AAI2007 Introduction to Algorithms

Week 12: Dynamic Programming 2

Instructor: Jinyoung Han (jinyounghan@skku.edu)

Schedule

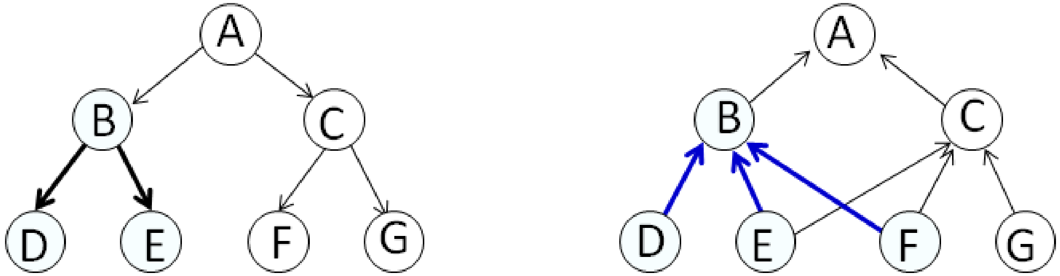
Tentative Schedule

수업일	내용
9/4	Course Introduction, Algorithm Basic, Level Test
9/11	Order of Complexity, List
9/18	Stack, Queue
9/25	건학 기념일
10/2	Tree, Binary Search Tree (BST)
10/9	Priority Queue, Heap, Heap Sort 한글날
10/16	Hash Table, Searching Revisited
10/23	Graph Basic
10/30	Midterm Exam
11/6	Graph Algorithms
11/13	Sorting
11/20	Dynamic Programming (1)
11/27	Dynamic Programming (2)
12/4	Greedy Algorithms
12/11	Algorithm Practice (Google software engineer)
12/18	Final Exam

Dynamic Programming

DP

- Dynamic Programming (DP)
  - An algorithm design technique for optimization problems
    - often minimizing or maximizing
- Like "divide and conquer", DP solves problems by combining solutions to sub-problems
- Unlike divide and conquer, sub-problems are not independent
  - Sub-problems may share sub-sub-problems



DC DP

Dynamic Programming

DP

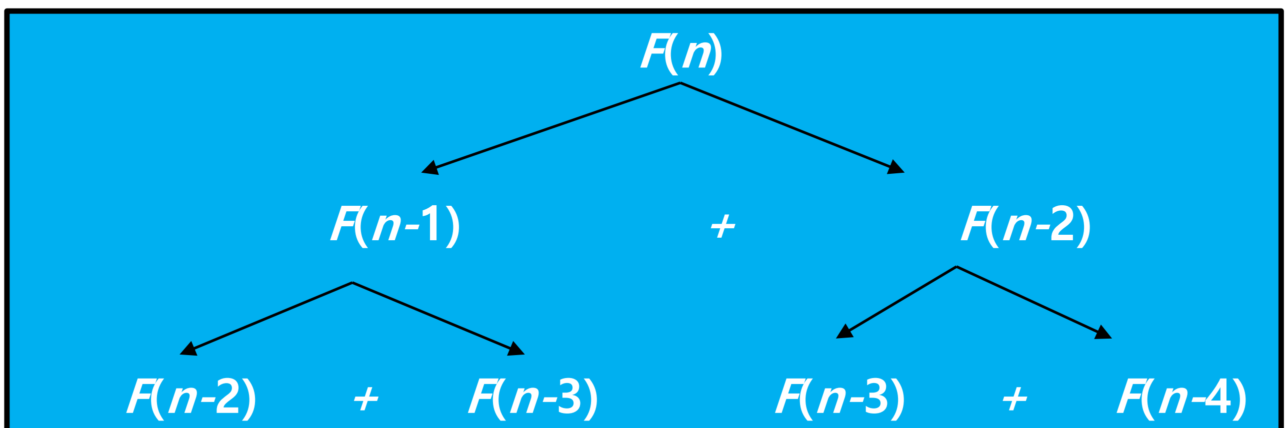
- When do we consider to apply DP?
  - Optimal substructure
    - Optimal solution of a bigger problem includes the optimal solution of a smaller problem
  - Overlapping recursive calls
    - Recursive solution results in many repetitions

Fibonacci Numbers

F-Numbers

- Computing the nth Fibonacci number recursively:
  - F(n) = F(n-1) + F(n-2)
  - F(0) = 0
  - F(1) = 1
  - **<u>A top-down approach</u>**

