The Oragne Juice of DP
動態規劃柳橙汁

球主 (kelvin)

January 27, 2014

DP 的故事

	A	B	C	N	Y	R	Q	C	L	C	R	P	M
A	**8**	7	6	6	5	4	4	3	3	2	1	0	0
Y	7	**7**	6	6	6	4	4	3	3	2	1	0	0
C	6	6	**7**	**6**	5	4	4	4	3	3	1	0	0
Y	6	6	6	5	**6**	4	4	3	3	2	1	0	0
N	5	5	5	6	**5**	4	4	3	3	2	1	0	0
R	4	4	4	4	4	**5**	**4**	3	3	2	2	0	0
C	3	3	4	3	3	3	3	**4**	3	3	1	0	0
K	3	3	3	3	3	3	3	3	**3**	2	1	0	0
C	2	2	3	2	2	2	2	3	2	**3**	1	0	0
R	2	1	1	1	1	2	1	1	1	1	**2**	0	0
B	1	2	1	1	1	1	1	1	1	1	**1**	0	0
P	0	0	0	0	0	0	0	0	0	0	0	**1**	0

DP 的故事

451499	May 13, 2011 8:48:28 PM	kelvin	D - Numbers	GNU C++	Wrong answer on test 37	230 ms	79700 KB
449351	May 13, 2011 8:00:13 PM	kelvin	C - Track	GNU C++	Wrong answer on test 13	160 ms	1500 KB
447761	May 13, 2011 7:29:58 PM	kelvin	B - Doctor	GNU C++	Wrong answer on test 50	90 ms	2200 KB
446558	May 13, 2011 7:15:05 PM	kelvin	A - Magical Array	GNU C++	Accepted	60 ms	1400 KB

DP 的故事

DP 到底是什麼！

DP 的故事

DP 只是個柳橙！

DP 的故事

5029972	17:52:40 22 Jun 2013	kelvin	I. Hong Kong Tram	G++ 4.7.2	Accepted
5029436	16:54:06 22 Jun 2013	kelvin	H. Glass Pyramid	G++ 4.7.2	Accepted
5029318	16:39:08 22 Jun 2013	kelvin	G. Programmer Casino	G++ 4.7.2	Accepted
5028990	15:59:08 22 Jun 2013	kelvin	F. Cycling Roads	G++ 4.7.2	Accepted
5028802	15:37:17 22 Jun 2013	kelvin	E. Pear Trees	G++ 4.7.2	Accepted
5028531	15:07:30 22 Jun 2013	kelvin	D. Chinese Dialects	G++ 4.7.2	Accepted

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離

我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離

我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離

我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離

我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

忽遠忽近、若即若離
我想我們都擁有自己的森林

DP 的故事

DP 只是個柳橙！

DP 的故事

寫 1000 題 ACM 就告訴你！

DP 的故事

kelvin (+1 ironwood branch) (kelvin)

2884	1356	1331	2005-11-02	2013-05-21
SUBMISSIONS	PROBLEMS TRIED	PROBLEMS SOLVED	FIRST SUBMISSION	LAST SUBMISSION

DP 的故事

DP 只是個流程！

DP 的故事

可重複利用的最佳子結構

是他的精神

DP 的故事

狀態間的轉移
是計算的方法

DP 的故事

狀態的設計

是箇中的蹊蹺

DP 的故事

Overview

- 子樹 DP
- 單調性 DP
- 凸包優化 DP
- DP 的空間壓縮

Subtree DP

Chapter I

SUBTREE DP

Subtree DP is Powerful

The Weakness of DP:

- 沒有好的狀態描述方式 / 狀態空間太龐大。
- 狀態難以定序 $\Rightarrow$ 無法決定 DP 順序。
- 狀態之間的循環依賴關係。

DP on a Tree:

