CSIEB0120
Lecture 03
Dynamic Programming

Shiow-yang Wu & Wei-Che Chien
(吳秀陽 & 簡暐哲)
Department of Computer Science and Information Engineering
National Dong Hwa University

Objectives

- Describe the dynamic programming technique
- Contrast the divide-and-conquer and dynamic programming approaches
- Identify when dynamic programming should be used to solve a problem
- Define the Principle of Optimality
- Apply the Principle of Optimality to solve optimization problems

Problems with Divide-and-Conquer

- D&C is a top-down approach
- Blindly divide problem into smaller instances and solve the smaller instances
- Technique works efficiently for problems where smaller instances are unrelated
- Inefficient solution to problems where smaller instances are related (why?)
- Example: Recursive solution to the Fibonacci sequence

Dynamic Programming

- Divide an instance of a problem into one or more smaller instances, like DAC
  - **Solve** small instances first.
  - **Store** the results.
  - **Reuse** the stored results, instead of re-computing.
- Bottom-up approach, unlike DAC.
  - **Establish** a recursive property that gives the solution to an instance of the problem.
  - **Solve** an instance of a problem in a bottom-up fashion by solving smaller instances first.

The Binomial Coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \text{ for } 0 \le k \le n$$

- Pascal's Triangle

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

Binomial Coefficient using D&C

- **Problem:** Compute the binomial coefficient $\binom{n}{k}$
- **Inputs:** nonnegative integers *n* and *k*, where k $\le$ *n*.
- **Outputs:** *bin*, the binary coefficient of *n* and *k*.

```
void bin(int n, int k) {
  if (k==0 || n==k)
    return 1;
  else
    return bin(n-1, k-1) + bin(n-1, k);
}
```

Analysis of Recursive BC

- **Basic operation:** the number of terms to compute.
- **Input size:** *n* and *k*.
- It can be proof by induction that:

$$T(n,k) = 2\binom{n}{k} - 1$$

Very inefficient!!

What's Wrong with the Recursive BC?

- Small instances are solved repeatedly in each recursive call.
- Eg., $bin(n-1, k-1)$ and $bin(n-1, k)$ both need $bin(n-2, k-1)$ which is solved repeatedly.
- Remember that D&C approach is inefficient when an instance is divided into smaller instances almost as large as the original instance.
- This is a good example of problem that should be solved with dynamic programming instead.

Dynamic Programming for BC

- Using the recursive property, construct an array B to store solutions to smaller instances.

$$B[i,j] = \binom{i}{j}$$

- Solve problem in a bottom-up fashion.
- Reuse stored solutions when ever needed.
- Each smaller instance needs to be computed only once.

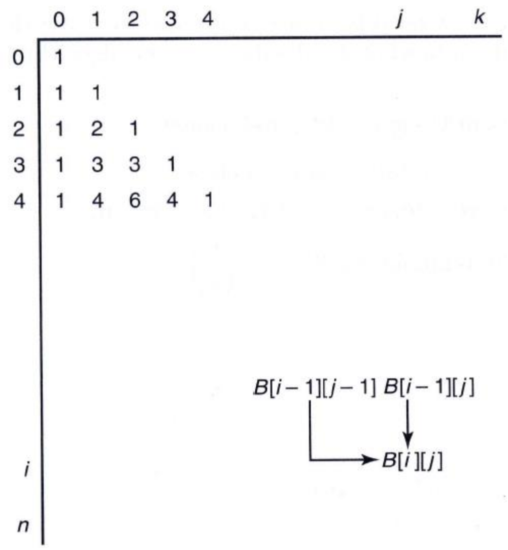
Dynamic Programming for BC

- Establish a recursive property s.t. larger instance is solved by smaller(usually a little) instances.

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & \text{if } 0 < j < i \\ 1 & \text{if } j = 0 \text{ or } j = i \end{cases}$$

- Solve an instance in a bottom-up fashion
  - Solve, store and keep going until we get to the point by reusing the stored results. (See Fig. 3.1)
  - Compute rows in B in sequence starting with row 1
  - At each iteration, the values needed for that iteration have already been computed.

Figure 3.1 The array B for BC



Algorithm 3.2 BC with Dyna Prog

- **Problem:** Compute the binomial coefficient
- **Inputs:** nonnegative int n and k, where $k \le n$.
- **Outputs:** the binary coefficient of n and k.

```
void bin2(int n, int k) {
  index i, j;
  int B[0..n][0..k];
  for (i = 0; i <= n; i++)
    for (j = 0; j <= min(i, k); j++)
      if (j == 0 || j == i) B[i][j] = 1;
      else B[i][j] = B[i-1][j-1] + B[i-1][j];
  return B[n][k];
}
```

Time Complexity of Algorithm 3.2

- **Basic operation:** # terms to compute.
- **Input size:** *n* and *k*.

i	0	1	2	3	$\cdots$	k	$k+1$	$\cdots$	n
Number of passes	1	2	3	4	$\cdots$	$k+1$	$k+1$	$\cdots$	$k+1$

$$1+2+3+\cdots+k+\overbrace{(k+1)+\cdots+(k+1)}^{n-k+1 \text{ times}}=\frac{k(k+1)}{2}+(n-k+1)(k+1)$$

$$=\frac{(2n-k+2)(k+1)}{2}\in\Theta(nk)$$

Very good!!

