Dynamic Programming - II

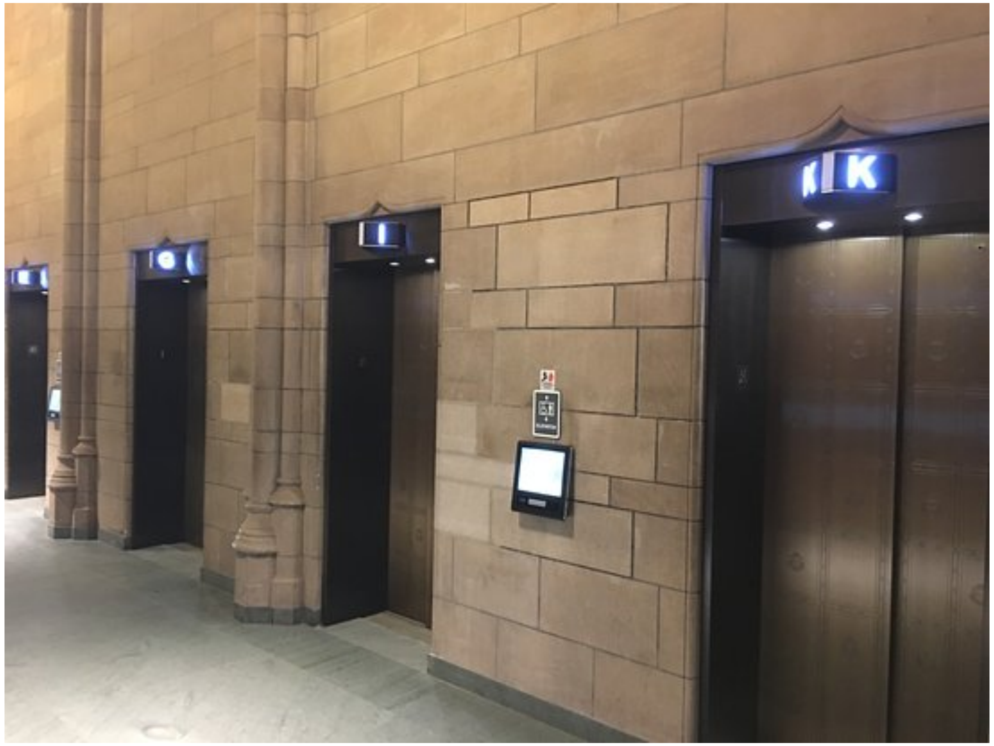
CSIE 2136 Algorithm Design and Analysis, Fall 2018

https://cool.ntu.edu.tw/courses/61

Hsu-Chun Hsiao

National Taiwan University

暖身運動




1 2 3
4 5 6
7 8 9
G 0 B
Please
enter floor
destination


Floor 36
Please proceed
to elevator
H

Announcement

Homework assignments

- Mini-hw4 due next week
- HW1 due next week
- HW2 due in 4 weeks

Email policy

- Please put [ADA 2018] in the title
- Please address to the TAs and mention who you are

Agenda

Sequence Alignment Problem (序列比對)

- Longest Common Subsequence
- A space-saving algorithm

Knapsack Problem (背包問題)

- 0/1 knapsack
- Unbounded knapsack
- Multiple-choice knapsack
- Multidimensional knapsack
- Fractional knapsack

DP and optimization problems

Dynamic programming are often applied to solving optimization problems (最佳化問題)

- 從問題的多個解之中，選出 最佳的
- 最佳的解可能有很多個，找出一個就好了

Examples of optimization problems

- 從兩個字串中，找出最長的共同子字串
- 給一個背包和一堆物品，找出背包最多能裝多少物品
- ...

DP and optimization problems

To apply DP, an optimization problem must exhibit two key properties:

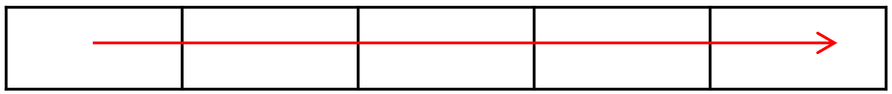
- **Overlapping subproblems**
- **Optimal substructure** – an optimal solution can be constructed from optimal solutions to subproblems
  - Reduce search space, as we don't need to consider non-optimal solutions to a subproblem

Dynamic programming: 4 steps

1. Characterize the structure of an optimal solution
   - **Overlapping subproblems:** revisits same subproblem repeatedly
   - **Optimal substructure:** an optimal solution to the problem contains within it optimal solutions to subproblems
2. Recursively define the value of an optimal solution
   - Express the solution of the original problem in terms of optimal solutions for smaller problems
3. Compute the value of an optimal solution
   - Typically in a bottom-up fashion
4. Construct an optimal solution from computed info
   - Step 3 and Step 4 may be combined

Bottom-up with tabulation

按問題大小順序填表（小問題要先解決）

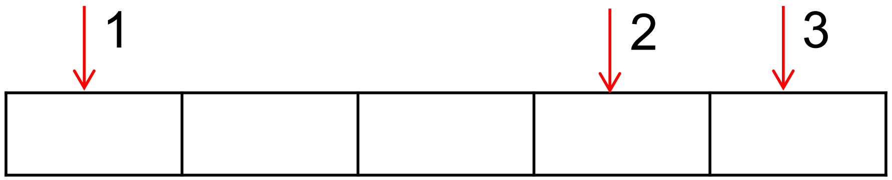


適合用於每個小問題都得解決的情況

Top-down with memoization

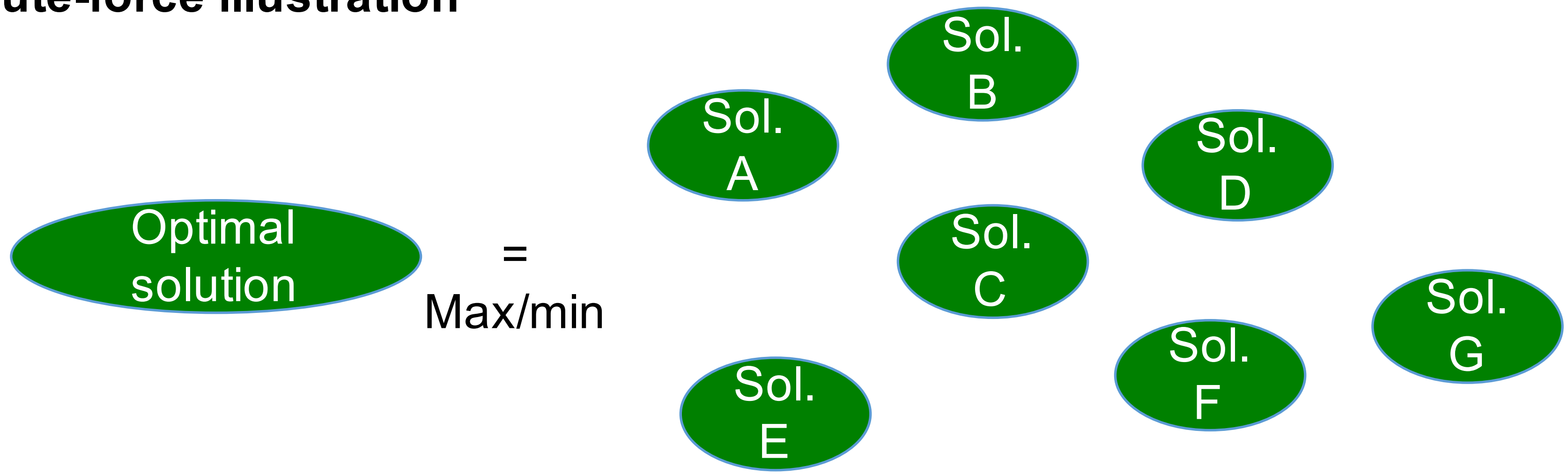
用遞迴解，把小問題的解答記在備忘錄裡

可看成是跳著填表

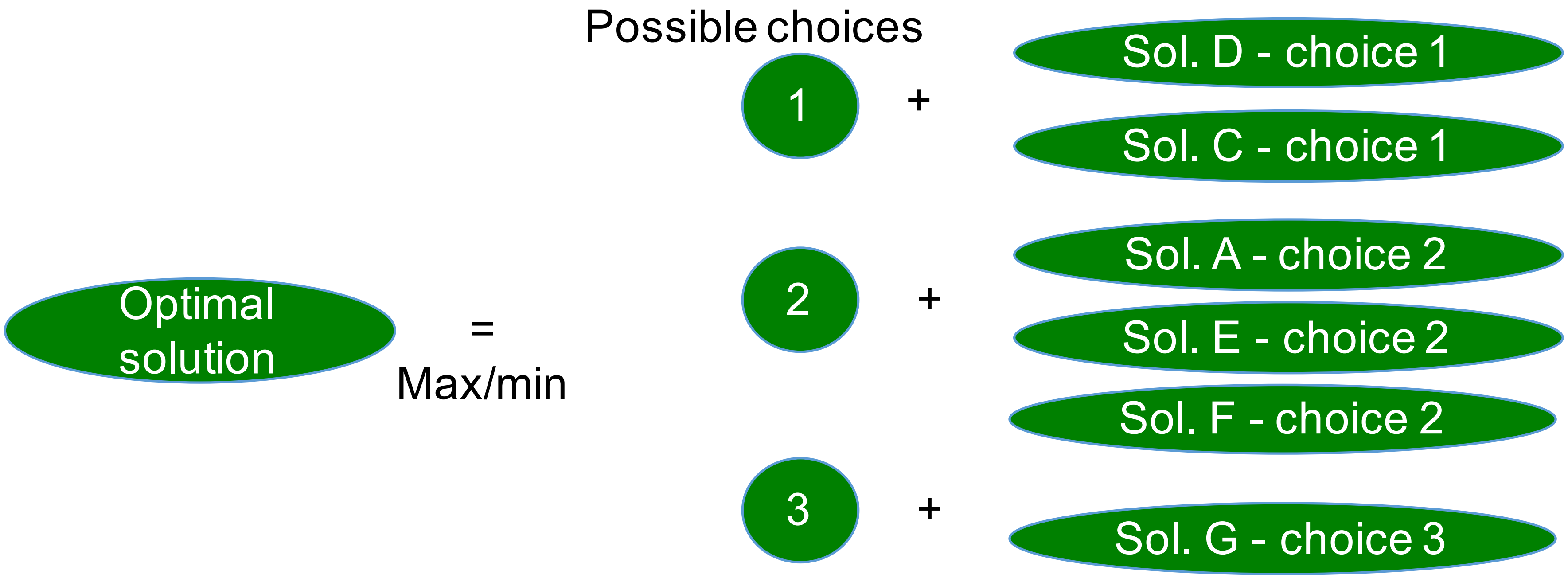


適合用於不需要解決所有的小問題的情況

Brute-force illustration

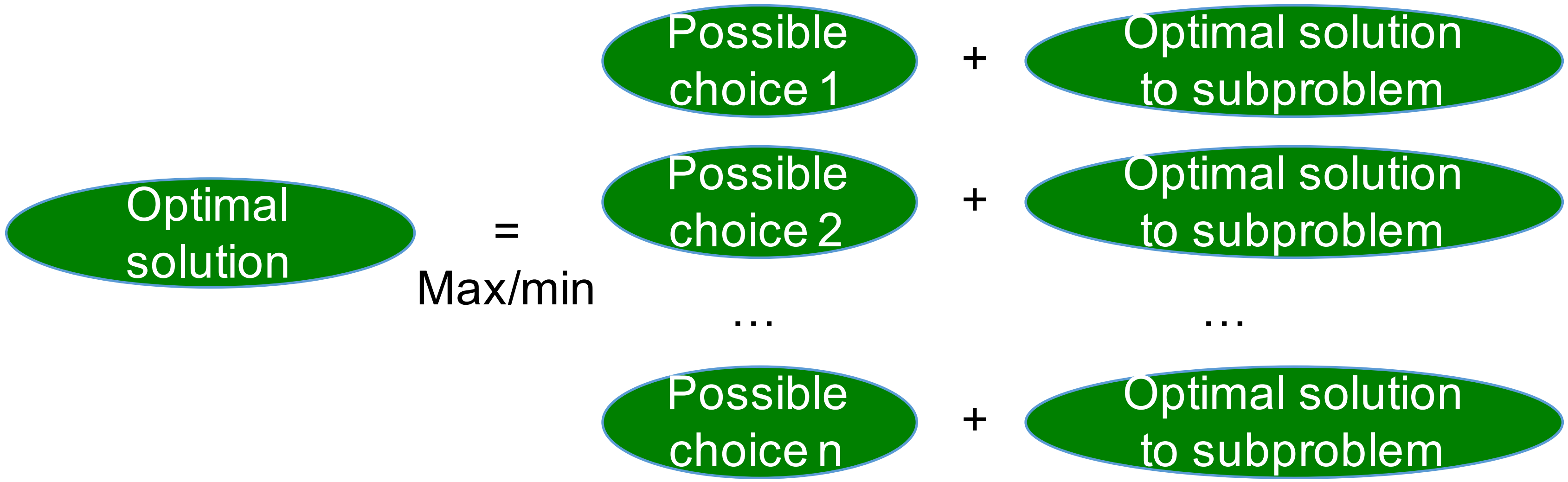


Same brute-force illustration (grouping solutions by exclusive choices)



