Chapter 7
Dynamic Programming

Dynamic Programming

- Fibonacci sequence:
  \[ f_1 = 1 \\ f_2 = 1 \\ f_3 = 2 \\ f_4 = 3 \\ f_5 = 5 \\ f_6 = 8 \\ f_7 = 13 \ldots \]
- \[ f(n) = \begin{cases} 1 & \text{if } (n=1) \text{ or } (n=2) \text{ then return 1;} \\ f(n-1) + f(n-2) & \text{else return } f(n-1) + f(n-2); \end{cases} \]
  This algorithm is far from being efficient, as there are many duplicate recursive calls to the procedure.

Dynamic Programming

- An algorithm that employs the dynamic programming technique is not necessarily recursive by itself, but the underlying solution of the problem is usually stated in the form of a recursive function.
- This technique resort to evaluating the recurrence in a bottom-up manner, storing intermediate results that are used later on to compute the desired solution.
- This technique applies to many combinatorial optimization problems to derive efficient algorithms.

The Longest Common Subsequence Problem

- Given two strings \( A \) and \( B \) of lengths \( n \) and \( m \), respectively, over an alphabet \( \sum \), determine the length of the longest subsequence that is common to both \( A \) and \( B \).

- A subsequence of \( A=a_1a_2\ldots a_n \) is a string of the form \( a_{i_1}a_{i_2}\ldots a_{i_k} \) where each \( i_j \) is between 1 and \( n \) and \( 1 \leq i_1 < i_2 < \ldots < i_k \leq n \).

The Longest Common Subsequence Problem

- Example: \( A=\text{"vehicle"}, \ B=\text{"vertices"} \)

  What is the longest common subsequence of \( A \) and \( B \)?
Designing a DP solution

How are the subproblems defined?
Where are the solutions stored?
How are the base values computed?
How do we compute each entry from other entries in the table?
What is the order in which we fill in the table?

The Longest Common Subsequence Problem

Suppose that both $i$ and $j$ are greater than 0. Then

- If $a_i = b_j$, $L[i, j] = L[i-1, j-1] + 1$
- If $a_i \neq b_j$, $L[i, j] = \max(L[i, j-1], L[i-1, j])$

We get the following recurrence for computing the length of the longest common subsequence of $A$ and $B$:

$$L[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1, j-1] + 1 & \text{if } i > 0, j > 0 \text{ and } a_i = b_j \\ \max(L[i, j-1], L[i-1, j]) & \text{if } i > 0, j > 0 \text{ and } a_i \neq b_j \\ \end{cases}$$

Recursive Version:

```c
int LCS(int i, int j) {
    if (i == 0 || j == 0) return 0;
    if (a[i] == b[j]) return LCS(i-1, j-1) + 1;
    return max(LCS(i, j-1), LCS(i-1, j));
}
```

Q: In what order $L[i, j]$ are filled?

The Longest Common Subsequence Problem

- Input: Two strings $A$ and $B$ of lengths $n$ and $m$, respectively, over an alphabet $\Sigma$.
- Output: The length of the longest common subsequence of $A$ and $B$.

Recursive Version:

```c
int LCS(int i, int j) {
    if (i == 0 || j == 0) return 0;
    if (a[i] == b[j]) return LCS(i-1, j-1) + 1;
    return max(LCS(i, j-1), LCS(i-1, j));
}
```

Q: In what order $L[i, j]$ are filled?

The Longest Common Subsequence Problem

- Let $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_m$.
- Let $L[i, j]$ denote the length of a longest common subsequence of $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$. When $i$ or $j$ be 0, it means the corresponding string is empty.
- Naturally, if $i = 0$ or $j = 0$, the $L[i, j] = 0$.

The Longest Common Subsequence Problem

- What’s the performance of this algorithm?
  - Time Complexity?
  - Space Complexity?

An optimal solution to the longest common subsequence problem can be found in $\Theta(nm)$ time and $\Theta(\min(m, n))$ space.

Dynamic Programming Paradigm

Subproblem Property: The problem can be recursively defined by the subproblem of the same kind.

Trade space for time: A table is used to store the solutions of the subproblems (the meaning of “programming” before the age of computers is “table”).