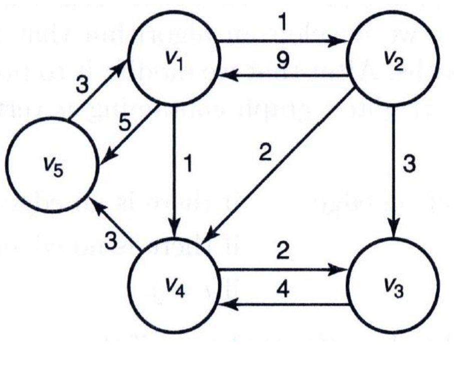
Summary of the DP Approach

- Dyn Prog is similar to D&C in recursively divides an instance into smaller instances.
- Key difference is to iteratively solve it, starting with smallest instance and bottom-up.
- Instead of blindly recursion, we compute and store solution of smaller instance just once.
- For larger instances, we reuse the stored solutions of smaller instances.
- In BC, once a row is computed, we no longer need rows that precedes it. Therefore a 1D array [0..k] is good enough. (why and how?)

Graph Revisited

- Graph consists of two elements: G = (V, E).
- E is a set of edges. Every edge has two endpoints in V.
- If edges in E can be defined as a set of ordered pairs, G is a directed graph or digraph in short.
- If edges have values associated with them, the values are called weights and G is a weighted graph.
- In a digraph, a path is a sequence of vertices such that there is an edge from each vertex to its successor.
- A path from a vertex to itself is called a cycle.
- If G contains a cycle, G is cyclic; otherwise, it is acyclic.
- A path is simple, if it never passes through the same vertex twice.
- A length of a path in a weighted graph is the sum of the weights on the path.

Example: A weighted, directed graph



- What is the length of the path v5→v1→v2→v3 ?
- Is this a cyclic or acyclic graph?
- If it cyclic, where is the cycle?

Shortest Path Problem

- A problem that has many applications is finding the shortest paths among vertices.
- A shortest path must be a simple path. (why?)
- How many simple paths from v1 to v3 ?
- There are three: [*v*1, *v*2, *v*3], [*v*1, *v*4, *v*3], and [*v*1, *v*2, *v*4, *v*3].
- Which on is the shortest?
- When traveling among cities, shortest paths help us in finding the shortest routes between cities.
- Can you come up with another application?

Optimization Problem

- Shortest path problem is an optimization problem.
- Usually have multiple candidate solutions.
- Each candidate solution has a value (length, cost, …) associated with it.
- Solution to the instance is a candidate solution with an optimal value.
- Depending on the problem, the optimal value could be minimum or maximum.
- Shortest path is to find the path(s) with minimum length.

Shotest Paths Problem (SP)

- **Problem:** Compute the shortest paths from each vertex in a weighted graph to each of the other vertices.
- **Inputs:** A weight digraph and *n,* the number of vertices. W[*i*][*j*] is the weight on the edge from the *i*-th vertex to the *j*-th vertex.
- **Outputs:** A two dimensional array D, which has both its rows and columns indexed from 1 to n, where **D**[*i*][*j*] is the length of a shortest path from the *i*-th vertex to the *j*-th vertex.
- Clearly an optimization problem.

Brute-force Algorithm for SP

- **Strategy**: Find all possible paths, compute their lengths, and select the minimal one.
- **Analysis**
  - Suppose there are *n* vertices in the graph.
  - The total number of paths from v_i to v_j is (n-2)!. (why?)
  - This is much worse than exponential.
- Our goal is to find a more efficient algorithm.
  - Let's apply DP strategy instead.
  - Robert Floyd: DP algorithm for SP in 1962.
  - Same as Bernard Roy's(1959) and Stephen Warshall's(1962) for finding a transitive closure.
  - Floyd-Warshall algorithm.

DP Strategy for SP

- Adjacency matrix representation of graph

$$W[i][j] = \begin{cases} weight & \text{If there is an edge from } v_i \text{ to } v_j \\ \infty & \text{If there is no edge from } v_i \text{ to } v_j \\ 0 & \text{If } i = j. \end{cases}$$

- Distance matrix for the recursive property

  $$D^{(k)}[i][j] = \{v_1, v_2, \ldots, v_k\}$$

  is the length of a shortest path from v_i to v_j using only vertices in the set $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices.

Key Ideas of the DP Strategy

- There are n vertices in the graph.
- Create a sequence of n+1 arrays D^k where $0 \leq k \leq n$.
- $D^k[i, j]$ = length of a shortest path from v_i to v_j using only vertices in the set $\{ v_1 , v_2 , . . . v_k \}$
- $D^n[i, j]$ = length of shortest path from v_i to v_j using all vertices in the graph.
- $D^0 = W$ and $D^n = D$.

Dynamic Programming Steps

- **Establish** a **recursive property** to compute D^k from $D^{(k-1)}$.
- Solve instances of the problem in **bottom-up** fashion by repeating the process for k=1 to n.
- The initial conditions (smallest instances) are usually trivial. The solution(s) can be determined directly.
- Because of the bottom-up fashion, whenever we want to compute D^k, the value of $D^{(k-1)}$ should already be available.