DP illustration

Optimal substructure property ensures that we only need to consider an optimal solution to each subproblem



對每個可能的情況，只要考慮相對應的subproblem的一個optimal解就可以了。
其他的solution都不用考慮了！

Greedy illustration

Greedy choice property ensures that we only need to consider one greedy choice (among all possible choices)

Optimal solution =? Greedy choice + Optimal solution to subproblem

Sequence Alignment (序列比對)

Textbook Chapter 15.4

Chapter 6.6 in Algorithm Design by Kleinberg & Tardos

廢文大賽

今天舉行動物園廢文大賽，勝利條件是亂打的文字內容最接近 “banana”

身為評審的鸚鵡們，該如何選出冠軍？

參賽者一號 :`aeniqadikjaz`

參賽者二號 :`svkbrlvpnzanczyqza`

Which one is more **similar** to banana?

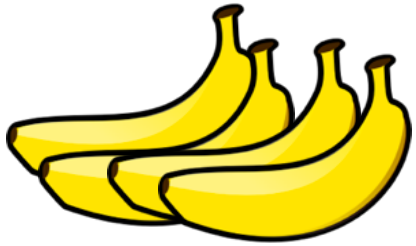
String similarity

Metric #1: Longest Common Subsequence (最長共同子序列)

- The longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings
- Textbook Chapter 15.4

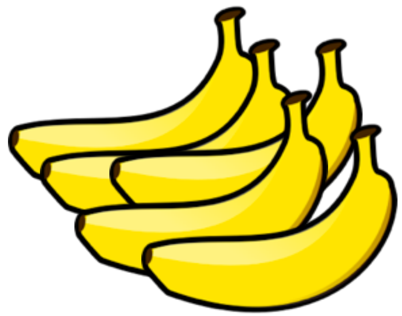
參賽者一號：

```
banana
aeniqadikjaz
```



參賽者二號：

```
banana
svkbrlvpnzanczyqza
```



參賽者一號：評審不公(叭)
打得比較長當然對得比較多

The infinite monkey theorem (無限猴子定理)：
從機率的觀點來看，只要時間夠長，亂打字的猴子幾乎必然能打出任何給定的內容，比如說背包問題的演算法。

String similarity

Metric #2: edit distance

- Quantifies the dissimilarity of two strings
- “Minimal” work to transform one string into the other

參賽者一號：

banana

aeniqadikjaz

Gap **mismatch**

b	a	-	n	-	-	a	n	-	-	-	a	-
-	a	e	n	i	q	a	d	i	k	j	a	z

8 gaps, 1 mismatch

參賽者二號：

banana

svkbrlvpnzanczyqza

-	-	-	b	a	-	-	-	n	-	a	n	-	-	-	-	-	a
s	v	k	b	r	l	v	p	n	z	a	n	c	z	y	q	z	a

12 gaps, 1 mismatch

Sequence Alignment Problem

Given two strings X and Y, find **min cost** alignment

- $X = x_1\ x_2\ \dots\ x_m$
- $Y = y_1\ y_2\ \dots\ y_n$
- Cost = # of gaps * δ_{gap} + $\Sigma_{p,q \text{ are aligned}}\ \delta_{pq}$

gap penalty Mismatch penalty for aligning p with q; $\delta_{pp}=0$

Ex. $\delta_{gap}=4$, $\delta_{pq}=7$ if p!=q:

X = `banana`, Y = `aeniqadikjaz`

Cost of 1st alignment = 39

Cost of 2nd alignment = 40

b	a	-	n	-	-	a	n	-	-	-	a	-
-	a	e	n	i	q	a	d	i	k	j	a	z

8 gaps, 1 mismatch

Ex. $\delta_{gap}=1$, $\delta_{pq}=7$ if p!=q:

Cost of 1st alignment = 15

Cost of 2nd alignment = 10

b	a	-	n	-	-	a	n	-	-	-	-	a	-
-	a	e	n	i	q	a	-	d	i	k	j	a	z

10 gaps

Step 1: Characterize an optimal solution

Sequence alignment: Given two strings X and Y, find **min cost** alignment
- $X = x_1 x_2 \dots x_m$, $Y = y_1 y_2 \dots y_n$
- Cost = # of gaps * $\delta_{gap} + \Sigma_{p,q \text{ are aligned}} \delta_{pq}$

SA(i, j) = Sequence Alignment Problem considering prefix strings $x_1 \dots x_i$ and $y_1 \dots y_j$

Suppose OPT is an optimal solution to SA(i, j)

Pick x_i and y_j, there are three possibilities:

1. x_i and y_j are aligned in OPT — OPT\{i,j}是SA(i-1,j-1)的一個最佳解
2. x_i is aligned with a gap in OPT — OPT\{i, -}是SA(i-1, j) 的一個最佳解
3. y_j is aligned with a gap in OPT — OPT\{-, j}是SA(i, j-1) 的一個最佳解

練習：證明以上觀察成立，以確認有 optimal substructure

Step 2: Recursively define the value of an optimal solution

SA(i, j) = Sequence Alignment Problem considering prefix strings $x_1 \ldots x_i$ and $y_1 \ldots y_j$

Case 1: x_i and y_j are aligned in OPT

- OPT\{i,j}是SA(i-1,j-1)的一個最佳解

$M[i,j]=\delta_{x_i,y_j}+M[i-1,j-1]$

Case 2: x_i is aligned with a gap in OPT

- OPT\{i, -} 是SA(i-1, j) 的一個最佳解

$M[i,j]=\delta_{gap}+M[i-1,j]$

Case 3: y_j is aligned with a gap in OPT

- OPT\{-, j} 是SA(i, j-1) 的一個最佳解

$M[i,j]=\delta_{gap}+M[i,j-1]$

M[i,j] = the value of an optimal solution to SA(i, j)

$$M[i,j]=\begin{cases} j\delta_{gap}, & \text{if } i = 0 \text{ (base case)} \\ i\delta_{gap}, & \text{if } j = 0 \text{ (base case)} \\ \min\{\delta_{gap}+M[i-1,j],\delta_{gap}+M[i,j-1],\delta_{x_i,y_j}+M[i-1,j-1]\}, & \text{otherwise} \end{cases}$$

Step 3: Compute the value of an optimal solution

SA(i, j) = Sequence Alignment Problem considering prefix strings $x_1 \ldots x_i$ & $y_1 \ldots y_j$

M[i,j] = the value of an optimal solution to SA(i, j)

$$M[i,j]=\begin{cases} j\delta_{gap}, & \text{if } i = 0 \text{ (base case)} \\ i\delta_{gap}, & \text{if } j = 0 \text{ (base case)} \\ \min\{\delta_{gap}+M[i-1,j],\ \delta_{gap}+M[i,j-1],\ \delta_{x_i,y_j}+M[i-1,j-1]\}, & \text{otherwise} \end{cases}$$

X\Y	0	1	2	3	…	n
0						
1						
…					M[i,j]	
m						

Our goal

填表順序

Step 3: Compute the value of an optimal solution

SA(i, j) = Sequence Alignment Problem considering prefix strings $x_1 \ldots x_i$ and $y_1 \ldots y_j$

M[i,j] = the value of an optimal solution to SA(i, j)

$$M[i,j] = \begin{cases} j\delta_{gap}, & \text{if } i = 0 \text{ (base case)} \\ i\delta_{gap}, & \text{if } j = 0 \text{ (base case)} \\ \min\{\delta_{gap}+M[i-1,j],\ \delta_{gap}+M[i,j-1],\ \delta_{x_i,y_j}+M[i-1,j-1]\}, & \text{otherwise} \end{cases}$$

Ex. $\delta_{gap}=4$, $\delta_{pq}=7$ if p!=q:

			a	e	n	i	q	a	d	i	k	j	a	z
	X\Y	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	4	8	12	16	20	24	28	32	36	40	44	48
b	1	4	7	11	15	19	23	27	31	35	39	43	47	51
a	2	8	4	8	12	16	20	23	27	31	35	39	43	47
n	3	12	8	12	8	12	16	20	24	28	32	36	40	44
a	4	16	12	15	12	15	19	16	20	24	28	32	36	40
n	5	20	16	19	15	19	22	20	23	27	31	35	39	43
a	6	24	20	23	19	22	26	22	26	30	34	38	35	39

Step 3: Compute the value of an optimal solution

SA(i, j) = Sequence Alignment Problem considering prefix strings $x_1 \ldots x_i$ and $y_1 \ldots y_j$

M[i,j] = the value of an optimal solution to SA(i, j)

$$
M[i,j] = \begin{cases} j\delta_{gap}, & \text{if } i = 0 \text{ (base case)} \\ i\delta_{gap}, & \text{if } j = 0 \text{ (base case)} \\ \min\{\delta_{gap}+M[i-1,j], \delta_{gap}+M[i,j-1], \delta_{x_i,y_j}+M[i-1,j-1]\}, & \text{otherwise} \end{cases}
$$

```
Input: X[1…m], Y[1…n], δgap, δpq for all p, q in alphabet
SA(m,n):
    for i = 0 to m
        M[i, 0] <- jδgap //|Y|=0, cost=|X|*gap penalty
    for j = 1 to n
        M[0, j] <- iδgap //|X|=0, cost=|Y|*gap penalty
    for i = 1 to m
        for j = 1 to n
            M[i, j] <- min(δgap+M[i-1,j],δgap+M[i,j-1],δxi,yj+M[i-1,j-1])
    return M[m,n]
```

Running time = $\Theta(mn)$

Step 4: Construct an optimal solution using backtracking

- SA(i, j) = Sequence Alignment Problem considering prefix strings $x_1 \ldots x_i$ and $y_1 \ldots y_j$
- **M[i,j] = the value of an optimal solution to SA(i, j)**

$$M[i,j] = \begin{cases} j\delta_{gap}, & \text{if } i = 0 \text{ (base case)} \\ i\delta_{gap}, & \text{if } j = 0 \text{ (base case)} \\ \min\{\delta_{gap}+M[i-1,j], \delta_{gap}+M[i,j-1], \delta_{xi,yj}+M[i-1,j-1]\}, & \text{otherwise} \end{cases}$$

Ex. $\delta_{gap}=4$, $\delta_{pq}=7$ if p!=q:

			a	e	n	i	q	a	d	i	k	j	a	z
	X\Y	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	4	8	12	16	20	24	28	32	36	40	44	48
b	1	4	7	11	15	19	23	27	31	35	39	43	47	51
a	2	8	4	8	12	16	20	23	27	31	35	39	43	47
n	3	12	8	12	8	12	16	20	24	28	32	36	40	44
a	4	16	12	15	12	15	19	16	20	24	28	32	36	40
n	5	20	16	19	15	19	22	20	23	27	31	35	39	43
a	6	24	20	23	19	22	26	22	26	30	34	38	35	39

Step 4: Construct an optimal solution using backtracking

```
Input: M[0…m, 0…n]
//return alignment
Find-Solution(m,n):
    if m = 0 or n = 0
        return {}
    //M[m,n] <- min{δgap+M[m-1,n],δgap+M[m,n-1],δxi,yj+M[m-1,n-1])
    if m[m,n] = δgap+M[m-1,n] //往上走
        return Find-Solution(m-1,n)
    if m[m,n] = δgap+M[m,n-1] //往左走
        return Find-Solution(m,n-1)
    return {(m, n)} ∪ Find-Solution(m-1,n-1)//左上
```

Running time = $\Theta(m+n)$

Sequence alignment 的應用

Unix `diff`

- X and Y are files
- Each elements of X and Y are lines of text

Computational biology （計算生物學）

- X and Y are DNA or protein sequences
  - DNA: {A, C, T, G}
  - Protein: {gly, trp, cys, ...}
- In practice, DP might still be too expensive even with optimizations
- Heuristics are often used to approximate the DP solution

Space-efficient solution

What is the storage overhead? $\Theta(mn)$

Can we reduce it to linear?