The All-Pairs Shortest Path Problem

- Example:

  **Weight:**
  
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>b</td>
<td>8</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

  **Distance:**
  
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>7</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Designing a DP solution

- How are the subproblems defined?
- Where are the solutions stored?
- How are the base values computed?
- How do we compute each entry from other entries in the table?
- What is the order in which we fill in the table?
Two DP algorithms for All-pairs shortest paths

Both are correct. Both produce correct values for all-pairs shortest paths.
The difference is the subproblem formulation, and hence in the running time.
Be prepared to provide one or both of these algorithms, and to be able to apply it to an input (on some exam, for example).

Dynamic Programming

First attempt: let \{1,2,...,n\} denote the set of vertices.
Subproblem formulation:
\[ M[i,j,k] = \text{min length of any path from } i \text{ to } j \text{ that uses at most } k \text{ edges.} \]
All paths have at most \( n-1 \) edges, so \( 1 \leq k \leq n-1 \).
When \( k=1 \), \( M[i,j,1] = w[i,j] \), the edge weight from \( i \) to \( j \).
Minimum paths from \( i \) to \( j \) are found in \( M[i,j,n-1] \)

Question: How to set \( M[i,j,k] \) from other entries?

How to set \( M[i,j,k] \) from other entries, for \( k>1 \)?
Consider a minimum weight path from \( i \) to \( j \) that has at most \( k \) edges.
1. Case 1: The minimum weight path has at most \( k-1 \) edges.
   \[ M[i,j,k] = M[i,j,k-1] \]
2. Case 2: The minimum weight path has exactly \( k \) edges.
   \[ M[i,j,k] = \min\{ M[i,x,k-1] + w(x,j) : x \in V \} \]
Combining the two cases:
\[ M[i,j,k] = \min\{\min\{M[i,x,k-1] + w(x,j) : x \in V\}, M[i,j,k-1]\} \]

Finishing the design

How are the subproblems defined?
Subproblem formulation:
\[ M[i,j,k] = \text{min length of any path from } i \text{ to } j \text{ that uses at most } k \text{ edges.} \]
Where is the answer stored?
- Minimum paths from \( i \) to \( j \) are found in \( M[i,j,n-1] \)

How are the base values computed?
- When \( k=1 \), \( M[i,j,1] = w[i,j] \), the edge weight from \( i \) to \( j \).

How do we compute each entry from other entries?
- \( M[i,j,k] = \min\{M[i,x,k-1] + w(x,j) : x \in V\}, M[i,j,k-1]\) \( M[i,j,k-1]\)

What is the order in which we fill in the matrix?
- For \( k \) from \( 1 \) to \( n-1 \), compute \( M[i,j,k] \)

Running time?

Pseudo-Code and Running time analysis

for \( j = 1 \) to \( n \) 
   for \( i = 1 \) to \( n \) 
      \[ M[i,j,1] = w[i,j]; \]
      for \( k = 2 \) to \( n-1 \) 
         for \( j = 1 \) to \( n \) 
            for \( i = 1 \) to \( n \) 
               \[ \text{minx} = M[i,j,k-1]; \]
               for \( x = 1 \) to \( n \) 
                  if (\( \text{minx} > M[i,x,k-1] + w(x,j) \)) \( \text{minx} = M[i,x,k-1] + w(x,j); \)
               \[ M[i,j,k] = \text{minx}; \]
            \]
      \]
   \]

Next DP approach

Try a new subproblem formulation!
\[ Q[i,j,k] = \text{minimum weight of any path from } i \text{ to } j \text{ that uses internal vertices drawn from } \{1,2,...,k\}. \]
Designing a DP solution

How are the subproblems defined?
- \( Q[i,j,k] = \) minimum weight of any path from \( i \) to \( j \) that uses internal vertices (other than \( i \) and \( j \)) drawn from \( \{1,2,\ldots,k\} \).

Where is the answer stored?
- \( Q[i,j,n] \) stores the min length from \( i \) to \( j \).

How are the base values computed?
- Base cases: \( Q[i,j,0] = w[i,j] \) for all \( i,j \).

How do we compute each entry from other entries?

What is the order in which we fill in the matrix?

Solving subproblems

\[ Q[i,j,k] = \text{minimum weight of any path from } i \text{ to } j \text{ that uses internal vertices drawn from } \{1,2,\ldots,k\}. \]

\( P \) is a minimum cost path from \( i \) to \( j \) that uses vertex \( k \), and has all internal vertices from \( \{1,2,\ldots,k\} \).

Path \( P_1 \) from \( i \) to \( k \), and \( P_2 \) from \( k \) to \( j \).