- 狀態通常都能用 V^k 描述。
- 若能夠定根，通常定序問題就徹底解決。
- 大部分的問題不會有循環依賴關係。

Subtree DP is Powerful

The Weakness of DP:

- **沒有好的狀態描述方式 / 狀態空間太龐大。**
- 狀態難以定序 $\Rightarrow$ 無法決定 DP 順序。
- 狀態之間的循環依賴關係。

DP on a Tree:

- 狀態通常都能用 V^k 描述。
- 若能夠定根，通常定序問題就徹底解決。
- 大部分的問題不會有循環依賴關係。

Subtree DP is Powerful

The Weakness of DP:

- **沒有好的狀態描述方式 / 狀態空間太龐大。**
- **狀態難以定序 $\Rightarrow$ 無法決定 DP 順序。**
- 狀態之間的循環依賴關係。

DP on a Tree:

- 狀態通常都能用 V^k 描述。
- 若能夠定根，通常定序問題就徹底解決。
- 大部分的問題不會有循環依賴關係。

Subtree DP is Powerful

The Weakness of DP:

- **沒有好的狀態描述方式 / 狀態空間太龐大。**
- **狀態難以定序 $\Rightarrow$ 無法決定 DP 順序。**
- **狀態之間的循環依賴關係。**

DP on a Tree:

- 狀態通常都能用 V^k 描述。
- 若能夠定根，通常定序問題就徹底解決。
- 大部分的問題不會有循環依賴關係。

Subtree DP is Powerful

The Weakness of DP:

- **沒有好的狀態描述方式 / 狀態空間太龐大。**
- **狀態難以定序 $\Rightarrow$ 無法決定 DP 順序。**
- **狀態之間的循環依賴關係。**

DP on a Tree:

- **狀態通常都能用 V^k 描述。**
- **若能夠定根，通常定序問題就徹底解決。**
- **大部分的問題不會有循環依賴關係。**

Subtree DP is Powerful

The Weakness of DP:

- 沒有好的狀態描述方式 / 狀態空間太龐大。
- 狀態難以定序 $\Rightarrow$ 無法決定 DP 順序。
- 狀態之間的循環依賴關係。

DP on a Tree:

- 狀態通常都能用 V^k 描述。
- 若能夠定根，通常定序問題就徹底解決。
- 大部分的問題不會有循環依賴關係。

Subtree DP is Powerful

The Weakness of DP:

- 沒有好的狀態描述方式 / 狀態空間太龐大。
- 狀態難以定序 $\Rightarrow$ 無法決定 DP 順序。
- 狀態之間的循環依賴關係。

DP on a Tree:

- 狀態通常都能用 V^k 描述。
- 若能夠定根，通常定序問題就徹底解決。
- 大部分的問題不會有循環依賴關係。

例題：感染問題

共 N 個城市的連通情形可以描述成一個 tree。

現在有若干個城市中爆發了嚴重的病毒。更糟的是，病毒會沿著道路不斷由被感染的城市擴散到相鄰的未被感染的城市。

迫不得已下，政府決定斷尾求生：封鎖若干條道路 (疫情也就無法通過這條道路傳播) 以防止疫情擴大，並希望至少能挽救 K 個城市中的人民。

今給定這個國家的城市連接情形，以及疫情爆發的城市，和想挽救的城市數目 K，試問：至少要封鎖幾條道路才能達到目的？

$(1 \leq K \leq N \leq 2000)$

例題：感染問題

- Input: 樹形、哪些點被感染、至少拯救 K 個點。
- Output: 最少封鎖多少道路。

- Observation:
  假如我們完全決定哪些點要被拯救，那些被放棄 (被感染)，則需要被砍掉的邊就是那些交界處的邊。
- Rephrase:
  給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，可把一些白點塗黑，求 至多塗黑 b 個白點時，至少要砍多少條邊？
- Reform:
  給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，至多砍 m 條邊時，最多能有幾個白點？