- How about keeping only the most recent two rows?

X\Y	0	1	2	3	…	n
i-1						
i					M[i,j]	

Can compute optimal value but cannot reconstruct the solution ☹

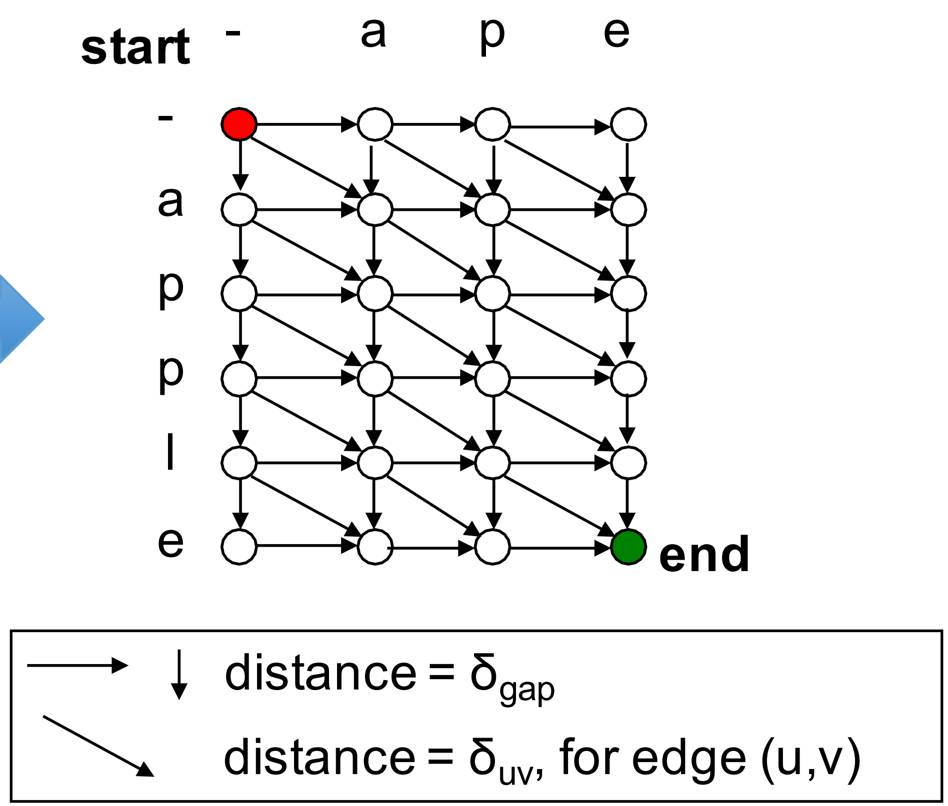
Let's see how to design a space-efficient sequence alignment algorithm using $O(m+n)$ space

- 組合技：dynamic programming + divide and conquer!

Viewing as a graph

Find minimal cost alignment => find shortest path

		-	a	p	e
	X\Y	0	1	2	3
-	0				
a	1				
p	2				
p	3				
l	4				
e	5				



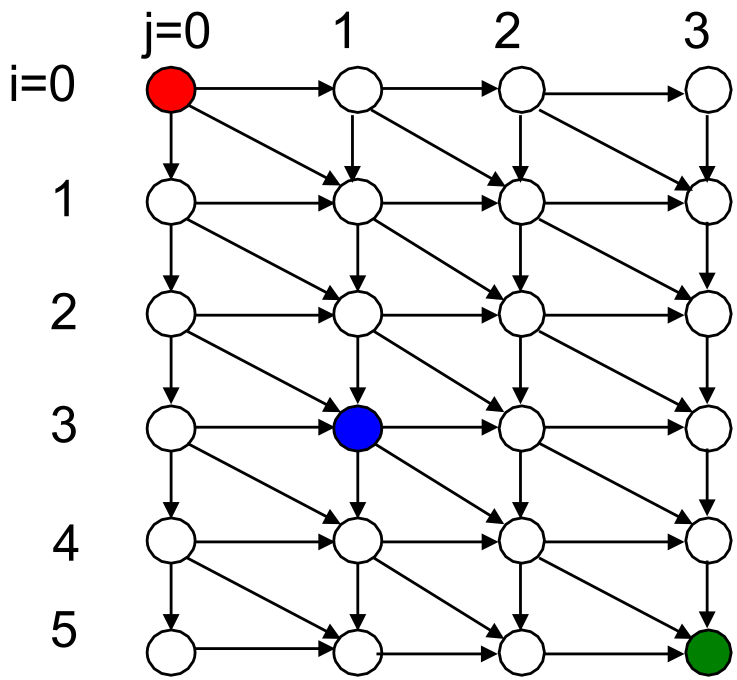
Shortest path in graph

Each edge has a length

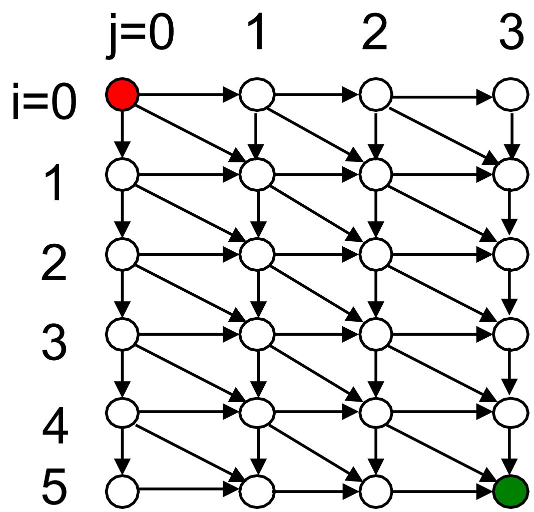
F[i, j] = length of the shortest path from (0, 0) to (i, j)

G[i, j] = length of the shortest path from (i, j) to (m, n)

=> F[m, n] = G[0, 0]



E.g.,
F[3,1] = shortest distance from Red to Blue
G[3,1] = shortest distance from Blue to Green



Formulations

Each edge has a length

F[i, j] = length of the shortest path from (0, 0) to (i, j)

G[i, j] = length of the shortest path from (i, j) to (m, n)

=> F[m, n] = G[0, 0]

Forward formulation

$$F[i,j]=\begin{cases} j\delta_{gap}, & \text{if } i = 0 \text{ (base case)} \\ i\delta_{gap}, & \text{if } j = 0 \text{ (base case)} \\ \min\{\delta_{gap}+F[i-1,j],\delta_{gap}+F[i,j-1],\delta_{xi,yj}+F[i-1,j-1]\}, & \text{otherwise} \end{cases}$$

Backward formulation

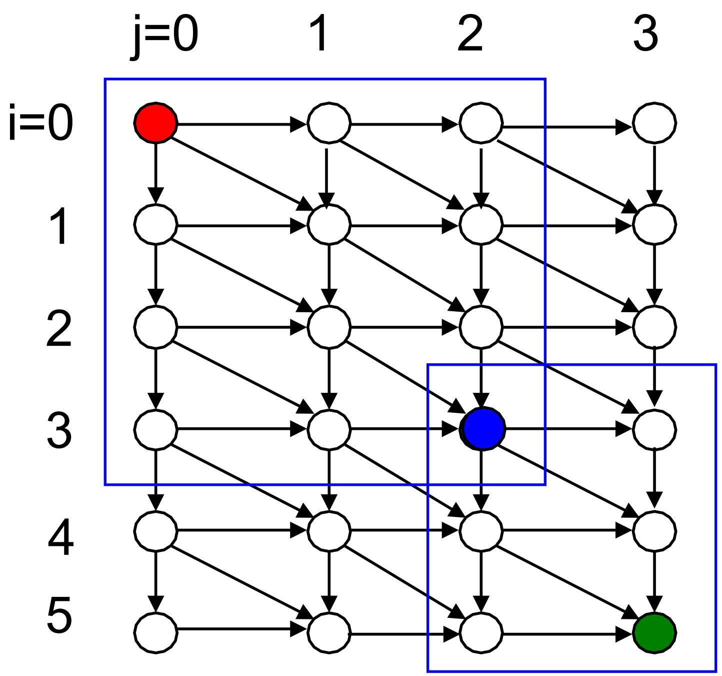
$$G[i,j]=\begin{cases} (n-j)\delta_{gap}, & \text{if } i = m \text{ (base case)} \\ (m-i)\delta_{gap}, & \text{if } j = n \text{ (base case)} \\ \min\{\delta_{gap}+G[i+1,j],\delta_{gap}+G[i,j+1],\delta_{xi,yj}+G[i+1,j+1]\}, & \text{otherwise} \end{cases}$$

Shortest path via a node

F[i, j] = length of the shortest path from (0, 0) to (i, j)
G[i, j] = length of the shortest path from (i, j) to (m, n)

Observation 1: The length of the shortest path between (0,0) and (m,n) that passes through (i,j) is F[i,j] + G[i,j]

- E.g., the length of the shortest path between Red and Green that passes through Blue is F[3,2] + G[3,2]

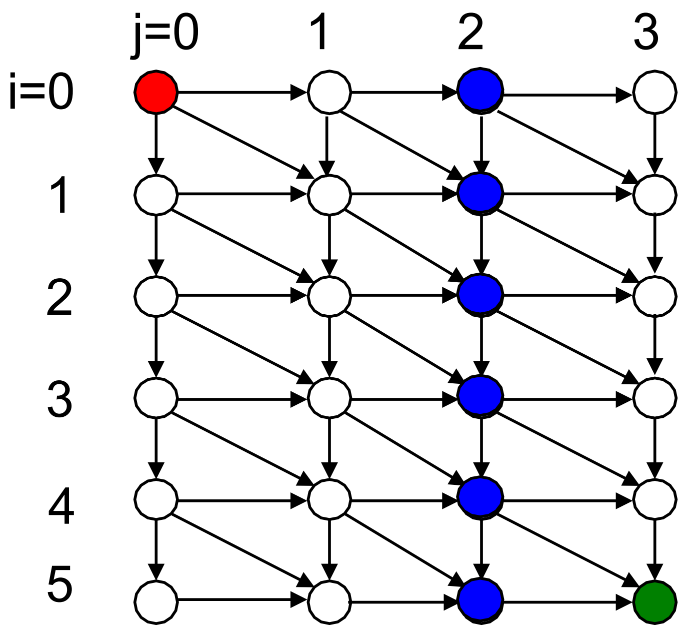


Shortest path must go across a vertical cut

F[i, j] = length of the shortest path from (0, 0) to (i, j)
G[i, j] = length of the shortest path from (i, j) to (m, n)

