Chapter 7
Dynamic Programming

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 resorts to evaluating the recurrence in a bottom-up manner, saving intermediate results that are used later on to compute the desired solution.

This technique applies to many combinatorial optimization problems to derive efficient algorithms.

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").

Dynamic Programming

Fibonacci sequence:

\[ f_1 = 1 \]
\[ f_2 = 1 \]
\[ f_3 = 2 \]
\[ f_4 = 3 \]
\[ f_5 = 5 \]
\[ f_6 = 8 \]
\[ f_7 = 13 \]
\[ \vdots \]

1. if \((n=1)\) or \((n=2)\) then return 1;
2. else return \(f(n-1)+f(n-2)\);

This algorithm is far from being efficient, as there are many duplicate recursive calls to the procedure.

Dynamic Programming

\[ f(n) \]
1. \( f(n) \)
2. \( \text{if } (n=1) \) or \( (n=2) \) then return 1;
3. else return \( f(n-1) + f(n-2) \);

This algorithm is far from being efficient, as there are many duplicate recursive calls to the procedure.

The Longest Common Subsequence Problem

Given two strings \(A\) and \(B\) of lengths \(n\) and \(m\), respectively, over an alphabet \(\Sigma\), 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

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 \( L[i, j] = 0 \)

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:

```java
int LCS(n, m) {
    if (L[i, j] == -1) {
        if (i == 0 || j == 0)
            L[i, j] = 0;
        else if (a[i] == b[j])
            L[i, j] = LCS(i-1, j-1) + 1;
        else
            L[i, j] = max(LCS(i, j-1), LCS(i-1, j));
    }
    return L[i, 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.

1. for \( i \) \( \leftarrow \) 0 to n
2. \( L[0, j] \leftarrow 0; \)
3. for \( j \) \( \leftarrow \) 0 to m
4. \( L[i, 0] \leftarrow 0; \)
5. for \( i \) \( \leftarrow \) 1 to n
6. for \( j \) \( \leftarrow \) 1 to m
7. if \( (a_i = b_j) \) \( L[i, j] \leftarrow L[i-1, j-1] + 1; \)
8. else \( L[i, j] \leftarrow \max(L[i, j-1], L[i-1, j]); \)
9. return \( L[n, m] \);

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.

The All-Pairs Shortest Path Problem

- Let \( G=(V, E) \) be a directed graph in which each edge \( (i, j) \) has a non-negative length \( w(i, j) \). If there is no edge from vertex \( i \) to vertex \( j \), then \( w(i, j) = \infty \).

The problem is to find the distance from each vertex to all other vertices, where the distance from vertex \( x \) to vertex \( y \) is the length of a shortest path from \( x \) to \( y \).

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?

Next DP approach

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

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 \(n\)  
\[ M[i,j,k] = \text{min} \{\text{min} \{ M[i,x,k-1] + w(x,j) : x \in V\}, M[i,j,k-1]\} \]

How many entries do we need to compute? \(O(n^3)\)
\(1 \leq i \leq n; 1 \leq j \leq n; 1 \leq k \leq n-1\)

How much time does it take to compute each entry? \(O(n)\)
Total time: \(O(n^4)\)

Finishing the design

Where is the answer stored?
How are the base values computed?
How do we compute each entry from other entries?
What is the order in which we fill in the matrix?
Running time?

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.
- Case 1: The minimum weight path has at most \(k-1\) edges.
  - \(M[i,j,k] = M[i,j,k-1]\)
- Case 2: The minimum weight path has exactly \(k\) edges.
  - \(M[i,j,k] = \text{min} \{ M[i,x,k-1] + w(x,j) : x \in V\}\)
Combining the two cases:
\[ M[i,j,k] = \text{min} \{\text{min} \{ M[i,x,k-1] + w(x,j) : x \in V\}, M[i,j,k-1]\} \]
Designing a DP solution

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

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,...,k}.

Base cases: Q[i,j,0] = w[i,j] for all i,j
Minimum paths from i to j are found in Q[i,j,n]
Once again, O(n^3) entries in the matrix

Solving subproblems

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

Such minimum cost path either includes vertex k or does not include vertex k.

If the minimum cost path P includes vertex k, then you can divide P into the path P1 from i to k, and P2 from k to j.
What is the weight of P1?
What is the weight of P2?

P is a minimum cost path from i to j that uses vertex k, and has all internal vertices from {1,2,...,k}.
Path P1 from i to k, and P2 from k to j.
The weight of P1 is Q[i,k,k-1] (why??).
The weight of P2 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

for j = 1 to n
  for i = 1 to n
    Q[i,j,0] = w[i,j]
  for k = 1 to n
    for j = 1 to n
      for i = 1 to n
        Q[i,j,k] = min{Q[i,j,k-1], Q[i,k,k-1] + Q[k,j,k-1]}

Each entry only takes O(1) time to compute.
There are O(n^3) entries
Hence, O(n^3) time.

Reusing the space

// Use R[i,j] for Q[i,j,0], Q[i,j,1], ..., Q[i,j,n]
for j = 1 to n
  for i = 1 to n
    R[i,j] = w[i,j]
for k = 1 to n
  for j = 1 to n
    for i = 1 to n
      R[i,j] = min(R[i,j], R[i,k] + R[k,j])
How to check negative cycles

```java
// Use R[i][j] for Q[i,j,0], Q[i,j,1], ... , Q[i,j,n]
for j = 1 to n
    for i = 1 to n
        R[i][j] = w[i][j];
    for k= 1 to n
        for j = 1 to n
            for i = 1 to n
                R[i][j] = min{R[i][j], R[i][k] + R[k][j]};
    for i = 1 to n
        if (R[i][i] < 0) print("There is a negative cycle");
```

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 $C$.

Example of Knapsack Problem

Knapsack problem.
- Given $n$ objects and a "knapsack." 
- Item $i$ weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of $W$ kilograms.
- Goal: fill knapsack so as to maximize total value.

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

<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>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td>7</td>
</tr>
</tbody>
</table>

W = 11

The Knapsack Problem

Let $OPT(i, j)$ denote the value obtained by filling a knapsack of size $j$ with items taken from the first $i$ items $\{ u_1, u_2, \ldots, u_i \}$ in an optimal way. Here the range of $i$ is from 0 to $n$ and the range of $j$ is from 0 to $C$. Thus, what we seek is the value $OPT(n, C)$.

Obviously, $OPT(0, j) = 0$ for all values of $j$, as there is nothing in the knapsack. On the other hand, $OPT(i, 0)$ is 0 for all values of $i$ since nothing can be put in a knapsack of size 0.
Dynamic Programming: A Recursive Solution

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

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

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

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

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

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

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

Input: n, W, w1,…,wN, v1,…,vN

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

Knapsack Algorithm Example

<table>
<thead>
<tr>
<th></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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n+1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w+1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n+1</td>
<td>{1}</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w+1</td>
<td>{1, 2}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n+1</td>
<td>{1, 2, 3}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>19</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w+1</td>
<td>{1, 2, 3, 4}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>18</td>
<td>22</td>
<td>24</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n+1</td>
<td>{1, 2, 3, 4, 5}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>22</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( OPT\{4, 3\} \)

value = 22 + 18 = 40

W = 11