The weight of \( P_1 \) is \( Q[i,k,k-1] \) (why??).

The weight of \( P_2 \) is \( Q[k,j,k-1] \) (why??).

Thus the weight of \( P \) is \( Q[i,k,k-1] + Q[k,j,k-1] \).

New DP algorithm

\[
\begin{align*}
&\text{for } j = 1 \text{ to } n \\
&\quad \text{for } i = 1 \text{ to } n \\
&\quad \quad Q[i,j,0] = w[i,j] \\
&\quad \text{for } k = 1 \text{ to } n \\
&\quad \quad \text{for } j = 1 \text{ to } n \\
&\quad \quad \quad \text{for } i = 1 \text{ to } n \\
&\quad \quad \quad \quad \quad Q[i,j,k] = \min\{Q[i,j,k-1], Q[i,k,k-1] + Q[k,j,k-1]\}
\end{align*}
\]

Each entry only takes \( O(1) \) time to compute

There are \( O(n^3) \) entries

Hence, \( O(n^3) \) time.

Total space: \( O(n^3) \) (or \( O(n^2) \))

Reusing the space

\[
\begin{align*}
&\text{// Use } R[i,j] \text{ for } Q[i,j,0], Q[i,j,1], \ldots, Q[i,j,n]. \\
&\text{for } j = 1 \text{ to } n \\
&\quad \text{for } i = 1 \text{ to } n \\
&\quad \quad R[i,j] = w[i,j] \\
&\quad \text{for } k = 1 \text{ to } n \\
&\quad \quad \text{for } j = 1 \text{ to } n \\
&\quad \quad \quad \text{for } i = 1 \text{ to } n \\
&\quad \quad \quad \quad \quad R[i,j] = \min\{R[i,j], R[i,k] + R[k,j]\}
\end{align*}
\]

Claim: For any \( k \), min path of \( i \) to \( j \) is \( R[i,j] = Q[i,j,k] \).

How to check negative cycles

\[
\begin{align*}
&\text{// Use } R[i,j] \text{ for } Q[i,j,0], Q[i,j,1], \ldots, Q[i,j,n]. \\
&\text{for } j = 1 \text{ to } n \\
&\quad \text{for } i = 1 \text{ to } n \\
&\quad \quad R[i,j] = w[i,j] \\
&\quad \text{for } k = 1 \text{ to } n \\
&\quad \quad \text{for } j = 1 \text{ to } n \\
&\quad \quad \quad \text{for } i = 1 \text{ to } n \\
&\quad \quad \quad \quad \quad R[i,j] = \min\{R[i,j], R[i,k] + R[k,j]\} \\
&\quad \text{for } i = 1 \text{ to } n \\
&\quad \quad \text{if } (R[i,i] < 0) \text{ print ("There is a negative cycle").}
\end{align*}
\]
How to compute transitive closure

// Post: T is the transitive closure of R
for j = 1 to n
    for i = 1 to n
        T[i,j] = R[i,j];
for k= 1 to n
    for j = 1 to n
        for i = 1 to n
            T[i,j] = T[i,j] || T[i,k] && T[k,j];

The Knapsack Problem

- Let \( U = \{ u_1, u_2, \ldots, u_n \} \) be a set of \( n \) items to be packed in a knapsack of max weight \( C \). For \( 1 \leq j \leq n \), let \( s_j \) and \( v_j \) be the weight and value of the \( j \)th item, respectively, where \( C \) and \( s_j, v_j \), \( 1 \leq j \leq n \), are all positive integers.

- The objective is to fill the knapsack with some items for \( U \) whose total weight is at most \( C \) and their total value is maximum. Assume without loss of generality that the weight of each item does not exceed \( C \).

The Knapsack Problem

- More formally, given \( U \) of \( n \) items, we want to find a subset \( S \subseteq U = \{ (s_j, v_j) \} \), such that

\[
\sum_{i \in S} v_i
\]

is maximized subject to the constraint

\[
\sum_{i \in S} s_i \leq C
\]

- This version of the knapsack problem is sometimes referred to in the literature as the 0/1 knapsack problem. This is because each item cannot be broken into pieces.

Example of Knapsack Problem

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>7</td>
</tr>
</tbody>
</table>

\( W = 11 \)

Ex: \( \{ 3, 4 \} \) has value 40.