Observation 2: for any v in {0,…,n}, there exists a u such that the shortest path between (0,0) and (m,n) goes through (u, v)

- E.g., pick v = 2, the shortest path must pass through at least one of nodes (0,2), (1,2), (2,2), (3,2), (4,2), (5,2)

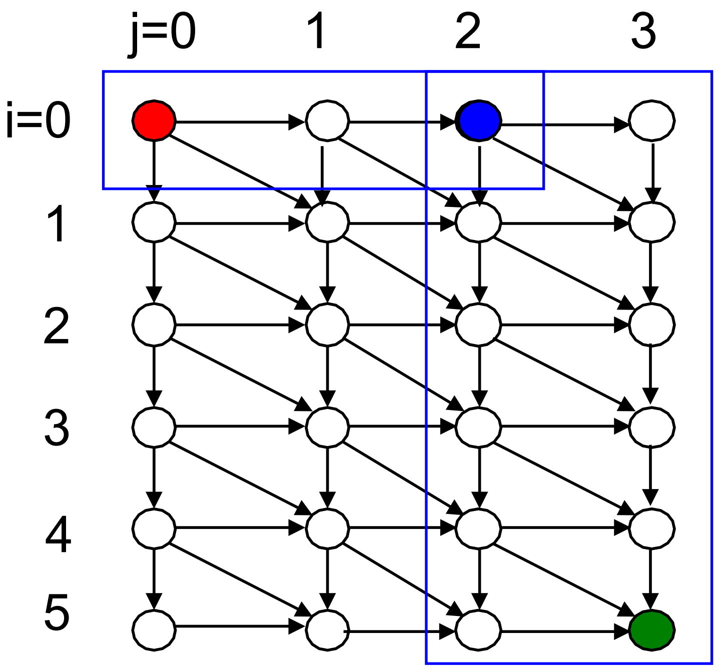


Shortest path in graph

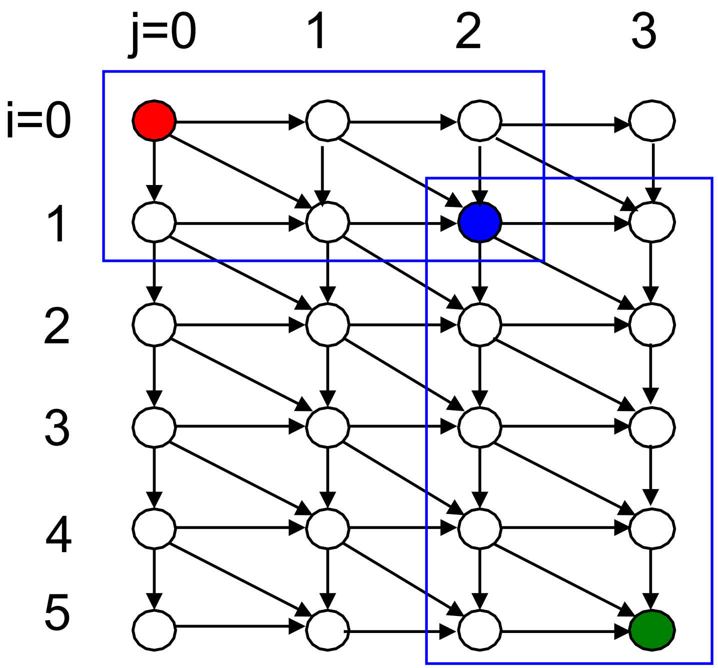
F[i, j] = length of the shortest path from (0, 0) to (i, j)
G[i, j] = length of the shortest path from (i, j) to (m, n)

Observations 1 & 2 imply

- F[m,n] = min{F[0,v] + G[0,v], F[1,v] + G[1,v], …, F[m,v] + G[m,v]}
- A shortest path goes through (u,v) when u $\in$ arg min$_i${F[i,v] + G[i,v]}



F[0,2] + G[0,2]



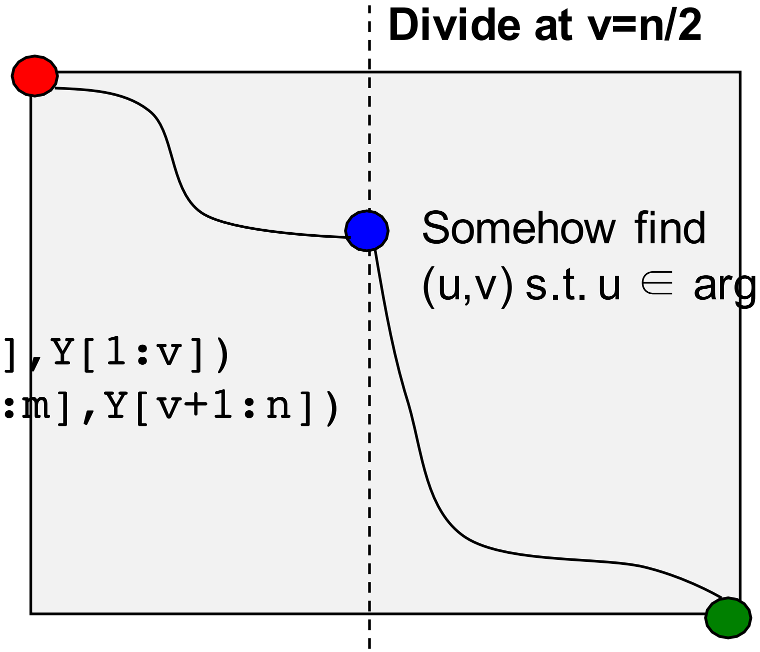
F[1,2] + G[1,2]

……

Divide and Conquer

- F[i, j] = length of the shortest path from (0, 0) to (i, j)
- G[i, j] = length of the shortest path from (i, j) to (m, n)
- F[m,n] = min{F[0,v] + G[0,v], F[1,v] + G[1,v], …, F[m,v] + G[m,v]}
- A shortest path goes through (u,v) when u ∈ arg $\min_i${F[i,v] + G[i,v]}
- `OriginalAlignment(X,Y)`: original algorithm with O(mn) space
- `SpaceEfficientAlignment(X,Y)`: algorithm with O(n) space (returning optimal value only)

Goal: `DCAlignment(X,Y)` to find optimal solution in O(m+n) space



Conquer

```
prefix=DCAlignment(X[1:u],Y[1:v])
suffix=DCAlignment(X[u+1:m],Y[v+1:n])
```

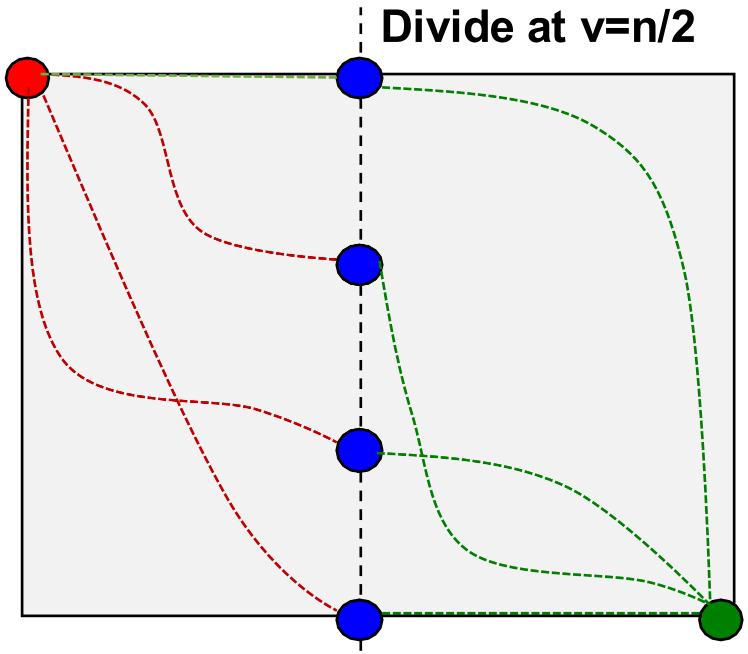
Combine

return prefix ∪ (u,v) ∪ suffix

Divide and Conquer

- $F[i, j]$ = length of the shortest path from (0, 0) to (i, j)
- $G[i, j]$ = length of the shortest path from (i, j) to (m, n)
- $F[m,n] = \min\{F[0,v] + G[0,v], F[1,v] + G[1,v], \ldots, F[m,v] + G[m,v]\}$
- A shortest path goes through (u,v) when $u \in \arg\min_i\{F[i,v] + G[i,v]\}$
- `OriginalAlignment(X,Y)`: original algorithm with O(mn) space
- `SpaceEfficientAlignment(X,Y)`: algorithm with O(n) space (returning optimal value only)

Find (u,v) s.t. $u \in \arg\min_i\{F[i,v] + G[i,v]\}$



Call SpaceEfficientAlignment(X,Y[1:v]) to find F[0,v], F[1,v], …,F[m,v]

Call BackwardSpaceEfficientAlignment(X,Y[v+1:n]) to find G[0,v], G[1,v], …,G[m,v]

Divide and Conquer

- F[i, j] = length of the shortest path from (0, 0) to (i, j)
- G[i, j] = length of the shortest path from (i, j) to (m, n)
- F[m,n] = min{F[0,v] + G[0,v], F[1,v] + G[1,v], …, F[m,v] + G[m,v]}
- A shortest path goes through (u,v) when u $\in$ arg $\min_i${F[i,v] + G[i,v]}
- `OriginalAlignment(X,Y)`: original algorithm with O(mn) space
- `SpaceEfficientAlignment(X,Y)`: algorithm with O(n) space (returning optimal value only)

Goal: DCAlignment(X,Y) to find optimal solution with O(m+n) space

Base case: if m≤2 or n≤2, call OriginalAlignment(X,Y)

Divide: divide vertically at v= n/2, find u $\in$ arg $\min_i${F[i,v] + G[i,v]}

- Call SpaceEfficientAlignment(X,Y[1:v]) to find F[0,v], F[1,v], …,F[m,v]
- Call BackwardSpaceEfficientAlignment(X,Y[v+1:n]) to find G[0,v], G[1,v], …,G[m,v]
- Let u be the index minimizing F[u,v] + G[u,v]

Conquer:

- prefix = DCAlignment(X[1:u],Y[1:v])
- suffix = DCAlignment(X[u+1:m], Y[v+1:n])

Combine: prefix $\cup$ (u,v) $\cup$ suffix

Analysis

Let $T(i,j)$ denote the maximum running time of the algorithm on strings of length i and j

Goal: DCAlignment(X,Y) to find optimal solution with $O(m+n)$ space

Base case: if $m \leq 2$ or $n \leq 2$, call OriginalAlignment(X,Y)

Time = $O(m)$ or $O(n)$

Divide: divide vertically at $v= n/2$, find $u \in \arg\min_i\{F[i,v] + G[i,v]\}$

- Call SpaceEfficientAlignment(X,Y[1:v]) to find F[0, v], F[1, v], …,F[m, v]
- Call BackwardSpaceEfficientAlignment(X,Y[v+1:n]) to find G[0,v], G[1,v], …,G[m,v]
- Let u be the index minimizing $F[u,v] + G[u,v]$

Time = $O(mn)$

Conquer:

