Introduction

Dynamic programming is a technique for solving optimization problems
- Key element: Decompose a problem into subproblems, solve them recursively, and then combine the solutions into a final (optimal) solution
- Important component: There are typically an exponential number of subproblems to solve, but many of them overlap
  ⇒ Can re-use the solutions rather than re-solving them
- Number of distinct subproblems is polynomial

Example: Rod Cutting (2)

A company has a rod of length $n$ and wants to cut it into smaller rods to maximize profit
- Have a table telling how much they get for rods of various lengths: A rod of length $i$ has price $p_i$
- The cuts themselves are free, so profit is based solely on the prices charged for of the rods
- If cuts only occur at integral boundaries $1, 2, \ldots, n - 1$, then can make or not make a cut at each of $n - 1$ positions, so total number of possible solutions is $2^{n-1}$

Example: Rod Cutting (3)

Given a rod of length $n$, want to find a set of cuts into lengths $l_1, \ldots, l_k$ (where $l_1 + \cdots + l_k = n$) and $r_n = p_l + \cdots + p_{i_k}$ is maximized
- For a specific value of $n$, can either make no cuts (revenue $= p_n$) or make a cut at some position $i$, then optimally solve the problem for lengths $i$ and $n - i$:
  \[
  r_n = \max (p_n, r_{n-1} + r_2 + r_{n-2}, \ldots, r_i + r_{n-i}, \ldots, r_{n-1} + r_1)
  \]
- Notice that this problem has the optimal substructure property, in that an optimal solution is made up of optimal solutions to subproblems
  - Can find optimal solution if we consider all possible subproblems
  - Alternative formulation: Don’t further cut the first segment:
    \[
    r_n = \max (p_i + r_{n-i})
    \]

Algorithm 1: Cut-Rod($p, n$)

```plaintext
1 if $n == 0$ then
2     return 0
3 $q = -\infty$
4 for $i = 1$ to $n$ do
5     $q = \max (q, p[i] + \text{Cut-Rod}(p, n - i))$
6 end
7 return $q$
```

What is the time complexity?
**Dynamic Programming Algorithm**

- Can save time dramatically by remembering results from prior calls
- Two general approaches:
  - **Top-down with memoization**: Run the recursive algorithm as defined earlier, but before recursive call, check to see if the calculation has already been done and memoized
  - **Bottom-up**: Fill in results for "small" subproblems first, then use these to fill in table for "larger" ones
- Typically have the same asymptotic running time

**Algorithm 3: Bottom-Up-Cut-Rod(p, n)**

1. Allocate \( r[0..n] \)
2. \( r[0] = 0 \)
3. for \( j = 1 \) to \( n \) do
4.     \( q = \infty \)
5.     for \( i = 1 \) to \( n-j \) do
6.         \( q = \max(q, p[i] + r[j-i]) \)
7.     end
8.     \( r[j] = q \)
9. end
10. return \( r[n] \)

First solves for \( n = 0 \), then for \( n = 1 \) in terms of \( r[0] \), then for \( n = 2 \) in terms of \( r[0] \) and \( r[1] \), etc.

**Time Complexity (2)**

Recursion Tree for \( n = 4 \)

![Recursion Tree](image)

Top-Down with Memoization

1. if \( r[n] \geq 0 \) then
2. return \( r[n] \) // \( r \) initialized to all \(-\infty\)
3. if \( n == 0 \) then
4. \( q = 0 \)
5. else
6. \( q = -\infty \)
7. for \( i = 1 \) to \( n \) do
8. \( q = \max(q, p[i] + Memoized-Cut-Rod-Aux(p, n-i, r)) \)
9. end
10. \( r[n] = q \)
11. return \( q \)

Algorithm 2: Memoized-Cut-Rod-Aux(p, n, r)

**Time Complexity**

Subproblem graph for \( n = 4 \)

![Subproblem Graph](image)

Both algorithms take linear time to solve for each value of \( n \), so total time complexity is \( \Theta(n^2) \)
Reconstructing a Solution

- If interested in the set of cuts for an optimal solution as well as the revenue it generates, just keep track of the choice made to optimize each subproblem
- Will add a second array $s$, which keeps track of the optimal size of the first piece cut in each subproblem

Reconstructing a Solution (2)

1. Allocate $r[0 \ldots n]$ and $s[0 \ldots n]$
2. $r[0] = 0$
3. for $j = 1$ to $n$
4. \hspace{1em} $q = \infty$
5. \hspace{1em} for $i = 1$ to $j$
6. \hspace{2em} if $q > r[i] + r[j] - r[i]s[j]$
7. \hspace{2em} $q = r[i] + r[j] - r[i]s[j]$
8. \hspace{1em} $s[j] = i$
9. \hspace{1em} end
10. \hspace{1em} end
11. \hspace{1em} return $r$, $s$