DP Design for Shortest Paths

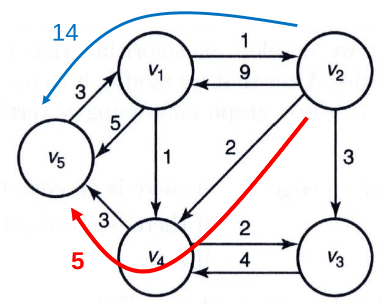
- **Establish** a recursive property: two cases

$$D^{(k)}[i][j] = minimum(\underbrace{D^{(k-1)}[i][j]}_{Case1}, \underbrace{D^{(k-1)}[i][k] + D^{(k-1)}[k][j]}_{Case2})$$

- **Case 1**: At least one shortest path from v_i to v_j, using only vertices in $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices, **does not** use v_k. Then $D^{(k)}[i][j] = D^{(k-1)}[i][j]$. (trivial, why?)
  - (e.g.) $D^{(5)}[1][3] = D^{(4)}[1][3] = 3$
- **Case 2**: All shortest paths from v_i to v_j, using only vertices in $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices, **do** use v_k. (example in next slide)

Examples of Shortest Paths Cases

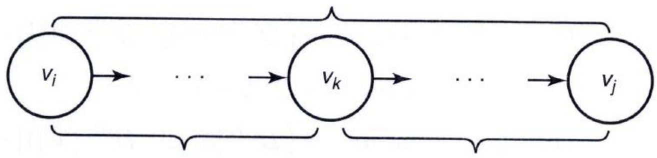
- Given the graph, what is the shortest path from $\mathbf{v_2}$ to $\mathbf{v_5}$ using only $\{v_1, v_2\}$?
- How about using only $\{v_1, v_2, v_3\}$? (case 1)
- How about using only $\{v_1, v_2, v_3, v_4\}$? (case 2)
- For case 2, the shortest path does go through v_4.
- Both sub-paths v_2~v_4 and v_4~v_5 must be shortest.



DP Design for SP: Case 2 Illustrated

- Case 2 represents the shortest path that go through v_k.

A shortest path from v_i to v_j using only vertices in $\{v_1, v_2, \ldots, v_k\}$

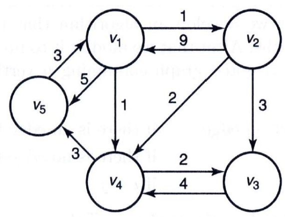


A shortest path from v_i to v_k using only vertices in $\{v_1, v_2, \ldots, v_{k-1}\}$

A shortest path from v_k to v_j using only vertices in $\{v_1, v_2, \ldots, v_{k-1}\}$

$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$$

Figure 3.3 Compute D from W



Given a weighted, directed graph G

Compute the shortest path matrix D from W.

W

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

D

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

Floyd's Algorithm I

```
void floyd(int n, const number W[][],
           number D[][]) {
  int i, j, k;
  D = W;
  for(k=1; k <= n; k++)
    for(i=1; i <= n; i++)
      for(j=1; j <= n; j++)
        D[i][j] = min(D[i][j],
                      D[i][k]+D[k][j]);
}
```

Every-Case Time Complexity of `floyd`

- It should be obvious that the nested loop is always executed the same number of times with a given n.
- Therefore the every-case time complexity is:

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3)$$

Floyd's Algorithm II

- **Problem**: Same as in Floyd's algorithm I, except shortest paths are also created.
- **Additional outputs**: an array P, which has both its rows and columns indexed from 1 to n, where

$$P[i][j] = \begin{cases} \text{Highest index of an intermediate vertex on the shortest path from } v_i \text{ to } v_j \text{, if at least one intermediate vertex exists.} \\ \text{0, if no intermediate vertex exists.} \end{cases}$$

Floyd's Algorithm II

```
void floyd2(int n, const number W[][],
            number D[][], index P[][]) {
  index i, j, k;
  for(i=1; i<=n; i++)
    for(j=1; j<=n; j++)
      P[i][j] = 0;
  D = W;
  for(k=1; k<=n; k++)
    for(i=1; i<=n; i++)
      for(j=1; j<=n; j++)
        if (D[i][k]+D[k][j] < D[i][j]) {
          P[i][j] = k;
          D[i][j] = D[i][k] + D[k][j];
        }
}
```


Figure 3.5 The array P by `floyd2`

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

Print Shortest Path

```
void path(index q, r) {
  if (P[q][r] != 0) {
    path(q, P[q][r]);
    cout << " v" << P[q][r];
    path(P[q][r], r);
  }
}
```

v_q .... $v_{P[q][r]}$ .... v_r

path(q, P[q][r])   path(P[q][r], r)

Example of Shortest Path Printing

- Using P, solve path(5, 3)

```
path(5,3) = 4
  path(5,4) = 1
    path(5,1) = 0
    v1
    path(1,4) = 0
  v4
  path(4,3) = 0
```

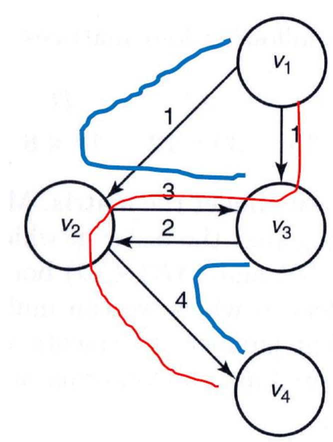
- **Result**: v1 v4. (The shortest path from v5 to v3 is v5, v1, v4, v3.)