- prefix = DCAlignment(X[1:u],Y[1:v])
- suffix = DCAlignment(X[u+1:m], Y[v+1:n])

$T(u, n/2) + T(m-u, n/2)$

Combine: prefix $\cup$ (u,v) $\cup$ suffix

Running time analysis

Let T(i,j) denote the maximum running time of the algorithm on strings of length i and j

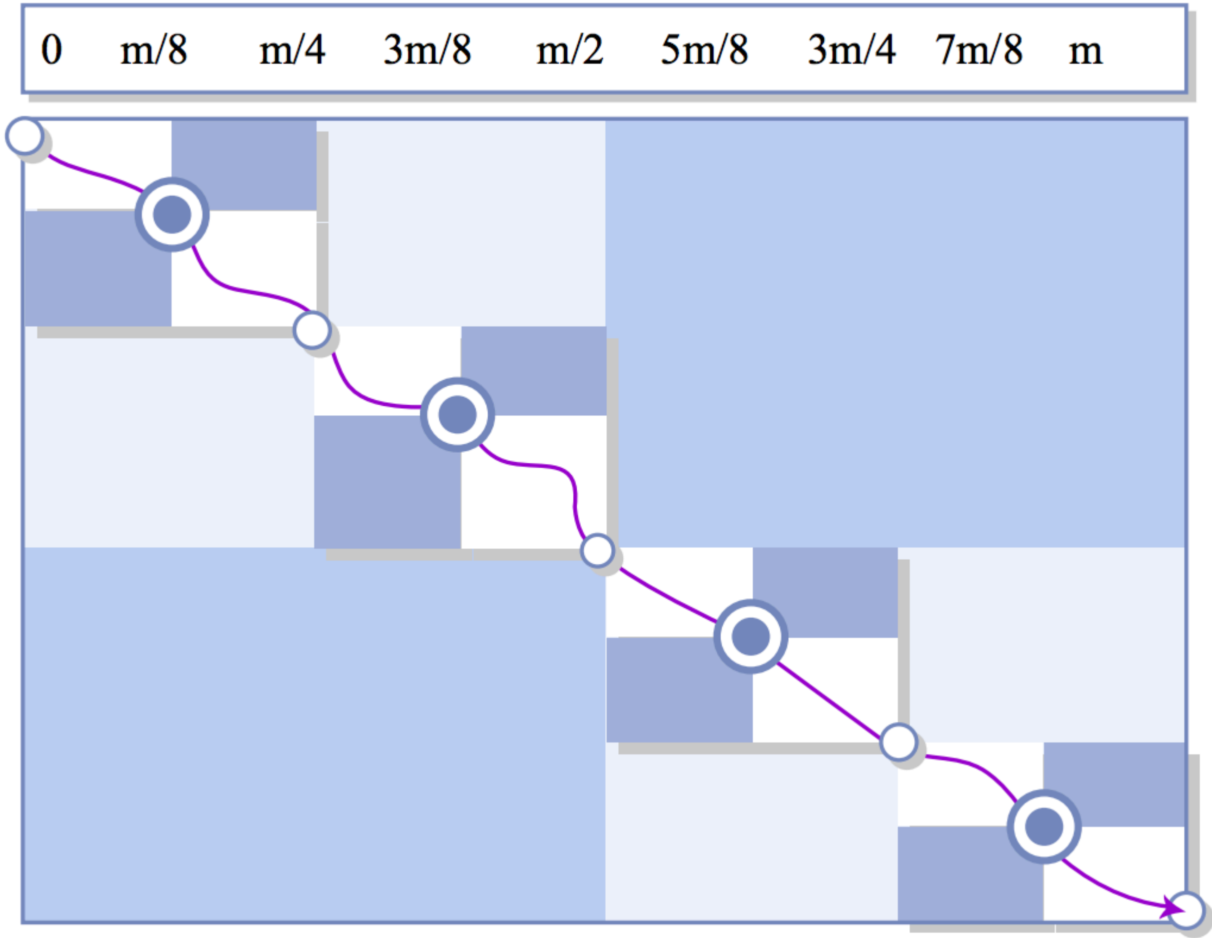
$$T(m,n) \le cmn + T(u, n/2) + T(m-u, n/2)$$

$$T(m,2) \le cm$$

$$T(2,n) \le cn$$

Prove that T(m,n) = O(mn)

Running time analysis



https://ocw.mit.edu/courses/mathematics/18-417-introduction-to-computational-molecular-biology-fall-2004/lecture-notes/lecture_07.pdf

Knapsack Problem (背包問題)

Textbook Exercise 16.2-2

Chapter 6.4 in Algorithm Design by Kleinberg & Tardos

幸運兒的煩惱

喵貓抽中航空公司大獎，可以在機場商店免費拿商品

商品總重不能超過5公斤(隨身行李限重)

要拿哪些商品才能讓喵貓最賺？

- 每項物品最多拿一個 **B+D+E => 4200g, $11,400**
- 每項物品可以拿多個 **D*1+E*5 => 5000g, $14,500**
- 每一類（食品、電器、藥妝）物品最多拿一個 **B+D+E => 4200g, $11,400**
- 背包空間有限只能裝共7單位體積的物品 **D+E => 3800g, $10,900**

A

1400g, $1000

體積：2單位

B

400g, $500

1單位

C

2000g, $700

4單位

D

3500g, $10000

6單位

E

300g, $900

1單位

Knapsack Problem

Given n objects and a "knapsack"

Object i weighs $w_i > 0$ and has value $v_i > 0$

Knapsack has capacity of W

最後再回來討論非整數的影響

W and w_is are non-negative integers

Goal: fill knapsack so as to maximize total value

In the example on the previous slide, W = 5000, and

i	Weight (w_i)	Value (v_i)
1	1400	1000
2	400	500
3	2000	700
4	3500	10000
5	300	900

Variants of knapsack problem

Goal: fill knapsack so as to maximize total value

Each variant considers different constraints

0/1 knapsack problem
- 每項物品只能拿一個

Unbounded knapsack problem
- 每項物品可以拿多個

Multiple-choice knapsack problem
- 每一類物品最多拿一個

Multidimensional knapsack problem
- 背包空間有限

…

Step 1: Characterize an optimal solution

0/1 Knapsack Problem: Fill a knapsack of capacity W so as to maximize total value, each of the n objects is selected **at most once**

- w_i = weight of object i, v_i = value of object i (w_i>0, v_i>0)

In this step, we need to answer two questions:

Step1-Q1: What can be the subproblems?

Step1-Q2: Does it exhibit optimal substructure?

- Can an optimal solution be represented by the optimal solutions to the subproblems?
- If we cannot find optimal substructure, either we have to go back to Step1-Q1 or there is no DP solution to this problem.

Step 1: Characterize an optimal solution

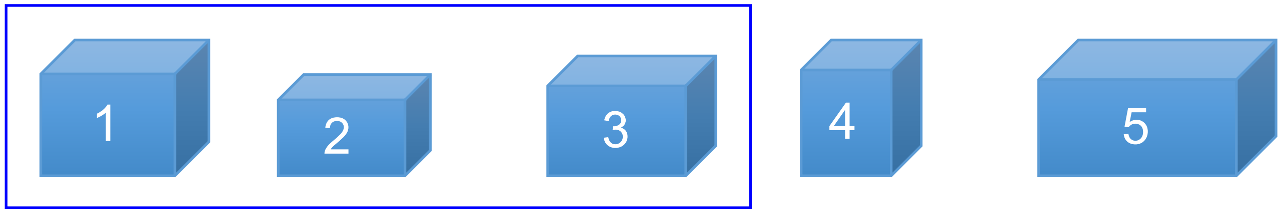
0/1 Knapsack Problem: Fill a knapsack of capacity W so as to maximize total value, each of the n objects is selected **at most once**

- w_i = weight of object i, v_i = value of object i (w_i>0, v_i>0)

An attempt to choosing subproblems

ZOKP(i): Zero/One Knapsack Problem using objects 1 to i

- Subproblems: ZOKP(1), ZOKP(2), …, ZOKP(n-1)



ZOKP(3): 只考慮前三個物品時，所能達到的最大總金額

Can we represent ZOKP(i) using solutions to “smaller” subproblems?

Step 1: Characterize an optimal solution

<u>0/1 Knapsack Problem:</u> Fill a knapsack of capacity W so as to maximize total value, each of the n objects is selected **at most once**
- w_i = weight of object i, v_i = value of object i (w_i>0, v_i>0)

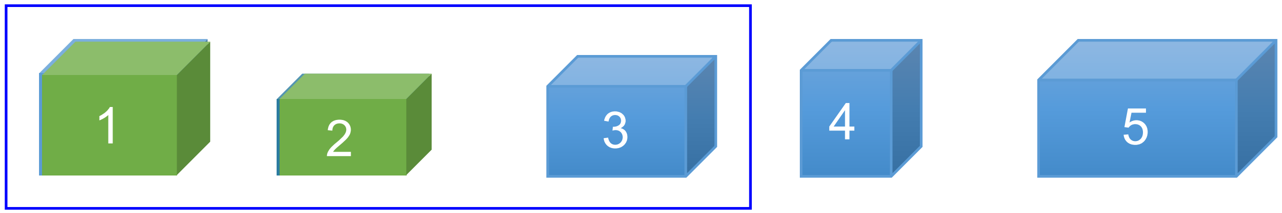
An attempt to choosing subproblems

ZOKP(i): 0/1 Knapsack Problem using objects 1 to i

Suppose OPT is an optimal solution to ZOKP(i)

<u>Case 1: object i not in OPT to ZOKP(i)</u>

- ZOKP(i) 的最佳解也是 ZOKP(i-1)的最佳解 ☺



ZOKP(3): 只考慮前三個物品時，所能達到的最大總金額
假設object 1, 2是ZOKP(3)的一個最佳解 (object 3不在最佳解裡)
那麼object 1, 2也是ZOKP(2)的最佳解

Step 1: Characterize an optimal solution

0/1 Knapsack Problem: Fill a knapsack of capacity W so as to maximize total value, each of the n objects is selected **at most once**

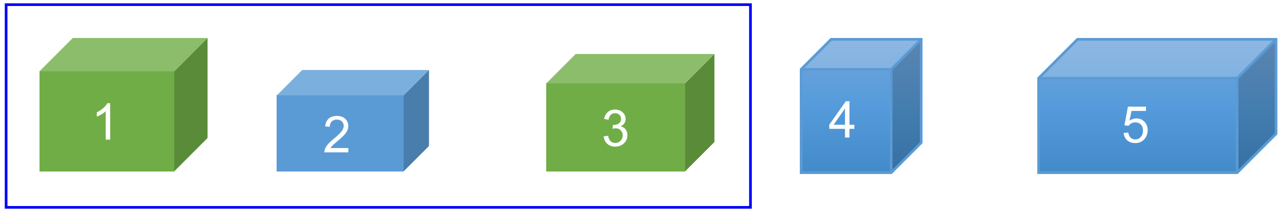
- w_i = weight of object i, v_i = value of object i (w_i>0, v_i>0)

An attempt to choosing subproblems

ZOKP(i): 0/1 Knapsack Problem using objects 1 to i

Case 2: object i in OPT to ZOKP(i)

- ZOKP(i)解可以用小問題的最佳解來表示嗎？NO!
- 知道小問題的最佳解沒幫助，因為不知道背包是否還放得下object i ☹



ZOKP(3): 只考慮前三個物品時，所能達到的最大總金額
假設object 1, 3是ZOKP(3)的一個最佳解…
ZOKP(3)跟ZOKP(2) or ZOKP(1)有關聯嗎？

定義subproblems時，需要把背包的重量限制納入考量

Step 1: Characterize an optimal solution

<u>0/1 Knapsack Problem:</u> Fill a knapsack of capacity W so as to maximize total value, each of the n objects is selected **at most once**
- w_i = weight of object i, v_i = value of object i (w_i>0, v_i>0)

Adding a new variable for weights

ZOKP(i, w) = 0/1 Knapsack Problem **with weight ≤ w** using objects 1 to i

- E.g., ZOKP(3, 10) = 只考慮前三個物品，且限重為10的背包問題

Can we represent ZOKP(i,w) using solutions to "smaller" subproblems?