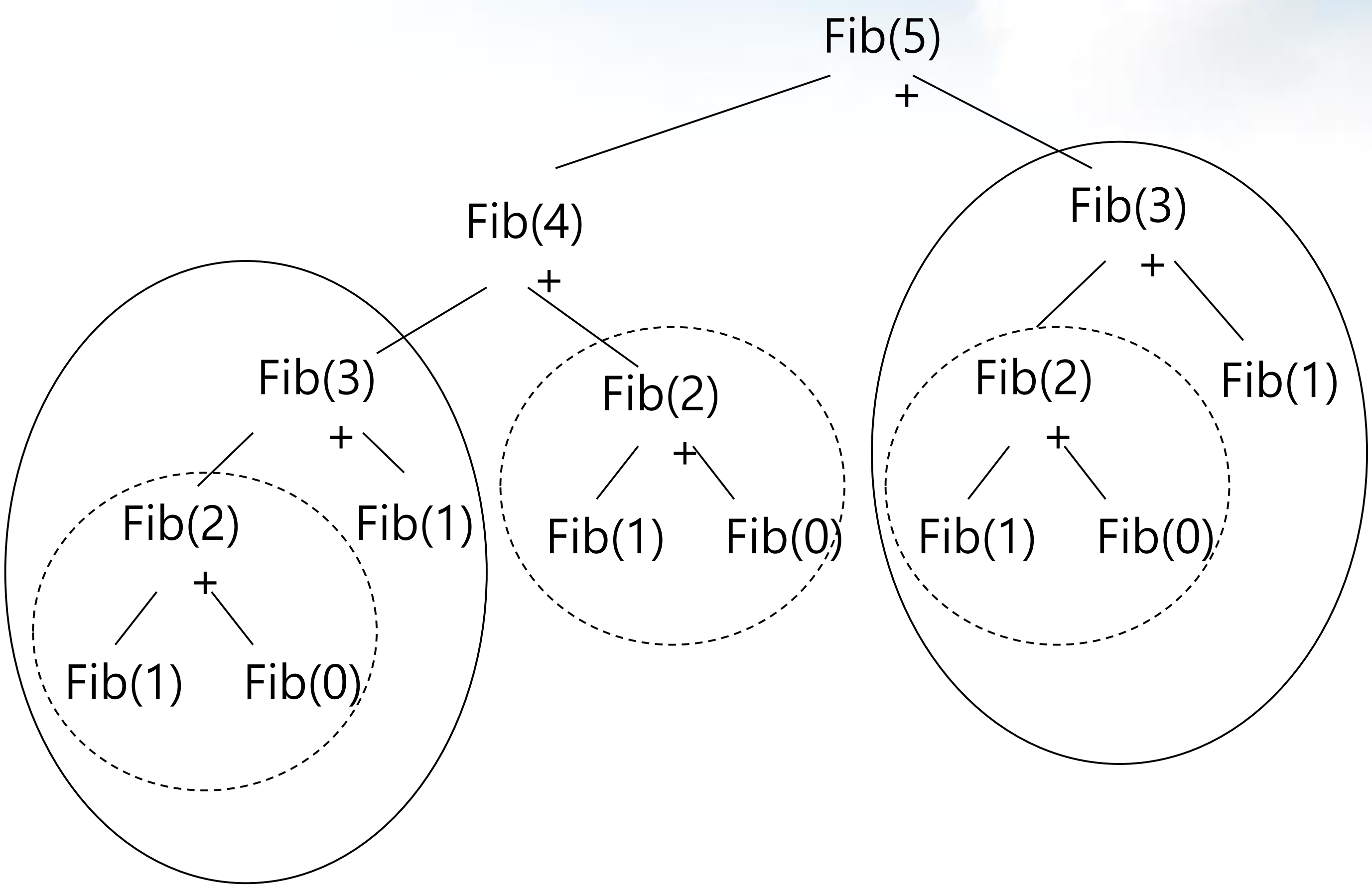
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```



Fibonacci Numbers

F-Numbers

- This top-down approach is not so inefficient
  - Re-compute many sub-problems



Fibonacci Numbers

F-Numbers

- Alternative bottom-up approach – dynamic programming (O(n)!)
  - F(0) = 0
  - F(1) = 1
  - F(2) = 1+0 = 1
  - …
  - F(n-2) =
  - F(n-1) =
  - F(n) = F(n-1) + F(n-2)

```
def fib(n):
  fibValues = [0,1]
  for i in range(2,n+1):
    fibValues.append(fibValues[i-1] + fibValues[i-2])
  return fibValues[n]
```

0	**1**	**1**	**. . .**	$F(n\text{-}2)$	$F(n\text{-}1)$	$F(n)$

Matrix Chain-Products

Optimal Solution

- Definition

$$\boxed{N_{0,n-1} = \min_{0\le k<n-1}\{N_{0,k} + N_{k+1,n-1} + d_0 d_{k+1} d_n\}}$$

- Then the optimal solution $N_{0,n-1}$ is the sum of two optimal sub-problems, $N_{0,k}$ and $N_{k+1,n-1}$ plus the time for the last multiplication