The Principle of Optimality

- The principle of optimality is said to apply if an optimal solution to an instance of a problem always contains optimal solutions to **all** substances.
  - Although it may seem that any optimization problem can be solved using dynamic programming, this is not the case.
  - The principle of optimality must apply in the problem.
  - Apply for **shortest path** problem: If v_k is a node on an optimal path from v_i to v_j then the sub-paths v_i to v_k and v_k to v_j are also optimal paths.
- **Longest Paths** problem is to find the longest simple paths from each vertex to all other vertices.
  - Can we solve the problem using dynamic programming?
  - Why or why not?

An Example that the Principle of Optimality does NOT apply

- The principle of optimality does NOT apply for the **longest path problem**.
- The sub-paths of the longest (simple) path from v_1 to v_4 may **not** be the longest sub-paths.
- Can't solve with dynamic programming.



Chained Matrix Multiplication

- In general, to multiply an $i \times j$ matrix times a $j \times k$ matrix using the standard method, it is necessary to do $i \times j \times k$ elementary multiplications.
- (e.g.) $A_1 \times A_2 \times A_3$.
  - Suppose A_1 is 10×100, A_2 is 100×5, and A_3 is 5×50.
  - $(A1 \times A2) \times A3$

    $$10 \times 100 \times 5 + 10 \times 5 \times 50 = 7{,}500$$
  - $A1 \times (A2 \times A3)$

    $$100 \times 5 \times 50 + 10 \times 100 \times 50 = 75{,}000$$
- Different order can be considerably different !!

Example of CMM Problem

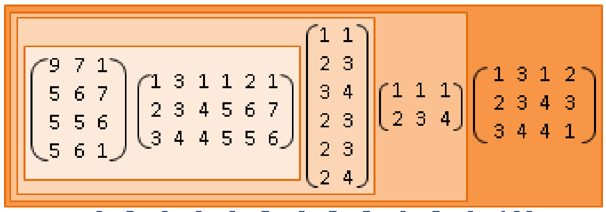
- Given a chain of matrices to multiply.

$$\underset{4\times3}{\begin{pmatrix} 9 & 7 & 1 \\ 5 & 6 & 7 \\ 5 & 5 & 6 \\ 5 & 6 & 1 \end{pmatrix}} \underset{3\times6}{\begin{pmatrix} 1 & 3 & 1 & 1 & 2 & 1 \\ 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 4 & 4 & 5 & 5 & 6 \end{pmatrix}} \underset{2\times6}{\begin{pmatrix} 1 & 1 \\ 2 & 3 \\ 3 & 4 \\ 2 & 3 \\ 2 & 3 \\ 2 & 4 \end{pmatrix}} \underset{2\times3}{\begin{pmatrix} 1 & 1 & 1 \\ 2 & 3 & 4 \end{pmatrix}} \underset{3\times4}{\begin{pmatrix} 1 & 3 & 1 & 2 \\ 2 & 3 & 4 & 3 \\ 3 & 4 & 4 & 1 \end{pmatrix}} = \underset{4\times4}{\begin{pmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{pmatrix}}$$

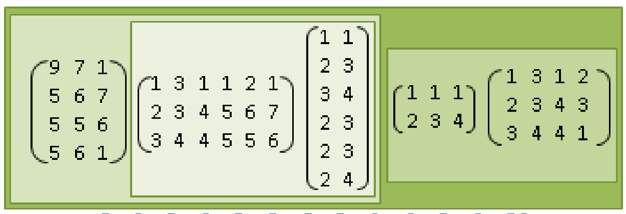
- How many elementary multiplications do we need?

Example of CMM Problem

- The # elementary multiplications varies with different order of matrix multiplication.



4 x 3 x 6 + 4 x 6 x 2 + 4 x 2 x 3 + 4 x 3 x 4 = 144



3 x 6 x 2 + 4 x 3 x 2 + 2 x 3 x 4 + 4 x 2 x 4 = 84

Chained Matrix Multiplication(CMM)

- **Brute-force** algorithm: Consider all possible orders and take the minimum.
- Let t_n be the number of different orders in which we can multiply n matrices: $A_1, A_2, \ldots, A_n$.
- $(A_1 \ldots A_{n-1})\ A_n$ will have t_{n-1} different orders.
- $A_1\ (A_2 \ldots A_n)$ will have t_{n-1} different orders.
- In other words, $t_n \geq t_{n-1} + t_{n-1} = 2\ t_{n-1}$ and $t_2 = 1$.
- Therefore, $t_n \geq 2t_{n-1} \geq 2^2 t_{n-2} \geq \ldots \geq 2^{n-2} t_2 = 2^{n-2} = \Theta(2^n)$.

Dynamic Programming for CMM

- Want to find the optimal order for chained-matrix multiplication which dependents on array dimensions.
- Brute-force algorithm is exponential.
- Principle of Optimality applies.
- We can develop a Dynamic Programming Solution.

