

Mentioned already:
- optimal substructure
- overlapping subproblems

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution. *We find that students often have trouble understanding the relationship between optimal substructure and determining which choice is made in an optimal solution. One way that helps them understand optimal substructure is to imagine that “God” tells you what was the last choice made in 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:
  - Suppose that one of the subproblem solutions is not optimal.
  - Cut it out.
  - Paste in an optimal solution.
  - Get a better solution to the original problem. Contradicts optimality of problem solution.

That was optimal substructure.

Need to ensure that you consider a wide enough range of choices and subproblems that you get them all. *“God” is too busy to tell you what that last choice really was.* Try all the choices, solve all the subproblems resulting from each choice, and pick the choice whose solution, along with subproblem solutions, is best.

How to characterize the space of subproblems?
- Keep the space as simple as possible.
- Expand it as necessary.

Examples:

Assembly-line scheduling
- Space of subproblems was fastest way from factory entry through stations $S_{1,j}$ and $S_{2,j}$.
- No need to try a more general space of subproblems.

Optimal binary search trees
- Suppose we had tried to constrain space of subproblems to subtrees with keys $k_1, k_2, \ldots, k_j$. 
• An optimal BST would have root $k_r$, for some $1 \leq r \leq j$.
• Get subproblems $k_1, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$.
• Unless we could guarantee that $r = j$, so that subproblem with $k_{r+1}, \ldots, k_j$ is empty, then this subproblem is not of the form $k_1, k_2, \ldots, k_j$.
• Thus, needed to allow the subproblems to vary at “both ends,” i.e., allow both $i$ and $j$ to vary.

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.

• Assembly-line scheduling:
  • 1 subproblem
  • 2 choices (for $S_{i,j}$ use either $S_{i,j-1}$ or $S_{2,j-1}$)
• Longest common subsequence:
  • 1 subproblem
  • Either
    • 1 choice (if $x_i = y_j$, LCS of $X_{i-1}$ and $Y_{j-1}$), or
    • 2 choices (if $x_i \neq y_j$, LCS of $X_{i-1}$ and $Y$, and LCS of $X$ and $Y_{j-1}$)
• Optimal binary search tree:
  • 2 subproblems ($k_1, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$)
  • $j - i + 1$ choices for $k_r$ in $k_1, \ldots, k_j$. Once we determine optimal solutions to subproblems, we choose from among the $j - i + 1$ candidates for $k_r$.

Informally, running time depends on (# of subproblems overall) $\times$ (# of choices).
• Assembly-line scheduling: $\Theta(n)$ subproblems, 2 choices for each $\Rightarrow \Theta(n)$ running time.
• Longest common subsequence: $\Theta(mn)$ subproblems, $\leq 2$ choices for each $\Rightarrow \Theta(mn)$ running time.
• Optimal binary search tree: $\Theta(n^2)$ subproblems, $O(n)$ choices for each $\Rightarrow O(n^3)$ running time.

Dynamic programming uses optimal substructure bottom up.
• First find optimal solutions to subproblems.
• Then choose which to use in optimal solution to the problem.

When we look at greedy algorithms, we’ll see that they work top down: first make a choice that looks best, then solve the resulting subproblem.

Don’t be fooled into thinking optimal substructure applies to all optimization problems. It doesn’t.

Here are two problems that look similar. In both, we’re given an unweighted, directed graph $G = (V, E)$. 


V is a set of vertices.
E is a set of edges.

And we ask about finding a path (sequence of connected edges) from vertex u to vertex v.

Shortest path: find path $u \leadsto v$ with fewest edges. Must be simple (no cycles), since removing a cycle from a path gives a path with fewer edges.

Longest simple path: find simple path $u \leadsto v$ with most edges. If didn’t require simple, could repeatedly traverse a cycle to make an arbitrarily long path.

Shortest path has optimal substructure.

![Diagram](image)

Suppose $p$ is shortest path $u \leadsto v$.
Let $w$ be any vertex on $p$.
Let $p_1$ be the portion of $p$, $u \leadsto w$.
Then $p_1$ is a shortest path $u \leadsto w$.

**Proof** Suppose there exists a shorter path $p'_1$, $u \leadsto w$. Cut out $p_1$, replace it with $p'_1$, get path $u \leadsto w \leadsto v$ with fewer edges than $p$.

Therefore, can find shortest path $u \leadsto v$ by considering all intermediate vertices $w$, then finding shortest paths $u \leadsto w$ and $w \leadsto v$.

Same argument applies to $p_2$.

Does longest path have optimal substructure?

It seems like it should.
It does not.

Consider $q \rightarrow r \rightarrow t = $ longest path $q \leadsto t$. Are its subpaths longest paths?
No!

Subpath $q \leadsto r$ is $q \rightarrow r$.
Longest simple path $q \leadsto r$ is $q \rightarrow s \rightarrow t \rightarrow r$.
Subpath $r \leadsto t$ is $r \rightarrow t$.
Longest simple path $r \leadsto t$ is $r \rightarrow q \rightarrow s \rightarrow t$. 
Not only isn’t there optimal substructure, but we can’t even assemble a legal solution from solutions to subproblems.

Combine longest simple paths:

\[ q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t \]

Not simple!

In fact, this problem is NP-complete (so it probably has no optimal substructure to find.)

What’s the big difference between shortest path and longest path?

- Shortest path has \textit{independent} subproblems.
- Solution to one subproblem does not affect solution to another subproblem of the same problem.
- Longest simple path: subproblems are \textit{not} independent.
- Consider subproblems of longest simple paths \( q \leadsto r \) and \( r \leadsto t \).
- Longest simple path \( q \leadsto r \) uses \( s \) and \( t \).
- Cannot use \( s \) and \( t \) to solve longest simple path \( r \leadsto t \), since if we do, the path isn’t simple.
- But we \textit{have} to use \( t \) to find longest simple path \( r \leadsto t \)!
- Using resources (vertices) to solve one subproblem renders them unavailable to solve the other subproblem.

\[ \text{[For shortest paths, if we look at a shortest path } p_1 \leadsto w \leadsto p_2 \text{, no vertex other than } w \text{ can appear in } p_1 \text{ and } p_2. \text{ Otherwise, we have a cycle.]} \]

Independent subproblems in our examples:

- Assembly line and longest common subsequence
  - 1 subproblem \( \Rightarrow \) automatically independent.
- Optimal binary search tree
  - \( k_i, \ldots, k_{r-1} \) and \( k_{r+1}, \ldots, k_j \) \( \Rightarrow \) independent.

\textbf{Overlapping subproblems}

These occur when a recursive algorithm revisits the same problem over and over. Good divide-and-conquer algorithms usually generate a brand new problem at each stage of recursion.

Example: merge sort
Won’t go through exercise of showing repeated subproblems.
Book has a good example for matrix-chain multiplication.
Alternative approach: memoization

• “Store, don’t recompute.”
• Make a table indexed by subproblem.
• When solving a subproblem:
  • Lookup in table.
  • If answer is there, use it.
  • Else, compute answer, then store it.
• In dynamic programming, we go one step further. We determine in what order we’d want to access the table, and fill it in that way.