Step 1: Characterize an optimal solution

0/1 Knapsack Problem: Fill a knapsack of capacity W so as to maximize total value, each of the n objects is selected **at most once**

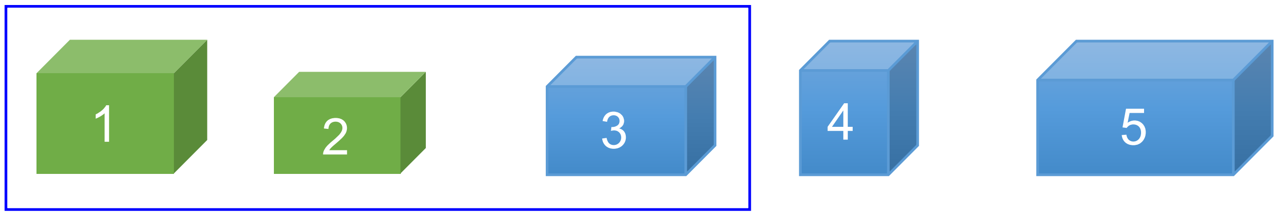
- w_i = weight of object i, v_i = value of object i (w_i>0, v_i>0)

Adding a new variable for weights

ZOKP(i, w) = 0/1 Knapsack Problem **with weight ≤ w** using objects 1 to i

Case 1: object i not in OPT to ZOKP(i, w)

- ZOKP(i, w) 的最佳解也是 ZOKP(i-1, w)的最佳解 ☺



ZOKP(3,10): 只考慮前三個物品，且限重為10時，所能達到的最大總金額
假設object 1, 2是ZOKP(3,10)的一個最佳解 (object 3不在最佳解裡)
那麼object 1, 2也是ZOKP(2,10)的最佳解

Proof of optimal substructure

- **Optimal substructure**: an optimal solution can be constructed from optimal solutions to subproblems
  - Proof by contradiction (specifically, a “cut-and-paste” argument)

Proof of case 1: when object i not in OPT to ZOKP(i, w)

Goal: 證明ZOKP(i, w) 的最佳解也是 ZOKP(i-1, w)的最佳解

- Suppose OPT is optimal to ZOKP(i, w) but not optimal to ZOKP(i-1, w)
- => there exist an optimal solution OPT’ to ZOKP(i-1, w) such that the value of OPT’ is higher than it of OPT
- => OPT’ is a better solution to ZOKP(i, w) than OPT
- => Contradiction!

Step 1: Characterize an optimal solution

0/1 Knapsack Problem: Fill a knapsack of capacity W so as to maximize total value, each of the n objects is selected **at most once**

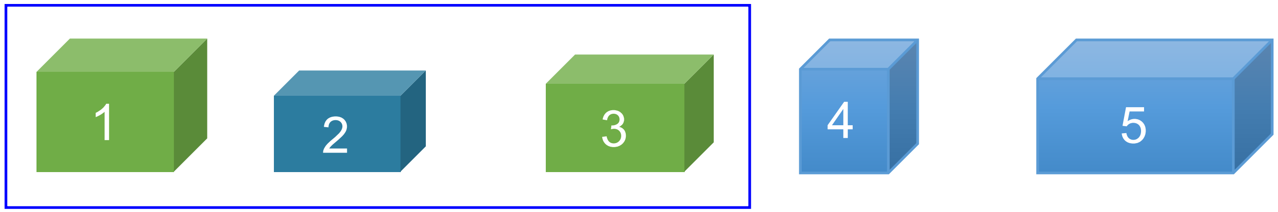
- w_i = weight of object i, v_i = value of object i (w_i>0, v_i>0)

Adding a new variable for weights

ZOKP(i, w) = 0/1 Knapsack Problem **with weight ≤ w** using objects 1 to i

Case 2: object i in OPT to ZOKP(i, w)

- 把object i拿走，背包裡剩下的物品是ZOKP(i-1, w-w_i)的一組最佳解 ☺



ZOKP(3,10): 只考慮前三個物品，且限重為10時，所能達到的最大總金額
假設object 1, 3是ZOKP(3,10)的一個最佳解
那麼object 1該是ZOKP(2,10-w_3)的最佳解

Proof of optimal substructure

- **Optimal substructure**: an optimal solution can be constructed from optimal solutions to subproblems
  - Proof by contradiction (specifically, a “cut-and-paste” argument)

Proof of case 2: when object i in OPT to ZOKP(i, w)

Goal: 證明把object i拿走後，背包裡剩下的物品是 ZOKP(i-1, w-w_i)的一組最佳解

  - Suppose OPT\{i} is not optimal to ZOKP(i-1, w-w_i)
  - => there exist an optimal solution OPT’ to ZOKP(i-1, w-w_i) such that the value of OPT’ is higher than it of OPT\{i}
  - => OPT’ ∪ {i} is a better solution to ZOKP(i, w) than OPT
  - => Contradiction!

DP illustration: 0/1 knapsack problem

加進第i個物品

考慮前i-1個物品
限重為w- w_i

Possible choice 1 + Optimal solution to subproblem

Optimal solution = Max

考慮前i個物品
限重為w

Possible choice 2 + Optimal solution to subproblem

沒拿第i個物品

考慮前i-1個物品
限重為w

Step 2: Recursively define the value of an optimal solution

ZOKP(i, w) = 0/1 knapsack with weight ≤ w using objects 1 to i

Case 1: object i not in OPT to ZOKP(i, w)

- ZOKP(i, w) 的最佳解也是 ZOKP(i-1, w)的最佳解

$\boxed{M[i,w]=M[i-1,w]}$

Case 2: object i in OPT to ZOKP(i, w)

- 把object i拿走，背包裡剩下的物品是ZOKP(i-1, w-w_i)的一組最佳解

$\boxed{M[i,w]=v_i+M[i-1,w-w_i]}$

M[i,w] = the value of an optimal solution to ZOKP(i, w)

用遞迴表示最佳解的值：

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

Step 3: Compute value of an optimal solution

Let's use the bottom-up approach to solve an example

Example:
capacity W = 5

Object i	Weight (w_i)	Value (v_i)
1	1	4
2	2	9
3	4	20

- Fill out table M, M[i,w] = value of an optimal solution to ZOKP(i, w)

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

i \ w	0	1	2	3	4	W=5
0, {}						
1, {1}			$M[i-1,w-w_i]$		$M[i-1,w]$	
2, {1,2}					$M[i, w]$	
3, {1,2,3}						

Our goal

Step 3: Compute value of an optimal solution

Let's use the bottom-up approach to solve an example

Example:
capacity W = 5

Object i	Weight (w_i)	Value (v_i)
1	1	4
2	2	9
3	4	20

- Fill out table M, M[i,w] = value of an optimal solution to ZOKP(i, w)

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

i \ w	0	1	2	3	4	W=5
0, {}	0	0	0	0	0	0
1, {1}	0	4	4	4	4	4
2, {1,2}	0	4	9	13	13	13
3, {1,2,3}	0	4	9	13	20	$\boxed{24}$

Our goal

Step 3: Compute the value of an optimal solution

M[i,w] = value of an optimal solution to ZOKP(i, w)

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

```
Input: w[1..n], v[1..n]
ZOKP(n,W):
    for w = 0 to W //initialize array M[]
        M[0, w] <- 0
    for i = 1 to n
        for w = 0 to W
            if(wi > w)
                M[i, w] <- M[i-1, w]
            else
                M[i, w] <- max(M[i-1, w], vi + M[i-1,w-wi])
    Return M[n,W]
```

Running time = $\Theta(nW)$

Step 4: Construct an optimal solution

Make a second pass for **backtracking** and find the solution

Example:
capacity W = 5

Object i	Weight (w_i)	Value (v_i)
1	1	4
2	2	9
3	4	20

- Table M, M[i,w] = value of an optimal solution to ZOKP(i, w)
- $M[i,w] \begin{cases} = 0, \text{ if } i=0 \\ = M[i-1,w], \text{ if } w_i > w \\ = \max\{M[i-1,w],\ v_i + M[i-1,w-w_i]\}, \text{ otherwise} \end{cases}$

i \ w	0	1	2	3	4	W=5
0, {}	0	0	0	0	0	0
1, {1}	0	4	4	4	4	4
2, {1,2}	0	4	9	13	13	13
3, {1,2,3}	0	4	9	13	20	24

Our goal

Step 4: Construct an optimal solution

Running time = Θ(nW)

```
Input: w[1..n], v[1..n]
ZOKP(n,W): //find optimal value
    for w = 0 to W //initialize array M[]
        M[0, w] <- 0
    for i = 1 to n
        for w = 0 to W
            if(wi > w)
                M[i, w] <- M[i-1, w]
            else
                M[i, w] <- max(M[i-1, w], vi + M[i-1,w-wi])
    Return M[n,W]
```

Running time = Θ(n)

```
Input: w[1…n], M[0…n, 0…W]
Find-Solution(n,W): //find optimal solution
    S <- {}
    w <- W
    for i = n to 1
        if (M[i, w] > M[i–1, w])
            w <- w – w[i]
            S <- S ∪ {i}
    return S
```

Pseudo-polynomial time

Running time = $\Theta(nW)$

- n = # of objects
- W = knapsack's capacity, W is a non-negative integers

Running time is pseudo-polynomial, not polynomial, in input size

- Pseudo-polynomial time: "if its running time is **polynomial in the numeric value** of the input, but is **exponential in the length of the input** – the number of bits required to represent it."

The size of the representation of W = lgW

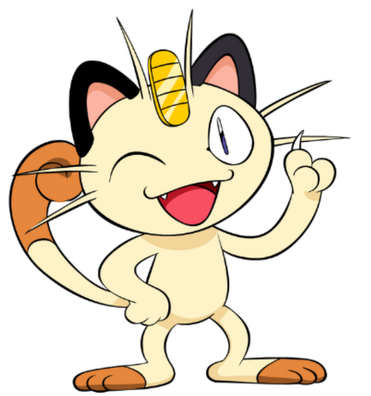
Variants of knapsack problem

Goal: fill knapsack so as to maximize total value

Each variant considers different constraints

1. 0/1 knapsack problem
   - 每項物品只能拿一個
2. Unbounded knapsack problem
   - 每項物品可以拿多個
3. Multiple-choice knapsack problem
   - 每一類物品最多拿一個
4. Multidimensional knapsack problem
   - 背包空間有限

…



Step 1: Characterize an optimal solution

Unbounded Knapsack Problem: Fill a knapsack of capacity W so as to maximize total value, each of the n objects has **unlimited supplies**

- w_i = weight of object i, v_i = value of object i (w_i>0, v_i>0)

An attempt to choosing subproblems

UKP(i, w) = Unbounded Knapsack Problem with weight ≤ w using objects 1 to i

What would be the running time?

- Compute a n * W table
- Each cell requires comparing n choices

0/1 knapsack problem	Unbounded knapsack problem
每種物品只有一個	每種物品有無限多個
一系列的binary choices (是否要放object n, 是否要放object n-1, …)	一系列的n choices (object1~n要放哪一個, object1~n要放哪一個, …)
$\Theta(nW)$	$\Theta(n^2W)$?