Dynamic Programming for CMM

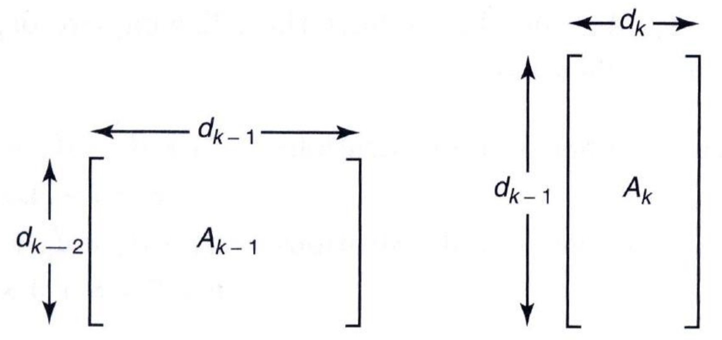
- Let d_k be the number of columns in A_k , $1 \le k \le n$.
- Let d_0 be the number of rows in A_1.
- In other words, $A_1\ A_2 \ldots A_n$ will be represented as $d_0 \times d_1 \times \ldots \times d_n$.
- Suppose $1 \le i \le j \le n$.
- $M[i][j]$ = minimum number of multiplications needed to multiply A_i through A_j, if $i < j$.

  $$MIN_{i \le k \le j-1}(M[i][k] + M[k+1][j] + d_{i-1}d_k d_j)$$

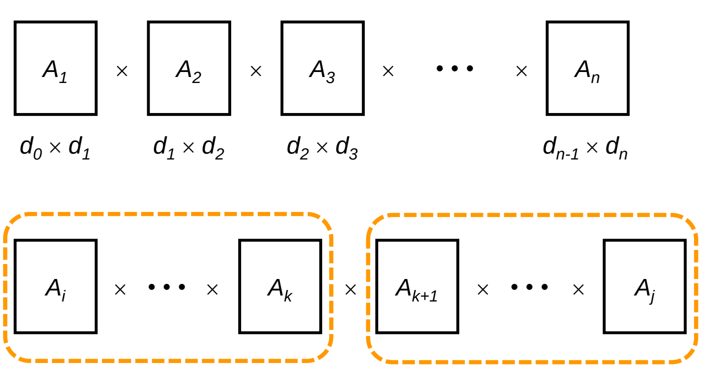
- $M[i][i] = 0$.

DP for CMM Illustrated

- The number of columns in A_{k-1} is the same as the number of rows in A_k.



DP for CMM Illustrated



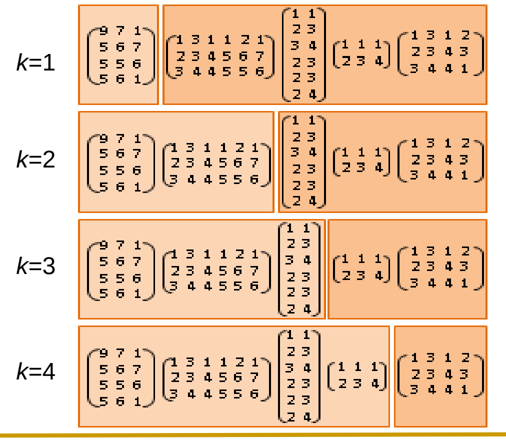
K is the last dividing point.

DP for CMM Illustrated

- For the previous example

$$\underset{4\times3}{\begin{pmatrix}9&7&1\\5&6&7\\5&5&6\\5&6&1\end{pmatrix}}\underset{3\times6}{\begin{pmatrix}1&3&1&1&2&1\\2&3&4&5&6&7\\3&4&4&5&5&6\end{pmatrix}}\underset{2\times6}{\begin{pmatrix}1&1\\2&3\\3&4\\2&3\\2&3\\2&4\end{pmatrix}}\underset{2\times3}{\begin{pmatrix}1&1&1\\2&3&4\end{pmatrix}}\underset{3\times4}{\begin{pmatrix}1&3&1&2\\2&3&4&3\\3&4&4&1\end{pmatrix}}=\underset{4\times4}{\begin{pmatrix}?&?&?&?\\?&?&?&?\\?&?&?&?\\?&?&?&?\end{pmatrix}}$$

- The last dividing point k for $A_1 \times A_2 \times \ldots \times A_5$ can be k=1, k=2, …, k=4



Example of Solving M

- Let

A_1	A_2	A_3	A_4	A_5	A_6
5×2	2×3	3×4	4×6	6×7	7×8

$$\begin{aligned}M[4][6] &= minimum_{4\le k\le5}(M[4][4]+M[5][6]+4\times6\times8, M[4][5]+M[6][6]+4\times7\times8)\\ &= minimum(0+6\times7\times8+4\times6\times8, 4\times6\times7+0+4\times7\times8)\\ &= minimum(528, 392) = 392\end{aligned}$$

$M[i][j]$	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

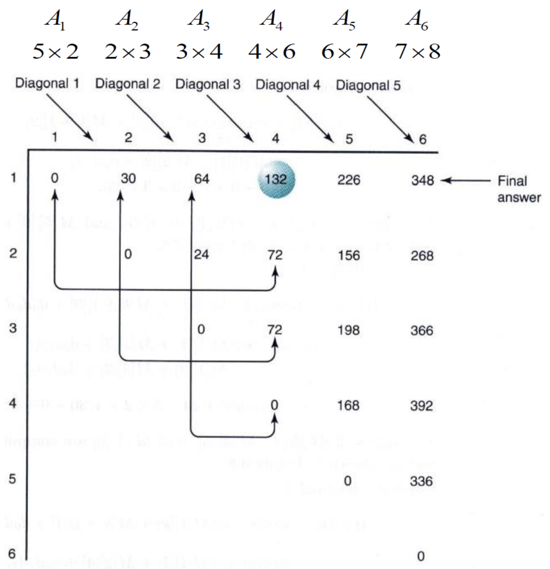
Example of computing M[1][4]

$5 \times 2 \times 6 = 60$
$5 \times 3 \times 6 = 90$
$5 \times 4 \times 6 = 120$

Why do we need these?