Matrix Chain-Products

DP

Algorithm *matrixChain(S)*:

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

for $i \leftarrow 1$ **to** $n - 1$ **do**

$\quad N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ **to** $n - 1$ **do**

$\quad$ { $b = j - i$ is the length of the problem }

$\quad$ **for** $i \leftarrow 0$ **to** $n - b - 1$ **do**

$\quad\quad j \leftarrow i + b$

$\quad\quad N_{i,j} \leftarrow +\infty$

$\quad\quad$ **for** $k \leftarrow i$ **to** $j - 1$ **do**

$\quad\quad\quad N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i\, d_{k+1}\, d_{j+1}\}$

return $N_{0,n-1}$

(1) Initialize C[1,1], C[2,2], ⋯, C[n-1,n-1] as 0
- No computation is needed for same matrix

Matrix Chain-Products

DP

Algorithm *matrixChain*(S):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

for $i \leftarrow 1$ **to** $n - 1$ **do**

$N_{i,i} \leftarrow \mathbf{0}$

for $b \leftarrow 1$ **to** $n - 1$ **do**

{ $b = j - i$ is the length of the problem }

for $i \leftarrow 0$ **to** n – b - 1 **do**

$j \leftarrow i + b$

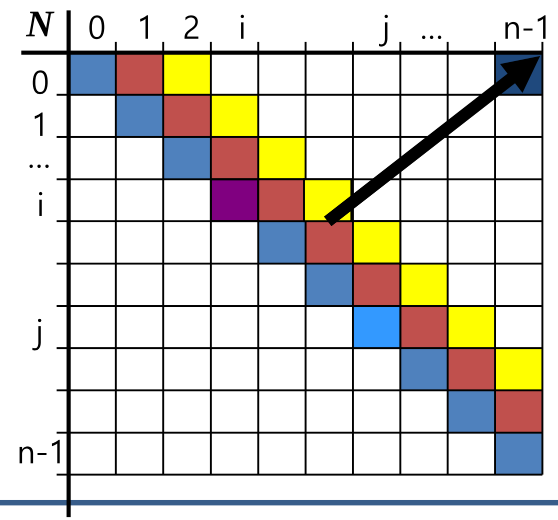
$N_{i,j} \leftarrow +\infty$

for $k \leftarrow i$ **to** $j - 1$ **do**

$N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i\, d_{k+1}\, d_{j+1}\}$

return $N_{0,n-1}$

(2) b: length of a sub-problem



Matrix Chain-Products

DP

Algorithm *matrixChain*(S):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

for $i \leftarrow 1$ **to** $n - 1$ **do**
  $N_{i,i} \leftarrow \mathbf{0}$
for $b \leftarrow 1$ **to** $n - 1$ **do**
  { $b = j - i$ is the length of the problem }
  for $i \leftarrow 0$ **to** $n - b - 1$ **do**
    $j \leftarrow i + b$
    $N_{i,j} \leftarrow +\infty$
    for $k \leftarrow i$ **to** $j - 1$ **do**
      $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$
return $N_{0,n-1}$

(3) b=1 -> subproblems with size 2
b=2 -> subproblems with size 3
(e.g., $A_1xA_2xA_3$, $A_2xA_3xA_4$, ...)
...