Step 1: Characterize an optimal solution

<u>Unbounded Knapsack Problem:</u> Fill a knapsack of capacity W so as to maximize total value, each of the n objects has **unlimited supplies**
- w_i = weight of object i, v_i = value of object i (w_i>0, v_i>0)

Consider weight only

UKP(w) = Unbounded Knapsack Problem with weight ≤ w

Suppose we know one optimal solution OPT to UKP(w)

Pick one object x in OPT, there are n possibilities:

- <u>Case 1: x = 1</u>
  - 拿走的是object 1，背包裡剩下的物品是UKP(w-w_1)的一組最佳解
- <u>Case 2: x = 2</u>
  - 拿走的是把object 2，背包裡剩下的物品是UKP(w-w_2)的一組最佳解
- …
- <u>Case n: x = n</u>
  - 把object n拿走，背包裡剩下的物品是UKP(w-w_n)的一組最佳解

DP illustration: Unbounded knapsack problem

加進第1個物品 限重為w - w_1

Possible choice 1 + Optimal solution to subproblem

加進第2個物品 限重為w - w_2

Optimal solution = Max Possible choice 1 + Optimal solution to subproblem

限重為w

… …

Possible choice n + Optimal solution to subproblem

加進第n個物品 限重為w - w_n

Proof of optimal substructure

Optimal substructure: an optimal solution can be constructed from optimal solutions to subproblems

- Proof by contradiction (specifically, a “cut-and-paste” argument)

Proof of case i: when object i in OPT to UKP(w)

Goal:證明把object i拿走，背包裡剩下的物品是UKP(w-w_i)的一組最佳解

- Suppose OPT\{i} is not optimal to UKP(w-w_i)
- => there exist an optimal solution OPT’ to UKP(w-w_i) such that the value of OPT’ is higher than it of OPT\{i}
- => OPT’ ∪ {i} is a better solution to UKP(w) than OPT
- => Contradiction!

Step 2: Recursively define the value of an optimal solution

UKP(w) = Unbounded Knapsack Problem with weight $\leq$ w

Suppose we know one optimal solution OPT to UKP(w)

Pick one object x in OPT, there are n possibilities:

Case i: x = i

- 把object i拿走，背包裡剩下的物品是UKP(w-w_i)的一組最佳解

$$\boxed{M[w] = v_i + M[w-w_i]}$$

M[w] = the value of an optimal solution to UKP(w)

用遞迴表示最佳解的值：

$$M[w] = \begin{cases} 0, & \text{if } w = 0 \text{ or } w < w_i \text{ for all } i \\ \max_{i,\boxed{w \geq wi}}\{v_i + M[w-w_i]\}, & \text{otherwise} \end{cases}$$

只需考慮w≥w_i（背包裝得下）的情況

Step 3: Compute value of an optimal solution

Let's use the bottom-up approach to solve an example

Example:
capacity W = 5

Object i	Weight (w_i)	Value (v_i)
1	1	4
2	2	9
3	4	18

Fill out table M, M[w] = value of an optimal solution to UKP(w)

$$M[w] = \begin{cases} 0, \text{ if } w = 0 \text{ or } w < w_i \text{ for all } i \\ \max_{i, w \geq w_i}\{v_i + M[w-w_i]\}, \text{ otherwise} \end{cases}$$

w	0	1	2	3	4	W=5
M[w]		$M[w-w_i]$		$\rightarrow$ M[w]		

Our goal

Step 3: Compute value of an optimal solution

Let's use the bottom-up approach to solve an example

Example:
capacity W = 5

Object i	Weight (w_i)	Value (v_i)
1	1	4
2	2	9
3	4	17

Fill out table M, M[w] = value of an optimal solution to UKP(w)

$$M[w] = \begin{cases} 0, & \text{if } w = 0 \text{ or } w < w_i \text{ for all } i \\ \max_{i, w \geq w_i}\{v_i + M[w-w_i]\}, & \text{otherwise} \end{cases}$$

w	0	1	2	3	4	W=5
M[w]	0	4	9	13	18	22

Our goal

M[1] =max{4+0}
M[2]=max{4+4, 9+0}
M[3]=max{4+9, 9+4}
M[4]=max{4+13, 9+9, 17+0}
M[5]=max{4+18, 9+13, 17+4}

Step 3: Compute value of an optimal solution

M[w] = value of an optimal solution to UKP(w)

$$M[w] = \begin{cases} 0, & \text{if } w = 0 \text{ or } w < w_i \text{ for all } i \\ \max_{i,\, w \geq w_i}\{v_i + M[w-w_i]\}, & \text{otherwise} \end{cases}$$

```
Input: w[1…n], v[1…n]
UKP(W):
    for w = 1 to W
        M[w] <- 0 //initialize array M[]
    for w = 1 to W
        for i = 1 to n
            if(w ≥ wi)
                tmp <- vi + M[w-wi]
                M[w] <- max{M[w], tmp}
    Return M[W]
```

Running time = $\Theta(nW)$

Step 4: Construct an optimal solution

Make a second pass for **backtracking** and find the solution

Example:
capacity W = 5

Object i	Weight (w_i)	Value (v_i)
1	1	4
2	2	9
3	4	17

M[w] = value of an optimal solution to UKP(w)

$$M[w] = \begin{cases} 0, & \text{if } w = 0 \text{ or } w < w_i \text{ for all } i \\ \max_{i,\, w \geq w_i}\{v_i + M[w-w_i]\}, & \text{otherwise} \end{cases}$$

w	0	1	2	3	4	W=5
M[w]	0	4	9	13	18	$\boxed{22}$

Our goal

Check if object 1 is in OPT

Check if object 2 is in OPT\{1,2}

Check if object 1 is in OPT\{1}

Check if object 2 is in OPT\{1}

2 2 1

Step 4: Construct an optimal solution

Running time = Θ(nW)

```
Input: w[1…n], v[1…n]
UKP(W):
    for w = 1 to W
        M[w] <- 0 //initialize array M[]
    for w = 1 to W
        for i = 1 to n
            if(w ≥ wi)
                tmp <- vi + M[w-wi]
                M[w] <- max{M[w], tmp}
    Return M[W]
```

Running time = O(n+W)

```
Input: w[1…n], v[1…n], M[0…W]
Find-Solution(n,W)://find optimal solution
    for i = 1 to n
        C[i] <- 0 //C[i]=#of object i in solution
    w <- W
    while (w > 0 && i <= n)
        if(w ≥ wi && M[w] == (vi + M[w-wi]))
            w <- w-wi
            C[i]+=1
        else
            i++
    return C
```

Variants of knapsack problem

Goal: fill knapsack so as to maximize total value

Each variant considers different constraints

1. 0/1 knapsack problem
   - 每項物品只能拿一個
2. Unbounded knapsack problem
   - 每項物品可以拿多個
3. Multiple-choice knapsack problem
   - 每一類物品最多拿一個
4. Multidimensional knapsack problem
   - 背包空間有限

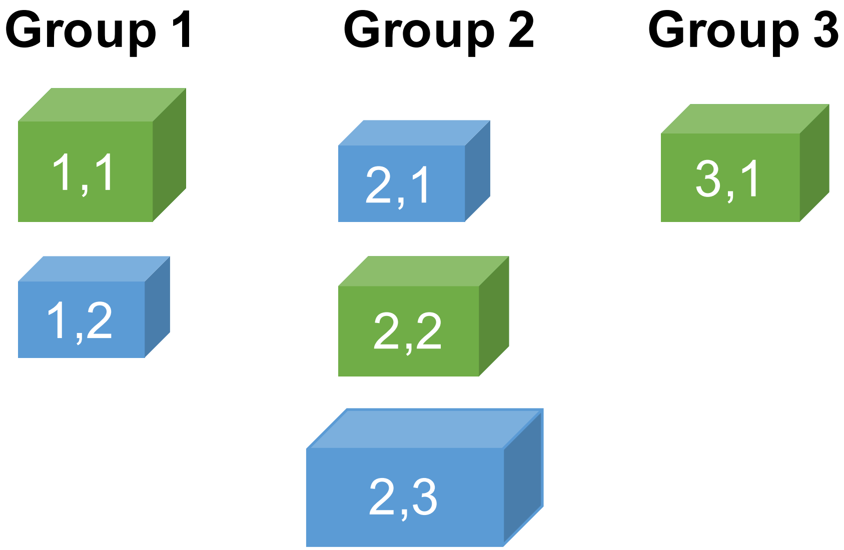
…

Multiple-choice knapsack problem

Fill a knapsack of capacity W so as to maximize total value

Select **at most one object from each group**

- $w_{i,j}$ = weight of j-th object in group i
- $v_{i,j}$ = value of j-th object in group i
- n_i = # of objects in group i
- n = total number of objects = $\Sigma_i n_i$
- G = # of groups



Knapsack capacity, W=6

Object i.j	Weight ($w_{i,j}$)	Value ($v_{i,j}$)
1,1	1	4
1,2	2	9
2,1	2	5
2,2	3	10
2,3	4	12
3,1	2	8

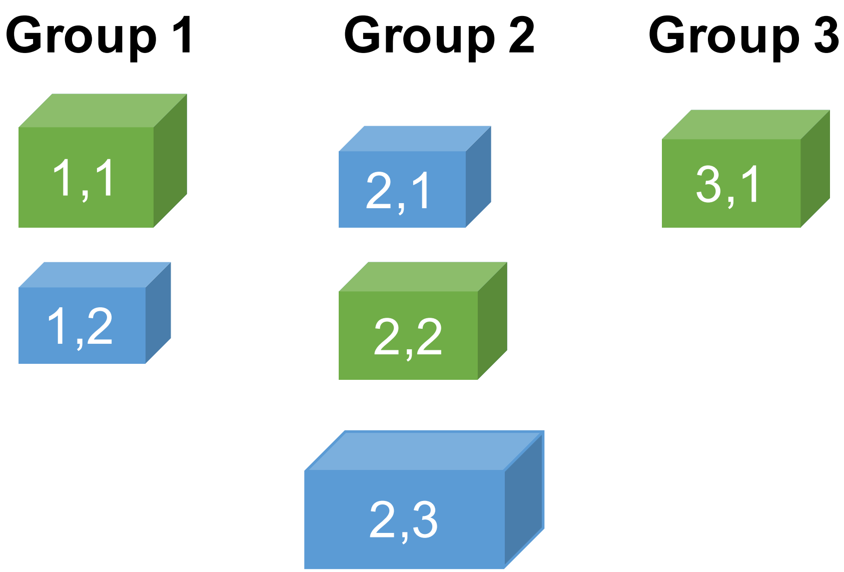
Multiple-choice knapsack problem

Fill a knapsack of capacity W so as to maximize total value

Select **at most one object from each group**

Q: What should the subproblems look like?

1. MCKP(w) = MCKP **with total weight ≤ w**
2. MCKP(i, w) = MCKP **with total weight ≤ w using group 1 to i**
3. MCKP(i, j, w) = MCKP **with total weight ≤ w using objects 1 to j in group 1 to i**



Knapsack capacity, W=6

Object i.j	Weight ($w_{i,j}$)	Value ($v_{i,j}$)
1,1	1	4
1,2	2	9
2,1	2	5
2,2	3	10
2,3	4	12
3,1	2	8

Step 1: Characterize an optimal solution

<u>Multiple-choice Knapsack Problem:</u> Fill a knapsack of capacity W so as to maximize total value; select **at most one object from each group**
- $W_{i,j}$ = weight of object (i,j), $v_{i,j}$ = value of object (i,j), n_i = #objects in group i, n = total number of objects = $\Sigma_i n_i$, G=#of groups

MCKP(i, w) = MCKP with total weight ≤ w **using group 1 to i**

Suppose OPT is an optimal solution to MCKP(i, w)

For group i, there are n_i+1 possibilities:
- <u>Case 0: OPT contains no object from group i</u>
  - OPT是MCKP(i-1,w)的一個最佳解
- <u>Case 1: OPT contains $obj_{i,1}$</u>
  - OPT移掉$obj_{i,1}$後，剩下的物品是MCKP(i-1, w-$w_{i,1}$) 的一個最佳解

  ……
- <u>Case n_i: OPT contains $obj_{i,ni}$</u>
  - OPT移掉$obj_{i,ni}$後，剩下的物品是MCKP(i-1, w-$w_{i,ni}$) 的一個最佳解

Step 2: Recursively define the value of an optimal solution

MCKP(i, w) = MCKP with total weight ≤ w using group 1 to i

- Case 0: OPT contains no object from group i
  - OPT是MCKP(i-1,w)的一個最佳解

M[i,w] = ?