例題：感染問題

- Input: **樹形、哪些點被感染、至少拯救 K 個點。**
- Output: **最少封鎖多少道路。**

- Observation:
  假如我們完全決定哪些點要被拯救，那些被放棄 (被感染)，則需要被砍掉的邊就是那些交界處的邊。
- Rephrase:
  給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，可把一些白點塗黑，求 至多塗黑 b 個白點時，至少要砍多少條邊？
- Reform:
  給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，至多砍 m 條邊時，最多能有幾個白點？

例題：感染問題

- Input: 樹形、哪些點被感染、至少拯救 K 個點。
- Output: 最少封鎖多少道路。

- Observation:
  假如我們完全決定哪些點要被拯救，那些被放棄 (被感染)，則需要被砍掉的邊就是那些交界處的邊。
- Rephrase:
  給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，可把一些白點塗黑，求 至多塗黑 b 個白點時，至少要砍多少條邊？
- Reform:
  給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，至多砍 m 條邊時，最多能有幾個白點？

例題：感染問題

- Input: 樹形、哪些點被感染、至少拯救 K 個點。
- Output: 最少封鎖多少道路。

- Observation:
  假如我們完全決定哪些點要被拯救，那些被放棄 (被感染)，則需要被砍掉的邊就是那些交界處的邊。
- Rephrase:
  給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，可把一些白點塗黑，求 至多塗黑 b 個白點時，至少要砍多少條邊？
- Reform:
  給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，至多砍 m 條邊時，最多能有幾個白點？

例題：感染問題

- Input: 樹形、哪些點被感染、至少拯救 K 個點。
- Output: 最少封鎖多少道路。

- Observation:
  假如我們完全決定哪些點要被拯救，那些被放棄 (被感染)，則需要被砍掉的邊就是那些交界處的邊。
- Rephrase:
  給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，可把一些白點塗黑，求 至多塗黑 b 個白點時，至少要砍多少條邊？
- Reform:
  給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，至多砍 m 條邊時，最多能有幾個白點？

例題：感染問題 | Greedy?

"是否其實可以 greedy 的挑邊來砍？"

- 把目標函數 (objective function) 描述為 $f(m) = w$，亦即砍 m 條邊時最多塗出 w 個白點，$f(m)$ 不嚴格遞增。

  $\Rightarrow$ 有時候要想要多塗幾個白點，得多砍不只一條邊。

- 以 greedy 來說非常不理想。

"是否其實可以 greedy 的挑點來塗？"

- 情形雷同。

例題：感染問題 | Greedy?

“是否其實可以 greedy 的挑邊來砍？”

- 把目標函數 (objective function) 描述為 $f(m) = w$，亦即砍 m 條邊時最多塗出 w 個白點，$f(m)$ 不嚴格遞增。

  $\Rightarrow$ 有時候要想要多塗幾個白點，得多砍不只一條邊。

- 以 greedy 來說非常不理想。

“是否其實可以 greedy 的挑點來塗？”

- 情形雷同。

例題：感染問題 | Greedy?

“是否其實可以 greedy 的挑邊來砍？”

- 把目標函數 (objective function) 描述為 $f(m) = w$，亦即砍 m 條邊時最多塗出 w 個白點，$f(m)$ 不嚴格遞增。

  $\Rightarrow$ 有時候要想要多塗幾個白點，得多砍不只一條邊。

- 以 greedy 來說非常不理想。

“是否其實可以 greedy 的挑點來塗？”

- 情形雷同。

例題：感染問題 | Greedy?

“是否其實可以 greedy 的挑邊來砍？”

- 把目標函數 (objective function) 描述為 $f(m) = w$，亦即砍 m 條邊時最多塗出 w 個白點，$f(m)$ 不嚴格遞增。

  $\Rightarrow$ 有時候要想要多塗幾個白點，得多砍不只一條邊。

- 以 greedy 來說非常不理想。

“是否其實可以 greedy 的挑點來塗？”

- 情形雷同。

例題：感染問題 | Greedy?