Matrix Chain-Products

DP

Algorithm *matrixChain*(S):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

for $i \leftarrow 1$ **to** $n - 1$ **do**

$N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ **to** $n - 1$ **do**

{ $b = j - i$ is the length of the problem }

for $i \leftarrow 0$ **to** n – b - 1 **do**

$j \leftarrow i + b$

$N_{i,j} \leftarrow +\infty$

for $k \leftarrow i$ **to** $j - 1$ **do** (4) To find the minimum # operations

$N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i\, d_{k+1}\, d_{j+1}\}$

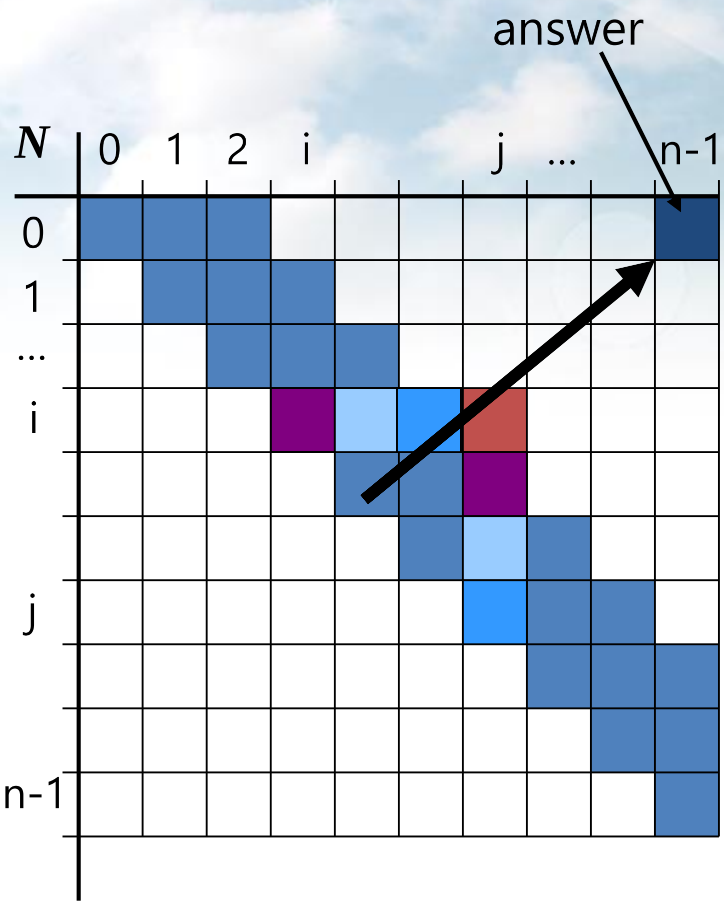
return $N_{0,n-1}$

Matrix Chain-Products

Visualization

- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from previous entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$



Matrix Chain-Products

Visualization

- A_0: 30 X 35; A_1: 35 X15; A_2: 15X5; A_3: 5X10; A_4: 10X20; A_5: 20 X 25

0	1	2	3	4	5	
0	15,750	7,875	9,375	11,875	15,125	0
	0	2,625	4,375	7,125	10,500	1
		0	750	2,500	5,375	2
			0	1,000	3,500	3
				0	5,000	4
					0	5

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

$$\begin{aligned}
&N_{1,4} = \min\{ \\
&N_{1,1} + N_{2,4} + d_1 d_2 d_5 = 0 + 2500 + 35*15*20 = 13000, \\
&N_{1,2} + N_{3,4} + d_1 d_3 d_5 = 2625 + 1000 + 35*5*20 = 7125, \\
&N_{1,3} + N_{4,4} + d_1 d_4 d_5 = 4375 + 0 + 35*10*20 = 11375 \\
&\} \\
&= 7125
\end{aligned}$$

In This Lecture

Outline

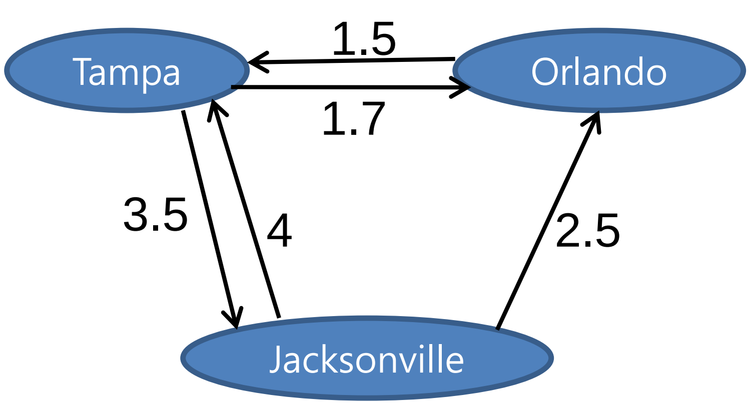
1. Floyd-Warshall Algorithm
2. Knapsack Problem
3. Longest Common Subsequence

01. Floyd-Warshall Algorithm

Problem

- A weighted, directed graph is a collection vertices connected by weighted edges (where the weight is some real number)
  - One of the most common examples of a graph in the real world is a road map
    - Each location is a vertex and each road connecting locations is an edge
    - We can think of the distance traveled on a road from one location to another as the weight of that edge

	Tampa	Orlando	Jaxville
Tampa	0	1.7	3.5
Orlando	1.5	0	∞
Jax	4	2.5	0

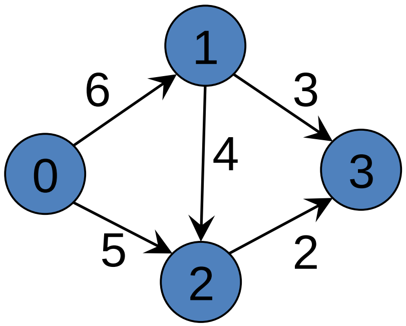


01. Floyd-Warshall Algorithm

Problem

- Store a weighted and directed graph -- Adjacency Matrix
  - Let D be an edge-weighted graph in adjacency-matrix form
  - D(i,j) is the weight of edge (i, j), or ∞ if there is no such edge
- Update matrix D, with the shortest path through immediate vertices

$$
D = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}
\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\
\begin{array}{|c|c|c|c|} \hline 0 & 6 & 5 & \infty \\ \hline \infty & 0 & 4 & 3 \\ \hline \infty & \infty & 0 & 2 \\ \hline \infty & \infty & \infty & 0 \\ \hline \end{array} \end{array}
$$



01. Floyd-Warshall Algorithm

Problem

- Given a weighted graph, we want to know the shortest path from one vertex in the graph to another
  - The Floyd-Warshall algorithm determines the shortest path between all pairs of vertices in a graph

- What is the difference between Floyd-Warshall and Dijkstra's??