Case j: OPT contains $obj_{i,j}$ (for $1 \le j \le n_i$)

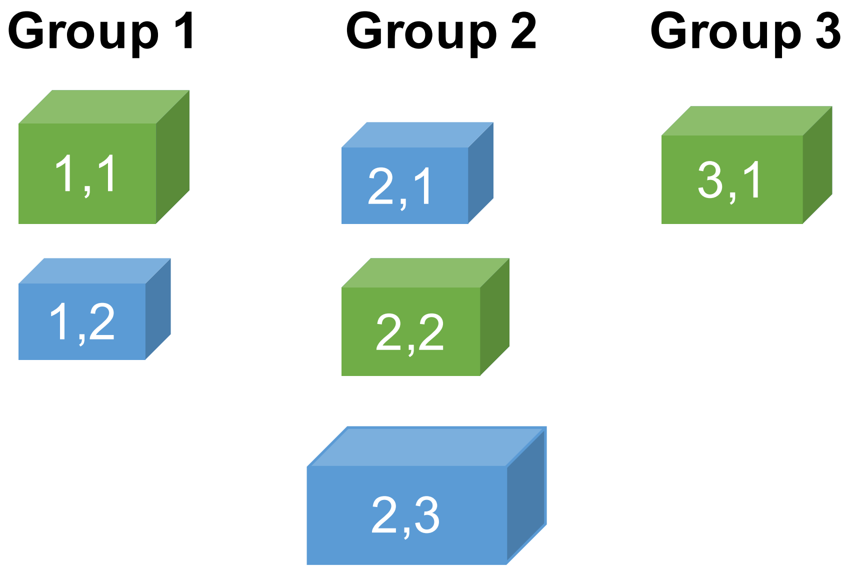
- OPT移掉$obj_{i,j}$後, 剩下的物品是MCKP(i-1, $w-w_{i,j}$) 的一個最佳解

M[i,w] = ? + M[i-1,w-$w_{i,j}$]

M[i,w] = the value of an optimal solution to MCKP(i, w)

用遞迴表示最佳解的值：

$$
M[i,w]=\begin{cases} 0, & \text{if } i = 0 \text{ (base case)} \\ M[i-1, w], & \text{if } w < w_{i,j} \text{ for all } j \\ \max\{M[i-1,w], \max\limits_{j, w \ge w_{i,j}}\{v_{i,j}+M[i-1,w-w_{i,j}]\}\}, & \text{otherwise} \end{cases}
$$



Knapsack capacity, W=6

Object i.j	Weight ($w_{i,j}$)	Value ($v_{i,j}$)
1,1	1	4
1,2	2	9
2,1	2	5
2,2	3	10
2,3	4	12
3,1	2	8

$$M[i,w] = \begin{cases} 0, \text{ if } i = 0 \text{ (base case)} \\ M[i-1, w], \text{ if } w < w_{i,j} \text{ for all } j \\ \max\{M[i-1,w], \max\limits_{j, w \geq w_{i,j}}\{v_{i,j} + M[i-1, w-w_{i,j}]\}\}, \text{ otherwise} \end{cases}$$

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	4	9	9	9	9	9
2	0	4	9	10	14	19	21
3	0	4	9	12	17	19	22

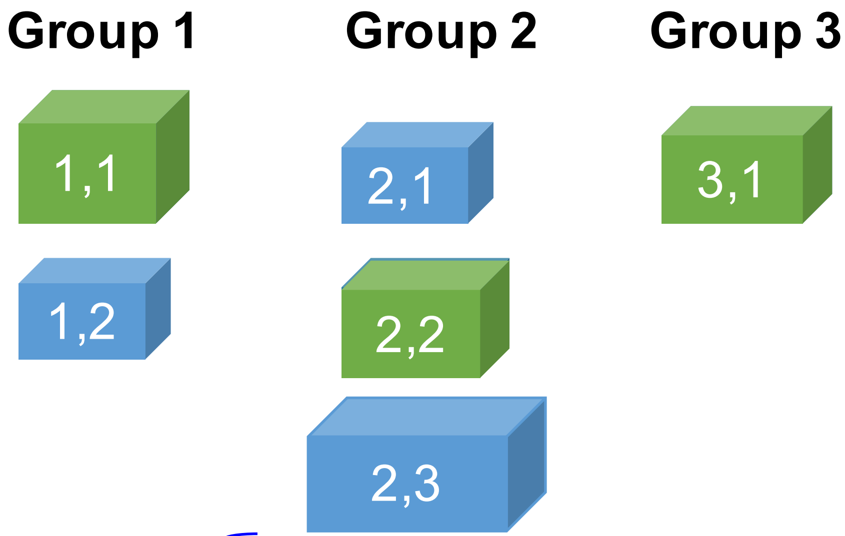
Step 3: Compute the value of an optimal solution

$$M[i,w] = \begin{cases} 0, & \text{if } i = 0 \text{ (base case)} \\ M[i-1, w], & \text{if } w < w_{i,j} \text{ for all } j \\ \max\{M[i-1,w], \max\limits_{j, w \geq w_{i,j}}\{v_{i,j} + M[i-1, w-w_{i,j}]\}\}, & \text{otherwise} \end{cases}$$

```
Input: w[i][j], v[i][j], 1≤i≤G, 1≤j≤n_i
MCKP(n,W):
    for w = 0 to W //initialize array M[]
        M[0, w] <- 0
    for i = 1 to G //consider groups 1 to i
        for w = 0 to W //consider knapsack's capacity = w
            M[i, w] <- M[i-1, w]
            for j = 1 to n_i //check objects in group i
                if(w ≥ w_i,j && (M[i, w] < (v_i,j + M[i-1,w-w_i,j])))
                    M[i, w] <- v_i,j + M[i-1,w-w_i,j]
    Return M[G,W]
```

Running time = O(nGW)? O(nW)?

$$\sum_{i=1}^{G}\sum_{w=0}^{W}\sum_{j=1}^{n_i} c = c\sum_{w=0}^{W}\sum_{i=1}^{G}\sum_{j=1}^{n_i} 1 = c\sum_{w=0}^{W} n = cnW$$



Knapsack capacity, W=6

Object i.j	Weight ($w_{i,j}$)	Value ($v_{i,j}$)
1,1	1	4
1,2	2	9
2,1	2	5
2,2	3	10
2,3	4	12
3,1	2	8

$$M[i,w] = \begin{cases} 0, \text{ if } i = 0 \text{ (base case)} \\ M[i-1, w], \text{ if } w < w_{i,j} \text{ for all } j \\ \max\{M[i-1,w], \max\limits_{j, w \ge w_{i,j}}\{v_{i,j} + M[i-1, w-w_{i,j}]\}\}, \text{ otherwise} \end{cases}$$

Another array to store which object was selected

i \ w	0	1	2	3	4	5	6
0	0, 0	0, 0	0, 0	0, 0	0, 0	0, 0	0, 0
1	0, 0	4, 1	9, 2	9, 2	9, 2	9, 2	9, $\boxed{2}$
2	0, 0	4, 0	9, 0	10, 2	14, 2	19, 2	19, 2
3	0, 0	4, 0	9, 0	12, 1	17, 1	19, 0	22, 1

Variants of knapsack problem

Goal: fill knapsack so as to maximize total value

Each variant considers different constraints

1. 0/1 knapsack problem
   - 每項物品只能拿一個
2. Unbounded knapsack problem
   - 每項物品可以拿多個
3. Multiple-choice knapsack problem
   - 每一類物品最多拿一個
4. Multidimensional knapsack problem
   - 背包空間有限

…

Step 1: Characterize an optimal solution

<u>Multidimensional Knapsack Problem:</u> Fill a knapsack of capacity W and **size D** so as to maximize total value; each object is selected at most once

- w_i = weight of object i, v_i = value of object i ($w_i>0$, $v_i>0$)

MKP(i, w, d) = Multidimensional Knapsack Problem with **total weight ≤ w** and **size ≤ d** using **object 1 to i**

Suppose OPT is an optimal solution to MKP(i, w, d)

For object i, there are 2 possibilities:

- <u>Case 1: object i is in OPT</u>
  - OPT\{i} is an optimal solution to MKP(i-1, w-w_i, d-d_i)
- <u>Case 2: object i is not in OPT</u>
  - OPT is an optimal solution to MKP(i-1, w, d)

Step 2: Recursively define the value of an optimal solution

MKP(i, w, d) = Multidimensional Knapsack Problem with total weight ≤ w and size ≤ d using group 1 to i

Case 1: object i is in OPT

$$M[i,w,d] = v_i + M[i-1,w-w_i,d-d_i]$$

- OPT/{i} is an optimal solution to MKP(i-1, w-w_i, d-d_i)

Case 2: object i is not in OPT

$$M[i,w,d] = M[i-1,w,d]$$

- OPT is an optimal solution to MKP(i-1, w, d)

M[i, w, d] = the value of an optimal solution to MKP(i, w, d)

用遞迴表示最佳解的值：

$$M[i,w,d] = \begin{cases} 0, & \text{if } i = 0 \\ M[i-1,w,d], & \text{if } w_i > w \text{ or } d_i > d \\ \max\{v_i + M[i-1,w-w_i,d-d_i], M[i-1,w,d]\}, & \text{otherwise} \end{cases}$$

Practice: finish Step 3 and 4

Non-integer weights

When weights are integer, # of subproblems and thus time complexity is linear to W

What happen if the weights of objects are not integer?

Considering 0/1 Knapsack: What are the number of possible weights of the knapsack when
$w_1 = 1/3$, $w_2 = 2/7$, $w_3 = 9/10$, $W = 2$?

Ans: $\sim 2^3$

Non-integer weights

If weights are non-integers, the number of possible weights of the knapsack can be up to 2^n

Integer weights ensure the number of possible weights of the knapsack is up to W

- Many overlapping subproblems!
- However, recall that this is pseudo-polynomial, as the length of W's bit representation is s = lgW. That is, the time complexity grows with 2^s

Variants of knapsack problem

Goal: fill knapsack so as to maximize total value

Each variant considers different constraints

1. 0/1 knapsack problem
   - 每項物品只能拿一個
2. Unbounded knapsack problem
   - 每項物品可以拿多個
3. Multiple-choice knapsack problem
   - 每一類物品最多拿一個
4. Multidimensional knapsack problem
   - 背包空間有限
5. Fractional knapsack problem
   - 物品可以只拿一部分

Fractional knapsack problem

What if we can take fractions of items, rather than taking 0 or 1 item?

While DP works, there is a simpler approach…

- Hint: act like a “greedy” customer!
- Our next topic

Knapsack capacity = 6kg

A. 瑞士
1kg, $300

B. Mozzarella
2kg, $500

C. 巧達
3kg, $1300

D. 藍乳酪
10kg, $4000

What did you learn about DP?

A powerful design paradigm that can be used to solve many problems in polynomial time for which a naive approach would take exponential time.

Applicable when subproblems are overlapping

Two equivalent ways to avoid recomputation

- Top-down with memoization
- Bottom-up method

Commonly used to solve optimization problems

- Such problems should have optimal substructure