Designing a DP solution

- How are the subproblems defined?
- Where are the solutions stored?
- How are the base values computed?
- How do we compute each entry from other entries in the table?
- What is the order in which we fill in the table?
Dynamic Programming: A Recursive Solution

Def. \( \text{OPT}(i, w) = \text{max profit subset of items 1, \ldots, i with weight limit w.} \)

- Case 1: \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using weight limit \( w \)

- Case 2: \( \text{OPT} \) selects item \( i \).
  - new weight limit = \( w - w_i \)
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using this new weight limit

\( \text{OPT}(i, w) \) selects the max of the two cases:

\[
\text{OPT}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{OPT}(i-1, w) & \text{if } w_i > w \\
\max\{\text{OPT}(i-1, w), v_i + \text{OPT}(i-1, w-w_i)\} & \text{otherwise}
\end{cases}
\]

Input: \( n, W, w_1, \ldots, w_n, v_1, \ldots, v_n \)

\[
\text{for } w = 0 \text{ to } W \\
M[0, w] = 0 \\
\text{for } i = 1 \text{ to } n \\
\text{for } w = 1 \text{ to } W \\
\text{if } (w_i > w) \\
M[i, w] = M[i-1, w] \\
\text{else} \\
M[i, w] = \max\{M[i-1, w], v_i + M[i-1, w-w_i]\} \\
\text{return } M[n, W]
\]

Knapsack Problem: Bottom-Up

Knapsack.

\( \text{OPT}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{OPT}(i-1, w) & \text{if } w_i > w \\
\max\{\text{OPT}(i-1, w), v_i + \text{OPT}(i-1, w-w_i)\} & \text{otherwise}
\end{cases} \)

Matrix-Chain multiplication

We are given a sequence

\( \langle A_1, A_2, \ldots, A_n \rangle \)

And we wish to compute

\( A_1 \times A_2 \times \ldots \times A_n \)

Example: \( \langle A_4, A_2, A_3 \rangle \)

\( A_1 = 10 \times 100 \)

\( A_2 = 100 \times 5 \)

\( A_3 = 5 \times 50 \)

\[
\begin{array}{cccccccccccc}
\text{Dim} & \text{Value} & \text{Weight} \\
1 & 1 & 1 \\
2 & 6 & 2 \\
3 & 18 & 5 \\
4 & 22 & 6 \\
5 & 28 & 7 \\
\end{array}
\]

Matrix multiplication

\[
\text{MATRIX-MULTIPLY} (A, B)
\]

\[
\text{if columns } [A] \neq \text{rows } [B] \\
\text{then error "incompatible dimensions"}
\]

\[
\text{else for } i \leftarrow 1 \text{ to rows } [A] \\
\text{for } j \leftarrow 1 \text{ to columns } [B] \\
\text{C[i, j] = 0} \\
\text{for } k \leftarrow 1 \text{ to columns } [A] \\
\text{C[i, j] += C[i, k]*A[i, k]*B[k, j]} \\
\text{return C}
\]

Time: \( O(m*n*p) \) if \( A \) is \( m \times n \) and \( B \) is \( n \times p \).
Matrix-Chain multiplication

Cost of the matrix multiplication:

An example:

\[
(\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3)
\]

\[
\begin{align*}
\mathbf{A}_1 & : 10 \times 100 \\
\mathbf{A}_2 & : 100 \times 5 \\
\mathbf{A}_3 & : 5 \times 50 \\
\end{align*}
\]

Dimensions: \( p = (10, 100, 5, 50) \).

Matrix-Chain multiplication

The problem:
Given a chain \( A_1, A_2, \ldots, A_n \) of \( n \) matrices, with \( p_i, p_i, \ldots, p_n \) where matrix \( A_i \) has dimension \( p_i \times p_i \), fully paranthesize the product \( A_1A_2\ldots A_n \) in a way that minimizes the number of scalar multiplications.

Overlapping subproblems:

Designing a DP solution

How are the subproblems defined?
Where are the solutions stored?
How are the base values computed?
How do we compute each entry from other entries in the table?
What is the order in which we fill in the table?

Matrix-Chain multiplication (cont.)

If we multiply \( (\mathbf{A}_1\mathbf{A}_2)\mathbf{A}_3 \) we perform \( 10 \times 100 \times 5 = 5000 \) scalar multiplications to compute the \( 10 \times 5 \) matrix product \( \mathbf{A}_1\mathbf{A}_2 \), plus another \( 10 \times 5 \times 50 = 2500 \) scalar multiplications to multiply this matrix by \( \mathbf{A}_3 \), for a total of \( 7500 \) scalar multiplications.