01. Floyd-Warshall Algorithm

Problem

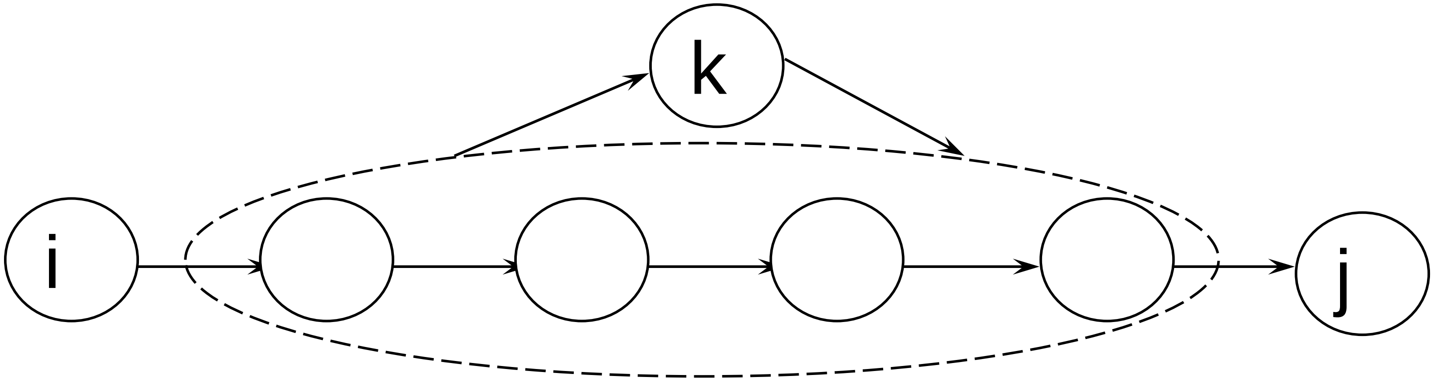
- If V is the number of vertices, Dijkstra's runs in $O(V^2)$
  - We could just call Dijkstra |V| times, passing a different source vertex each time
  - $O(V \times V^2) = O(V^3)$
    - same runtime as the Floyd-Warshall Algorithm

- BUT, Dijkstra's doesn't work with negative-weight edges
- Also, the structure of the Floyd's algorithm is simple

01. Floyd-Warshall Algorithm

Algorithm

- Let's go over the premise of how Floyd-Warshall algorithm works
  - Let the vertices in a graph be numbered from 1 … n
  - Consider the subset {1,2,…, k} of these n vertices
  - Imagine finding the shortest path from vertex i to vertex j that uses vertices in the set {1,2,…,k} only
  - There are two situations:
    - k is an intermediate vertex on the shortest path
    - k is not an intermediate vertex on the shortest path

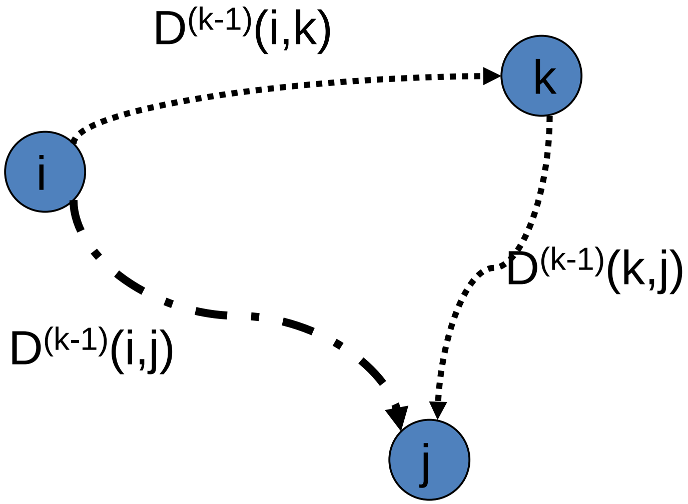


01. Floyd-Warshall Algorithm

Algorithm

- Sub-problems
  - Pass through k: $D^{K-1}[i][k] + D^{K-1}[k][j]$
  - Otherwise: $D^{K-1}[i][j]$