For M[1][4], we need:
M[1][1], M[2][4]
M[1][2], M[3][4]
M[1][3], M[4][4]

Why?



Minimum Multiplication

- **Problem**: Determine the minimum number of multiplications needed to multiply n matrices and an order that produces that minimum number.
- **Inputs**: The number of matrices n, and an array of integers d_k, indexed from 0 to n, where $d_{i-1} \times d_i$ is the dimension of the i-th matrix.
- **Outputs**: the minimum number of elementary multiplications needed to multiply the n matrices; a two-dimensional array P from which the optimal order can be obtained. $P[i][j]$ is the point where matrices i through j are split in an optimal order for multiplying the matrices.
- See Algorithm 3.6 in p.116.
- Check if the principle of optimality works for this case.

Minimum Multiplication Algorithm

```
int minmult(int n, const int d[], index P[][]) {
  index i, j, k, diagonal;
  int M[1..n, 1..n];
  for(i=1; i <= n; i++)
    M[i][j] = 0;
  for(diagonal = 1; diagonal <= n-1; diagonal++)
    for(i=1; i <= n-diagonal; i++) { // (i-loop)
      j = i + diagonal;
      M[i][j] = min( M[i][k] + M[k+1][j] +
                     d[i-1]*d[k]*d[j] );
                         where i <= k <= j-1 (k-loop)
      P[i][j] = a value of k that gave the min;
    }
  return M[1][n];
}
```


The P produced by the algorithm

	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

$P[1][6] = 1$

$(A_1((((A_2 A_3) A_4) A_5) A_6)).$

Every-Case Time Complexity of MM

- **Basic operation**: The instructions executed for each value of *n*. Included is a comparison to test for the minimum.
- **Input size**: *n*, the number of matrices to be multiplied.
- **Analysis**:
  - *j* = *i* + *diagonal*.
  - For a given values of *diagonal* and *i*, the number of passes through the ***k*-loop** =
    $$(j-1)-i+1 = i + diagonal - 1 - i + 1 = diagonal$$
  - For a given value of *diagonal*, the number of passes through the ***i*-loop** = *n* – *diagonal*
  - Therefore,
    $$\sum_{diagonal=1}^{n-1} [(n - diagonal) \times diagonal] = \frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

Remarks on Minimum Multiplication

- See Algorithm 3.7 to print the optimal order for multiplying *n* matrices.
  - *Order(i, j)* prints the optimal order for multiplying $A_i \times \ldots \times A_j$ with parentheses.
- Our algorithm $\Theta(n^3)$ for chained matrix multiplication is from Godbole(1973).
- Other algorithms:
  - Yao(1982) - $\Theta(n^2)$ by speeding up certain dynamic programming solutions.
  - Hu and Shing(1982, 1984) - $\Theta(n \lg n)$

Longest Common Subsequence (LCS) Problem

- The Longest Common Subsequence (LCS) problem is to find the longest common subsequence in two given sequences.
- Unlike the longest common substring, subsequences are not required to be consecutive.
- **Example**: Given X: ABCBDAB and Y: BDCABA, the LCS are BDAB, BCAB, and BCBA with length 4. (may not be unique)
- LCS are used in computational linguistics, bioinformatics, revision control systems, etc.

Brute Force Solution to LCS

- Given X[1..m] and Y[1..n], we can generate every subsequence of X and test if it is also in Y.
- There are 2^m possible subsequences of X. Each one can be any where in Y.
- Therefore the time complexity of the naïve solution above is $O(n\times2^m)$.
- The main problem is the combinatorial generation of all possible subsequences of X.
- Can LCS be solved with divide-and-conquer or dynamic-programming?

Recursive(D&C) Solution for LCS

- If X and Y both **end with the same element**, i.e. X[m]=Y[n], then the LCS(X[1..m], Y[1..n]) can be solved by solving the smaller instance LCS(X[1..m-1], Y[1..n-1]) and append X[m](or Y[n]) at the end.
- More specifically, if X[m]=Y[n], then

  LCS(X[1..m], Y[1..n]) = LCS(X[1..m-1], Y[1..n-1]) + X[m]
- This can be easily done with a recursive call.

Recursive(D&C) Solution for LCS

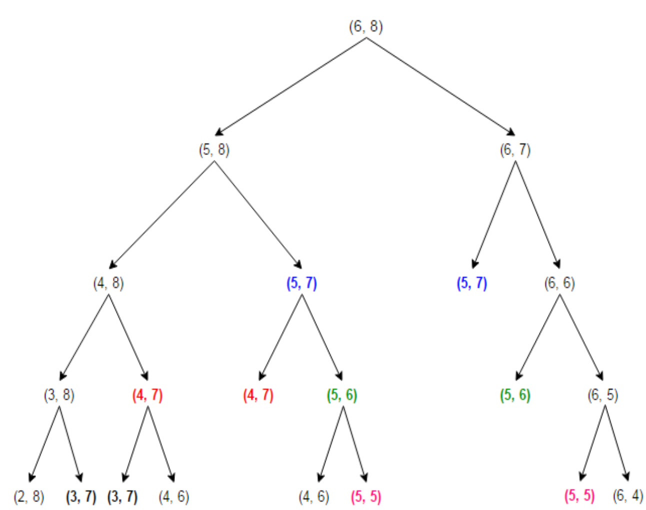
- If X and Y **does not** end with the same element, then the LCS of X and Y must be the longer of the two sequences LCS(X[1..m-1], Y[1..n]) and LCS(X[1..m], Y[1..n-1]).
- **Example**: Given X: ABCBDAB and Y: BDCABA.
- **Case 1**: the LCS ends with B. Then it cannot end with A. We can remove A from Y and solve the problem LCS(X[1..m], Y[1..n-1]).
- **Case 2**: the LCS does not end with B. Then we can remove B from X and solve the porblem LCS(X[1..m-1], Y[1..n]).