If we multiply \( \mathbf{A}_1(\mathbf{A}_2\mathbf{A}_3) \) we perform \( 100 \times 5 \times 50 = 25000 \) scalar multiplications to compute the \( 100 \times 50 \) matrix product \( \mathbf{A}_2\mathbf{A}_3 \), plus another \( 100 \times 50 \times 50 = 50000 \) scalar multiplications to multiply \( \mathbf{A}_1 \) by this matrix, for a total of \( 75000 \) scalar multiplications.
How are the subproblems defined?

Let \( A_{i, j} \) where \( i \leq j \), denote the matrix product
\[ A_i A_{i+1} \ldots A_j. \]

Subproblems: Let \( m[i, j] \) be the minimum number of scalar multiplications needed to compute the matrix \( A_i \ldots A_j \) where \( 1 \leq i \leq j \leq n \).

Any parenthesization of \( A_i A_{i+1} \ldots A_j \) must split the product between \( A_k \) and \( A_{k+1} \) for \( i \leq k < j \).

So \( A_i \ldots A_n \) is the final product, which can be split into \( A_i \ldots A_k \) and \( A_{k+1} \ldots A_n \).

Designing a DP solution

How are the subproblems defined?
- Let \( m[i, j] \) be the minimum number of scalar multiplications needed to compute the matrix \( A_i \ldots A_j \), where \( 1 \leq i \leq j \leq n \).

Where are the solutions stored?
The minimum cost is stored in \( m[1,n] \).

How are the base values computed?
The minimum cost of \( m[i,i] \) are 0 for any \( 1 \leq i \leq n \).

How do we compute each entry from other entries in the table?

What is the order in which we fill in the table?

Matrix-Chain multiplication (cont.)

To help us keep track of how to construct an optimal solution we define \( s[i, j] \) to be a value of \( k \) at which we can split the product \( A_i \ldots A_j \) to obtain an optimal parenthesization.

That is \( s[i, j] \) equals a value \( k \) such that
\[
m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j,
\]
\[
s[i, j] = k
\]

Matrix-Chain multiplication

\[
\text{MATRIX-CHAIN-ORDER}(p) \{
\text{n} \leftarrow \text{length}[p]-1
\text{for } i \leftarrow 1 \text{ to } n
\text{m}[i,i] \leftarrow 0
\text{for } l \leftarrow 1 \text{ to } n - 1
\text{for } i \leftarrow 1 \text{ to } n - l
\text{j} \leftarrow i + l
\text{m}[i,j] \leftarrow \infty
\text{for } k \leftarrow i \text{ to } j-1
\text{q} \leftarrow m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
\text{if} \ (q < m[i,j])
\text{m}[i,j] \leftarrow q
\text{s}[i,j] \leftarrow k
\}
// results are in m[][] and s[][].
\}
\]

Running time: \( O(n^3) \)

Matrix-Chain multiplication

\[
\text{PRINT-OPTIMAL-PARENS}(s, i, j)
\text{if } i=j
\text{then print \"A\"
\text{else print \" ( \"
\text{PRINT-OPTIMAL-PARENS}(s, i, s[i,j])
\text{PRINT-OPTIMAL-PARENS}(s, s[i,j]+1, j)
\text{Print \" ) \"
\}
\]
Elements of dynamic programming

When should we apply the method of Dynamic Programming?

Two key ingredients:

- **Optimal substructure**: an optimal solution to the problem uses optimal solutions to subproblems.

- **Overlapping subproblems**: a recursive algorithm revisits the same subproblem over and over again.

How many scalar multiplications using the recursive definition of \( m[i,j] \)?

\[
m[i,j] = \begin{cases} 
0 & \text{if } i = j \\
\min_{i < k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j.
\end{cases}
\]

Let \( T[j-i-1] \) denote the number of scalar multiplications using the recursive definition of \( m[i,j] \).

\[
\begin{align*}
T(i) &= 0, \\
T(n) &= \sum_{i=1}^{n-1} (T(k)+T(n-k)+2) \quad \text{for } n > 1 \\
&= 2\sum_{i=1}^{n-1} T(i) + 2(n-1)
\end{align*}
\]

We guess that \( T(n) = \Omega(2^n) \).

I.e., using \( T(n) \geq 2^n \).