Algorithm 4: Extended-Bottom-Up-Cut-Rod($p$, $n$)

Example:

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r[i]$</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>17</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>[\vdots]</td>
<td></td>
</tr>
<tr>
<td>$s[j]$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

If $n = 10$, optimal solution is no cut; if $n = 7$, then cut once to get segments of sizes 1 and 6

Reconstructing a Solution (3)

Algorithm 5: Print-Cut-Rod-Solution($p$, $n$)

Example:

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r[i]$</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>17</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>[\vdots]</td>
<td></td>
</tr>
<tr>
<td>$s[j]$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

Matrix-Chain Multiplication

- The matrix-chain multiplication problem is to take a chain $\langle A_1, \ldots, A_n \rangle$ of $n$ matrices, where matrix $i$ has dimension $p_{i-1} \times p_i$, and fully parenthesize the product $A_1 \cdots A_n$ so that the number of scalar multiplications is minimized
- Brute force solution is infeasible, since its time complexity is $\Omega(4^n/n^{3/2})$
- Will follow 4-step procedure for dynamic programming:
  - Characterize the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Construct the value of an optimal solution
  - Construct an optimal solution from computed information

Matrix-Chain Multiplication (2)

- Let $A_{i,j}$ be the matrix from the product $A_iA_{i+1} \cdots A_j$
- To compute $A_{i,j}$, must split the product and compute $A_{i,k}$ and $A_{k+1,j}$ for some integer $k$, then multiply the two together
- Cost is the cost of computing each subproduct plus cost of multiplying the two results
- Say that in an optimal parenthesization, the optimal split for $A_iA_{i+1} \cdots A_j$ is at $k$
- Then in an optimal solution for $A_iA_{i+1} \cdots A_j$, the parenthesization of $A_i \cdots A_k$ is itself optimal for the subchain $A_i \cdots A_k$ (if not, then we could do better for the larger chain)
- Similar argument for $A_{k+1} \cdots A_j$
- Thus if we make the right choice for $k$ and then optimally solve the subproblems recursively, we’ll end up with an optimal solution
- Since we don’t know optimal $k$, we’ll try them all
Recursively Defining the Value of an Optimal Solution

- Define $m[i, j]$ as minimum number of scalar multiplications needed to compute $A_{i...j}$.
- (What entry in the $m$ table will be our final answer?)
- Computing $m[i, j]$:  
  1. If $i = j$, then no operations needed and $m[i, j] = 0$ for all $i$.
  2. If $i < j$ and we split at $k$, then optimal number of operations needed is the optimal number for computing $A_{i...k}$ and $A_{k+1...j}$, plus the number to multiply them: 
     
     \[
     m[i, j] = m[i, k] + m[k + 1, j] + p_i \cdot p_{k+1} \cdot p_j
     \]
- Since we don’t know $k$, we’ll try all possible values: 
  
  \[
  m[i, j] = \begin{cases} 
  0 & \text{if } i = j \\
  \min_{k < j}(m[i, k] + m[k + 1, j] + p_i \cdot p_{k+1} \cdot p_j) & \text{if } i < j
  \end{cases}
  \]
- To track the optimal solution itself, define $s[i, j]$ to be the value of $k$ used at each split.

Computing the Value of an Optimal Solution

- As with the rod cutting problem, many of the subproblems we’ve defined will overlap.
- Exploiting overlap allows us to solve only $\Theta(n^2)$ problems (one problem for each $(i, j)$ pair), as opposed to exponential.
- We’ll do a bottom-up implementation, based on chain length.
- Chains of length 1 are trivially solved ($m[i, i] = 0$ for all $i$).
- Then solve chains of length 2, 3, etc., up to length $n$.
- Linear time to solve each problem, quadratic number of problems, yields $O(n^3)$ total time.