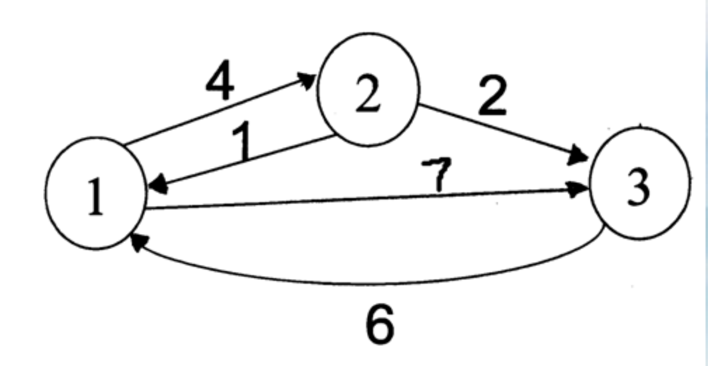
$$D^{(k)}(i,j) = \min\{D^{(k-1)}(i,j),\ D^{(k-1)}(i,k) + D^{(k-1)}(k,j)\}$$



01. Floyd-Warshall Algorithm

Algorithm

- Example



$$D^{(0)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix}$$

Original weights

$$D^{(1)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 1:

D(3,2) = D(3,1) + D(1,2)

$$D^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 2:

D(1,3) = D(1,2) + D(2,3)

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 3:

Nothing changes

01. Floyd-Warshall Algorithm

Algorithm

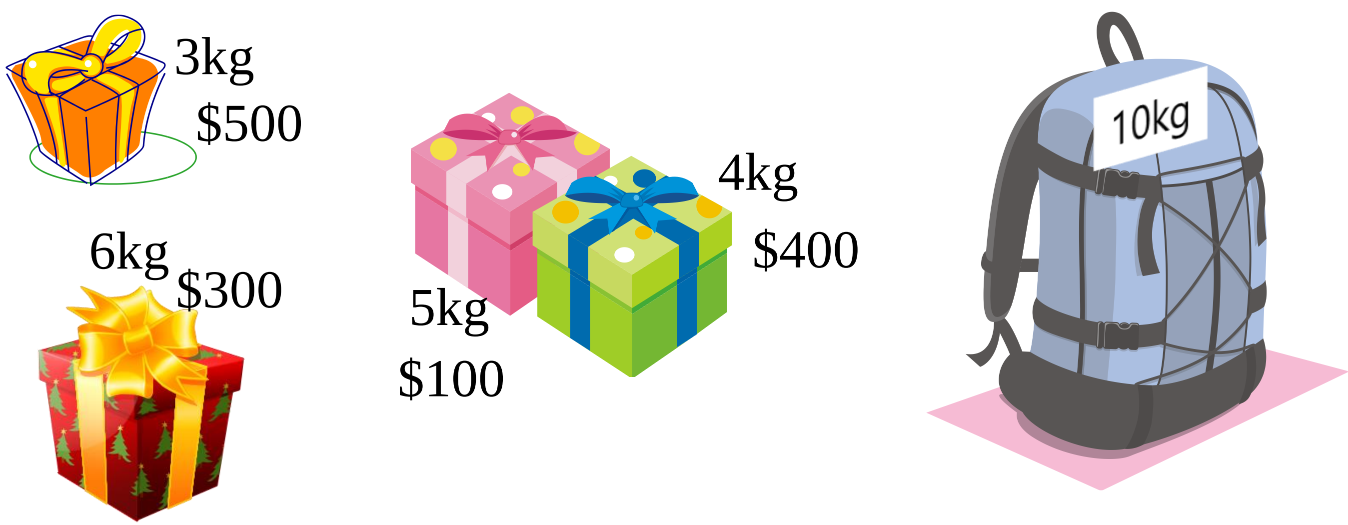
- Looking at this example, we can come up with the following algorithm:
  - Let D store the matrix with the initial graph edge information initially, and update D with the calculated shortest paths
- ```
  For k=1 to n {
      For i=1 to n {
              For j=1 to n
                      D[i,j] = min(D[i,j],D[i,k]+D[k,j])
      }
  }
  ```
- The final D matrix will store all the shortest paths
  ```

Knapsack Problem

02. Knapsack Problem

Problem

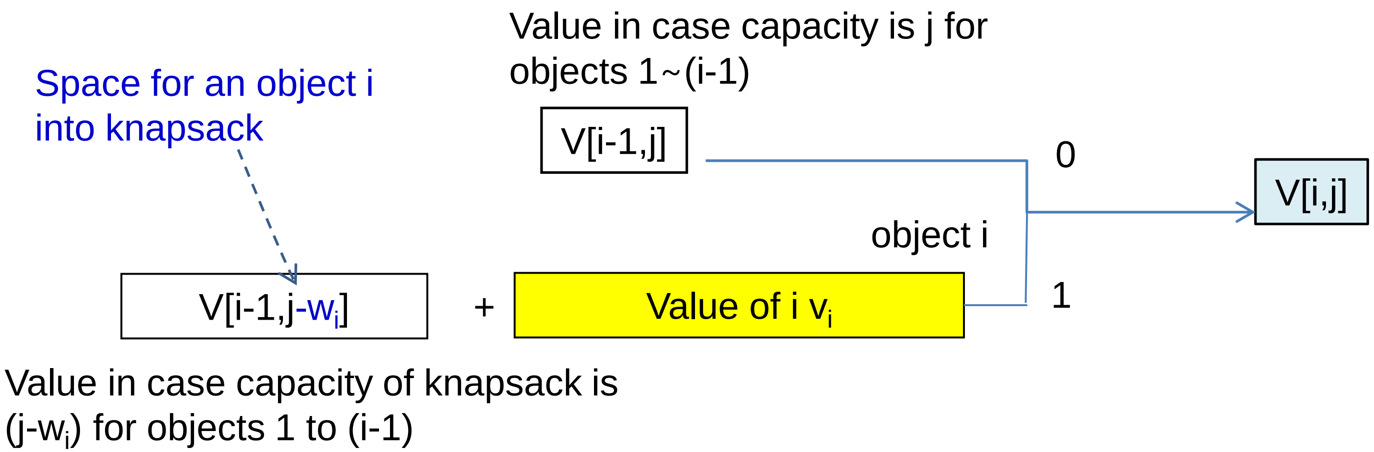
- Given n items of
  - integer weights: $w_1 \quad w_2 \; \dots \; w_n$
  - values: $v_1 \quad v_2 \; \dots \; v_n$
  - a knapsack of integer capacity W
- Find most valuable subset of the items that fit into the knapsack
  - 0-1 Knapsack problem



02. Knapsack Problem

Problem

- Consider instance defined by first i items and capacity j (j <= W)
- Let V[i,j] be optimal value of such an instance. Then,
- $$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{if } j-w_i >= 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$
- Initial conditions: V[0,j] = 0 and V[i,0] = 0



02. Knapsack Problem

Algorithm

- Algorithm Knapsack(w[1..n], v[1..n], W)