Recursive(D&C) Solution for LCS

```
int LCSrecursive(int m, int n,
       sequence X[1..m], sequence Y[1..n]) {
  if (m == 0 || n == 0) {
    return 0;
  }
  if (X[m] == Y[n]) {
    return LCSrecursive(X, Y, m-1, n-1) + 1;
  }
  // otherwise, X[m]≠Y[n]
  return max(LCSrecursive(X, Y, m, n-1),
             LCSrecursive(X, Y, m-1, n));
}
```


Analysis of LCSrecursive

- It is easy to show that the worst case time complexity of LCSrecursive is $O(2^{(m+n)})$.
- Subproblems are solved repeatedly.



Dynamic Programming for LCS

- The recursive property still apply. However, we need to formulate it in a bottom-up manner.

$$
\text{LCS[i][j]} = \begin{cases} 0 & \text{if } i==0 \text{ or } j==0 \\ \text{LCS[i-1][j-1]} + 1 & \text{if } X[i-1]==Y[j-1] \\ \text{larger(LCS[i–1][j], LCS[i][j–1])} & \text{if } X[i-1] \mathrel{!=} Y[j-1] \end{cases}
$$

- Note that X and Y are X[0..m-1] and Y[0..n-1].

$$
\begin{tikzcd}
\text{LCS[i-1][j-1]} \arrow[dr] & \text{LCS[i–1][j]} \arrow[d] \\
\text{LCS[i][j–1]} \arrow[r] & \text{LCS[i][j]}
\end{tikzcd}
$$

Example of DP for LCS 1/5

- Let X=ACADB, Y=CBDA
- Initially, fill the first row and first column with 0. (why?)

		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

Example of DP for LCS 2/5

- Fill the cells row by row:
  - If the row and column elements match, fill the cell with diagonal value + 1. (why?) Also point an arrow to that cell. (why?)
  - If they don't match, fill the cell with the larger of the left (column-1) and up (row-1) element. (why?) Also point an arrow to the element where you get the value from. (why?)

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0				
A	0				
D	0				
B	0				

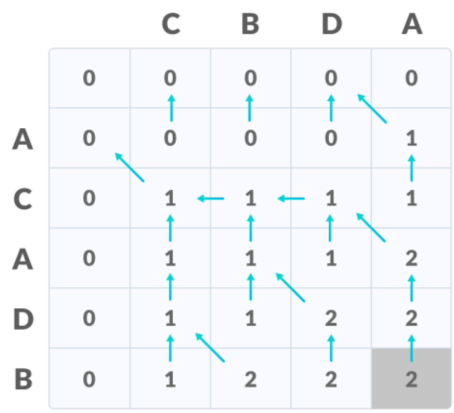
Example of DP for LCS 3/5

- Repeat until the matrix is filled completely.

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

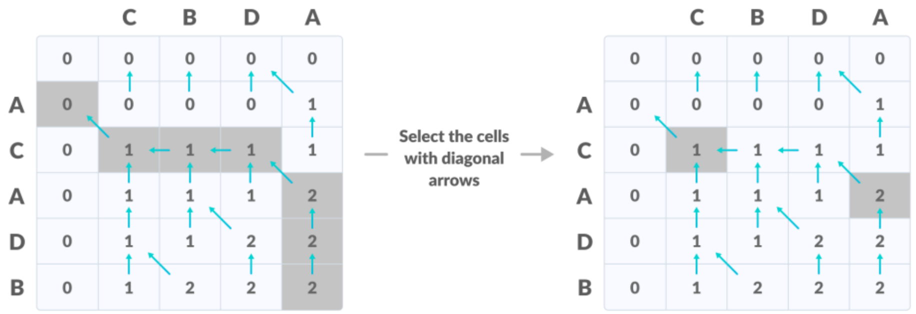
Example of DP for LCS 4/5

- The value of the last row and the last column is the length of the LCS.



Example of DP for LCS 5/5

- To find the longest common subsequence, start from the last element and follow the direction of the arrow. Select the cells with diagonal arrows. (why?) For this example, it is CA.



DP Algorithm for LCS

```
int LCSdynaProg(int m, int n,
       sequence X[0..m-1], sequence Y[0..n-1]) {
  index i, j;
  int LCS[0..m, 0..n];
  for (i=0; i<=m; i++) LCS[i][0]=0; // 1st column
  for (j=0; j<=n; j++) LCS[0][j]=0; // 1st row
  for (i=1; i<=m; i++)
    for (j=1; j<=n; j++)
      if (X[i-1] == Y[j-1]) // the chars matches
        LCS[i][j] = LCS[i-1][j-1] + 1;
      else // otherwise, the chars don't match
        LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1]);
  return LCS[m][n];
}
```


Analysis of Algorithm LCSdynaProg

- It is easy to prove that the every-case time complexity of `LCSdynaProg` is $\Theta(m\times n)$.
- Also need an extra space complexity of $\Theta(m\times n)$.
- The space complexity can be improved to $\Theta(n)$ since only the current row and the previous row are required. (why?)
- The algorithm does not keep the arrows.
- Try to improve the algorithm by keeping the arrows and print out both the LCS and its length.