Algorithm 6: Matrix-Chain-Order($p$, $n$)

```
allocate $m[1...n, 1...n]$ and $s[1...n, 1...n]$

for $i = 1$ to $n$ do
  $m[i, i] = 0$
  $s[i, i] = i$
end

for $l = 2$ to $n$ do
  for $i = 1$ to $n-l+1$ do
    $j = i + l - 1$
    $m[i, j] = \infty$
    for $k = i$ to $j-1$ do
      $q = m[i, k] + m[k+1, j] + p_i \cdot p_{k+1} \cdot p_j$
      if $q < m[i, j]$ then
        $m[i, j] = q$
        $s[i, j] = k$
      end
    end
  end
end

return $(m, s)$
```

Algorithm 7: Print-Optimal-Parens($s$, $i$, $j$)

```
if $i = j$ then
  print "$ A_i$
else
  print "$$
  PRINT-OPTIMAL-PARENS($s, i, s[i, j]$)
  PRINT-OPTIMAL-PARENS($s, s[i, j]+1, j$)
  print "$$
end
```

Constructing an Optimal Solution from Computed Information

- Cost of optimal parenthesization is stored in $m[1, n]$
- First split in optimal parenthesization is between $s[1, n]$ and $s[1, s[1, n]] + 1$
- Descending recursively, next splits are between $s[1, s[1, n]]$ and $s[1, s[1, n]] + 1$ for left side and between $s[s[1, n] + 1, n]$ and $s[s[1, n] + 1, n] + 1$ for right side
- and so on...
Constructing an Optimal Solution from Computed Information (3)

Optimal parenthesization: \(((A_1(A_2A_3))(A_4A_5)A_6))

Example of How Subproblems Overlap

Entire subtrees overlap:

See Section 15.3 for more on optimal substructure and overlapping subproblems

Longest Common Subsequence

- Sequence \(Z = \langle z_1, z_2, \ldots, z_k \rangle\) is a subsequence of another sequence \(X = \langle x_1, x_2, \ldots, x_m \rangle\) if there is a strictly increasing sequence \((i_1, i_2, \ldots, i_k)\) of indices of \(X\) such that for all \(j = 1, \ldots, k\), \(x_{i_j} = z_j\)
- i.e., as one reads through \(Z\), one can find a match to each symbol of \(X\) in order (though not necessarily contiguous)
- E.g., \(Z = \langle B, C, D, B \rangle\) is a subsequence of \(X = \langle A, B, C, B, D, A, B \rangle\) since \(z_1 = x_2, z_2 = x_4, z_3 = x_5\), and \(z_4 = x_7\)
- \(Z\) is a common subsequence of \(X\) and \(Y\) if it is a subsequence of both
- The goal of the longest common subsequence problem is to find a maximum-length common subsequence (LCS) of sequences \(X = \langle x_1, x_2, \ldots, x_m \rangle\) and \(Y = \langle y_1, y_2, \ldots, y_n \rangle\)

Characterizing the Structure of an Optimal Solution

- Given sequence \(X = \langle x_1, \ldots, x_m \rangle\), the \(i\)th prefix of \(X\) is \(X_i = \langle x_1, \ldots, x_i \rangle\)

    \[ Z = \langle z_1, \ldots, z_k \rangle, \]  
    \[ X_m = y_n \Rightarrow z_k = x_m = y_n \text{ and } Z_{k-1} \text{ is LCS of } X_{m-1} \text{ and } Y_{n-1} \]

- If \(z_k \neq x_m\), can lengthen \(Z\), ⇒ contradiction

    - If \(Z_{k-1}\) not LCS of \(X_{m-1}\) and \(Y_{n-1}\), then a longer CS of \(X_{m-1}\) and \(Y_{n-1}\) could have \(x_m\) appended to it. To get CS of \(X\) and \(Y\) that is longer than \(Z\), ⇒ contradiction

    - If \(x_m \neq y_n\), then \(Z_k \neq x_m\) implies that \(Z\) is an LCS of \(X_{m-1}\) and \(Y_{n-1}\)

    - If \(Z_k \neq x_m\), then \(Z\) is a CS of \(X_{m-1}\) and \(Y_{n-1}\). Any CS of \(X_{m-1}\) and \(Y_{n-1}\) that is longer than \(Z\) would also be a longer CS for \(X\) and \(Y\), ⇒ contradiction

- If \(x_m \neq y_n\), then \(z_k \neq y_n\) implies that \(Z\) is an LCS of \(X\) and \(Y_{n-1}\)

    - Similar argument to (2)

Recursively Defining the Value of an Optimal Solution

- The theorem implies the kinds of subproblems that we'll investigate
- to find LCS of \(X = \langle x_1, \ldots, x_m \rangle\) and \(Y = \langle y_1, \ldots, y_n \rangle\)

    - If \(x_m = y_n\), then find LCS of \(X_{m-1}\) and \(Y_{n-1}\) and append \(x_m = y_n\) to it
    - If \(x_m \neq y_n\), then find LCS of \(X_{m-1}\) and \(Y_{n-1}\) and find LCS of \(X_{m-1}\) and \(Y\) and identify the longest one