```
var V[0..n,0..W],  P[1..n,1..W]: int
for j := 0 to W do
        V[0,j] := 0
for i := 0 to n do
        V[i,0] := 0
for i := 1 to n do
        for j := 1 to W do
                if  w[i] <= j and v[i] + V[i-1,j-w[i]] > V[i-1,j] then
                        V[i,j] := v[i] + V[i-1,j-w[i]];
                else
                        V[i,j] := V[i-1,j];
return V[n,W]
```

O(nW)

02. Knapsack Problem

Example

- Example: Knapsack of capacity W = 5

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity j

		0	1	2	3	4	5
	0	0	0	0	0	0	0
w1 = 2, v1= 12	1	0	0	12	12	12	12
w2 = 1, v2= 10	2	0	10	12	22	22	22
w3 = 3, v3= 20	3	0	10	12	22	30	32
w4 = 2, v4= 15	4	0	10	15	25	30	37

Backtracing finds the actual optimal subset, i.e. solution.

Longest Common Subsequence

03. Longest Common Subsequence

Problem

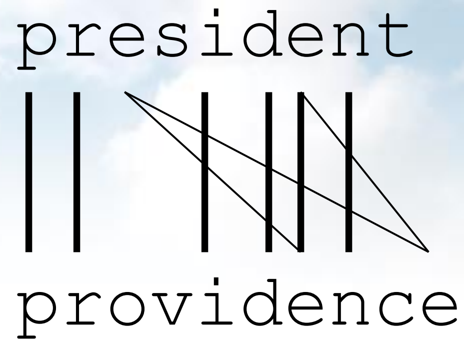
- Given sequences x[1..m] and y[1..n], find a longest common subsequence of both
  - A subsequence of a sequence/string S is obtained by deleting zero or more symbols from S
  - For example, the following are some subsequences of “president”: pred, sdn, predent
  - In other words, the letters of a subsequence of S appear in order in S, but they are not required to be consecutive

- Example: x=ABCBDAB and y=BDCABA,
  - BCA is a common subsequence and
  - BCBA and BDAB are two LCSs

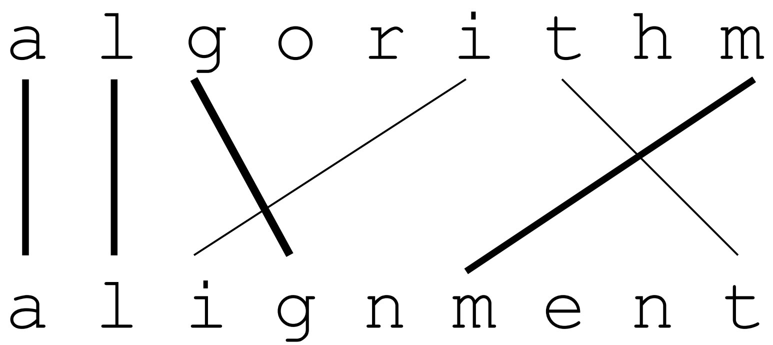
03. Longest Common Subsequence

Examples

- An example
  - Sequence 1: president
  - Sequence 2: providence
  - Its LCS is priden



- Another example
  - Sequence 1: algorithm
  - Sequence 2: alignment
  - One of its LCS is algm



03. Longest Common Subsequence

Problem

- LenLCS(i, j): the length of an LCS of Xi and Yj
- Zk: an LCS of Xi and Yj
- If Xi and Yj do not end with the same character there are two possibilities:
  - Case 1: either the LCS does not end with xi,
  - Case 2: or it does not end with yj

03. Longest Common Subsequence

Problem

- Case 1: Xi and Yj end with xi=yj

$$X_i \quad \boxed{x_1 \;\; x_2 \;\; \ldots \; x_{i-1} \;\big|\; x_i}$$

$$Y_j \quad \boxed{y_1 \; y_2 \;\; \ldots \;\; y_{j-1} \;\big|\; y_j = x_i}$$