Coin Change Problem (CCP)

- DP is also good for many counting problems.
- **Coin Change Problem**: Given an unlimited supply of coins of given denominations(面值), find the **total number** of distinct **ways** to get the desired change.
- Examples:

```
Input: S = { 1, 2, 3 }, N = 4

The total number of ways is 4

{ 1, 3 }
{ 2, 2 }
{ 1, 1, 2 }
{ 1, 1, 1, 1 }
```

```
Input: S = { 1, 3, 5, 7 }, N = 8

The total number of ways is 6

{ 1, 7 }
{ 3, 5 }
{ 1, 1, 3, 3 }
{ 1, 1, 1, 5 }
{ 1, 1, 1, 1, 1, 3 }
{ 1, 1, 1, 1, 1, 1, 1, 1 }
```


Recursive Solution to CCP

- Let $S = \{ S_1, S_2, .. , S_m \}$ be the set of coins and N is the desired change.
- CCP can be solved recursively by dividing all possible solutions into two sets:
  - Solutions that do not contain coin S_m.
  - Solutions that contain at least one S_m.
- Let count(S[], m, n) be the total count, than it can be written as the sum of count(S[], m - 1, n) and count(S[], m, n - S_m).
- **Exercise**: Formulate the recursive algorithm and analyze its complexity.

Dynamic Programming for CCP

- It should be easy to see that blind recursion of the above idea leads to repeated computation.
- DP to the rescue!
- Let `T[i][j]` = the total count of solutions for desired change `i` given coins { S_1, S_2, … , S_j }.
- The count can be divided into solutions with S_j,

  x = `T[i-`S_j`][j]` if `i-`$S_j \geq 0$ else 0

  and solutions w/o S_j.

  y = `T[i][j-1]` if `j` ≥ 1 else 0
- `T[i][j]` = x + y

Dynamic Programming for CCP

- **Exercise**: Formulate the dynamic programming algorithm for CPP with the `T[i][j]` defined in previous slide.
- Demonstrate that the time complexity of the algorithm is $O(m \times n)$. However, the additional space requirement is also $O(m \times n)$.
- Try to improve the algorithm so that the additional space required is $O(n)$ only.
- An optimization version of CPP is to find the minimum number of coins required to get the desired change. Give a DP algorithm to solve it.

Memorization (Top-down) Approach to Dynamic Programming

- The key idea to DP is to store solutions of sub-problems and look them up w/o recomputation.
- Therefore it is also possible to do DP in a top-down fashion by memorization of solutions (in a lookup table, for example) along the way.
- When doing recursive calls, we check the lookup table first.
- If the solution exists already, return immediately.
- If not, do the recursion and store the result.

Memorization DP for Fibonacci

```
int lookup[MAX]; // initialize all to NIL
int fibDP(int n) {
  if (lookup[n] == NIL) {
    if (n <= 1)
      lookup[n] = n;
    else
      lookup[n] = fibDP(n-1) + fibDP(n-2);
  }
  return lookup[n];
}
```

Exercises on Memorization Approach

- Try to redesign all our DP examples (and textbook examples) with memorization approach.
- How do we analyze the DP algorithms using the memorization approach?
- In comparison with the bottom-up approach, which one is more efficient?
- Which one is more intuitive and easier to formulate?
- All good exercises for mastering DP!!

When to Use DP

- DP is very general. Almost all problems that require to maximize or minimize certain quantity or counting problems (say to count all possible arrangements under certain condition) or certain probability problems can be solved with DP.
- The problems also exhibit the overlapping subproblems property. (i.e. subproblems are encountered repeatedly)
- Most DP problems (max/min) also satisfy the principle of optimality.

When Not to Use DP

- The benefit of DP comes from stored solutions.
- If the problem does not have overlapping subproblems property, then DP is not useful.
- For example, the binary search problem is not at all good for DP.
- The optimal binary search tree problem, however, is perfect for DP. (see textbook 3.5)

Assignment 3: Dynamic Programming

1. Assume that we want to develop a shopping APP in which a consumer can provide a wish list of items with preferences in the range of 1~100. Then given the current market prices of all items and a budget cap, find a set of items that maximize the sum of preferences with total spending under(≤) the budget cap.
   - Explain why the problem is (or not) good for DP.
   - Design and implement an algorithm for the problem.
   - Analyze the complexity of your algorithm.

Assignment 3: Dynamic Programming

2. Assume that you are in the Department of Tourism of the Hualien County. You are to divide the n miles Isozaki Coast (磯崎海灘) into segments of length i miles, $1 \leq i \leq n$, with different prices (pre-determined by the Taiwan government) for renting to the travel vendors. Design and implement an algorithm to find the optimal way of dividing the coast to maximize profit(sum of prices). Do the same as problem 1.
3. Textbook exercises: 3-4, 3-5, 3-6, 3-13

Due date: three weeks.