    - Let \(c[i, j]\) = length of LCS of \(X_i\) and \(Y_j\)

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

Computing the Value of an Optimal Solution

- What is the time complexity?
Computing the Value of an Optimal Solution (2)

\[ X = \langle A, B, C, B, D, A, B \rangle, \ Y = \langle B, D, C, A, B, A \rangle \]

Constructing an Optimal Solution from Computed Information

- Length of LCS is stored in \( c[m, n] \)
- To print LCS, start at \( b[m, n] \) and follow arrows until in row or column 0
- If in cell \((i, j)\) on this path, when \( x_i = y_j \) (i.e. when arrow is "\( \searrow \)"), print \( x_i \) as part of the LCS
- This will print LCS backwards

Algorithm 9: Print-LCS\((b, X, i, j)\)

```java
1 if \( i == 0 \) or \( j == 0 \) then
2 return
3 if \( b[i, j] == \searrow \) then
4 \( \text{print } x_i \), \( \text{PRINT-LCS}(b, X, i-1, j-1) \)
5 else if \( b[i, j] == \nearrow \) then
6 \( \text{print } x_i \), \( \text{PRINT-LCS}(b, X, i-1, j) \)
7 else \( \text{PRINT-LCS}(b, X, i, j-1) \)
```

What is the time complexity?

Optimal Binary Search Trees

- Goal is to construct binary search trees such that most frequently sought values are near the root, thus minimizing expected search time
- Given a sequence \( K = \langle k_1, \ldots, k_n \rangle \) of \( n \) distinct keys in sorted order
- Key \( k_i \) has probability \( p_i \), that it will be sought on a particular search
- To handle searches for values not in \( K \), have \( n + 1 \) dummy keys \( d_0, d_1, \ldots, d_n \) to serve as the tree’s leaves
- Dummy key \( d_i \) will be reached with probability \( q_i \)
- If depth\(_T\)(\( k_i \)) is distance from root of \( k_i \) in tree \( T \), then expected search cost of \( T \) is

\[
1 + \sum_{i=1}^{n} p_i \text{depth}_{T}(k_i) + \sum_{i=0}^{n} q_i \text{depth}_{T}(d_i)
\]

- An optimal binary search tree is one with minimum expected search cost

Optimal Binary Search Trees (2)
Characterizing the Structure of an Optimal Solution

- Observation: Since \( K \) is sorted and dummy keys interspersed in order, any subtree of a BST must contain keys in a contiguous range \( k_1, \ldots, k_j \) and have leaves \( d_1, \ldots, d_j \)
- Thus, if an optimal BST \( T \) has a subtree \( T' \) over keys \( k_1, \ldots, k_j \), then \( T' \) is optimal for the subproblem consisting of only the keys \( k_1, \ldots, k_j \)
- If \( T' \) weren’t optimal, then a lower-cost subtree could replace \( T' \) in \( T \), which contradicts the optimal BST.
- Given keys \( k_1, \ldots, k_j \), say that its optimal BST roots at \( k_i \) for some \( i \leq r \leq j \)
- Thus if we make right choice for \( k_i \) and optimally solve the problem for \( k_1, \ldots, k_{i-1} \) (with dummy keys \( d_{1, \ldots, d_{i-1}} \)) and the problem for \( k_{i+1}, \ldots, k_j \) (with dummy keys \( d_{i+1, \ldots, d_{j}} \), we’ll end up with an optimal solution
- Since we don’t know optimal \( k_i \), we’ll try them all

Recursively Defining the Value of an Optimal Solution

- Define \( e[i, j] \) as the expected cost of searching an optimal BST built on keys \( k_i, \ldots, k_j \)
- If \( j = i - 1 \), then there is only the dummy key \( d_i \), so
  \[ e[i, i-1] = q_i \]
- If \( j > i \), then choose root \( k_r \) from \( k_i, \ldots, k_j \) and optimally solve subproblems \( k_i, \ldots, k_{r-1} \) and \( k_{r+1}, \ldots, k_j \)
- When combining the optimal trees from subproblems and making them children of \( k_r \), we increase their depth by 1, which increases the cost of each by the sum of the probabilities of its nodes
- Define \( w(i, j) = \sum_{k \leq i} q_k + \sum_{p > j} q_p \) as the sum of probabilities of the nodes in the subtree built on \( k_1, \ldots, k_i \), and get
  \[ e[i, j] = p_i + e[i, r-1] + w(i, r-1) + e[r+1, j] + w(r+1, j) \]

Computing the Value of an Optimal Solution

```plaintext
Algorithm 10: Optimal-BST(\( p, q, n \))
```

What is the time complexity?