$$Z_k \quad \boxed{z_1 \; z_2 \ldots z_{k-1} \;\big|\; z_k = y_j = x_i}$$

Z_k is Z_{k-1} followed by $z_k = y_j = x_i$ where
Z_{k-1} is an LCS of X_{i-1} and Y_{j-1} and
$LenLCS(i, j) = LenLCS(i-1, j-1) + 1$

03. Longest Common Subsequence

Problem

- Case 2: Xi and Yj end with xi ≠ yj

X_i $\boxed{x_1\ \ x_2\ \ \ldots\ x_{i-1}\ \ x_i}$

Y_j $\boxed{y_1\ y_2\ \ \ldots\ \ y_{j-1}} \boxed{y_j}$

Z_k $\boxed{z_1\ z_2 \ldots z_{k-1}} \boxed{z_k \neq y_j}$

Z_k is an LCS of X_i and $Y_{j\text{-}1}$

X_i $\boxed{x_1\ \ x_2\ \ \ldots\ x_{i-1}} \boxed{x\ _i}$

Y_j $\boxed{y_j\ y_1\ y_2\ \ \ldots y_{j-1}\ \ y_j}$

Z_k $\boxed{z_1\ z_2 \ldots z_{k-1}} \boxed{z_k \neq x_i}$

Z_k is an LCS of $X_{i\text{-}1}$ and Y_j

$$LenLCS(i, j)=\max\{LenLCS(i, j\text{-}1), LenLCS(i\text{-}1, j)\}$$

03. Longest Common Subsequence

Problem

- Formulation

$$lenLCS(i,j) = \begin{cases} 0 & \text{if } i = 0, \text{ or } j = 0 \\ lenLCS(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{lenLCS(i-1, j), lenLCS(i, j-1)\} & \text{otherwise} \end{cases}$$

03. Longest Common Subsequence

Solution

- Initialize the first row and the first column of the matrix LenLCS to 0
- Calculate LenLCS (1, j) for j = 1,…, n
- Then the LenLCS (2, j) for j = 1,…, n, etc.
- It is easy to see that the computation is O(mn)

03. Longest Common Subsequence

Algorithm

```
m ← length[X]
n ← length[Y]
for i ← 1 to m do
   c[i, 0] ← 0
for j ← 1 to n do
   c[0, j] ← 0

for i ← 1 to m do
 for j ← 1 to n do
    if x_i = y_j
      c[i, j] ← c[i-1, j-1]+1
    else
            if c[i-1, j] ≥ c[i, j-1]
                  c[i, j] ← c[i-1, j]
            else
                  c[i, j] ← c[i, j-1]
return c and b
```

03. Longest Common Subsequence

Example

- Example

	y_j	B	D	C	A
x_i	0	0	0	0	0
A	0	0	0	0	1
B	0	1	1	1	1
C	0	1	1	2	2
B	0	1	1	2	2

What You Need to Know

Summary

- Dynamic programming problems
  - Fibonacci Numbers
  - Matrix Chain-Products Problem
  - Floyd-Warshall Algorithm
  - Knapsack Problem
  - Longest Common Subsequence
  - And many others…

Thanks

Week 12: Dynamic Programming 2
Instructor: Jinyoung Han (jinyounghan@skku.edu)