“是否其實可以 greedy 的挑邊來砍？”

- 把目標函數 (objective function) 描述為 $f(m) = w$，亦即砍 m 條邊時最多塗出 w 個白點，$f(m)$ 不嚴格遞增。

  $\Rightarrow$ 有時候要想要多塗幾個白點，得多砍不只一條邊。

- 以 greedy 來說非常不理想。

“是否其實可以 greedy 的挑點來塗？”

- 情形雷同。

例題：感染問題 | Greedy?

"是否其實可以 greedy 的挑邊來砍？"

- 把目標函數 (objective function) 描述為 $f(m) = w$，亦即砍 m 條邊時最多塗出 w 個白點，$f(m)$ 不嚴格遞增。

  $\Rightarrow$ 有時候要想要多塗幾個白點，得多砍不只一條邊。

- 以 greedy 來說非常不理想。

"是否其實可以 greedy 的挑點來塗？"

- 情形雷同。

例題：感染問題 | Flow?

“黑白交界？Min-Cut 乎？”

- 「塗 k 個白點 + 最少砍多少邊」。

  $\Rightarrow$ cut 某側至多有多少點

- 上述條件一般來說是 min-cut 的大忌。
- flow 掰掰。

例題：感染問題 | Flow?

“黑白交界？Min-Cut 乎？”

- 「塗 k 個白點 + 最少砍多少邊」。

  $\Rightarrow$ cut 某側至多有多少點

- 上述條件一般來說是 min-cut 的大忌。
- flow 掰掰。

例題：感染問題 | Flow?

“黑白交界？Min-Cut 乎？”

- 「塗 k 個白點 + 最少砍多少邊」。

  $\Rightarrow$ cut 某側至多有多少點

- 上述條件一般來說是 min-cut 的大忌。
- flow 掰掰。

例題：感染問題 | Flow?

"黑白交界？Min-Cut 乎？"

- 「塗 k 個白點 + 最少砍多少邊」。

  $\Rightarrow$ cut 某側至多有多少點

- 上述條件一般來說是 min-cut 的大忌。
- flow 掰掰。

例題：感染問題 | Flow?

"黑白交界？Min-Cut 乎？"

- 「塗 k 個白點 + 最少砍多少邊」。

  $\Rightarrow$ cut 某側至多有多少點

- 上述條件一般來說是 min-cut 的大忌。
- flow 掰掰。

例題：感染問題 | DP

Form 1:

- 給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，可把一些白點塗黑，求 至多塗黑 b 個白點時，至少要砍多少條邊？

Form 2:

- 給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，至多砍 m 條邊時，最多能有幾個白點？

$$\text{bool } DP[v][b][m] \Rightarrow \begin{cases} \text{int } DP[v][b] = \min m \\ \text{int } DP[v][m] = \min b \end{cases}$$

例題：感染問題 | DP

Form 1:

- 給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，可把一些白點塗黑，求 至多塗黑 b 個白點時，至少要砍多少條邊？

Form 2:

- 給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，至多砍 m 條邊時，最多能有幾個白點？

$$\text{bool } DP[v][b][m] \Rightarrow \begin{cases} \text{int } DP[v][b] = \min m \\ \text{int } DP[v][m] = \min b \end{cases}$$

例題：感染問題 | DP

Form 1:

- 給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，可把一些白點塗黑，求 至多塗黑 b 個白點時，至少要砍多少條邊？

Form 2:

- 給一顆有黑點 (受感染) 與白點 (未被感染) 的樹，至多砍 m 條邊時，最多能有幾個白點？

$$\text{bool } DP[v][b][m] \Rightarrow \begin{cases} \text{int } DP[v][b] = \min m \\ \text{int } DP[v][m] = \min b \end{cases}$$

例題：感染問題 | DP

1. **定根：隨意以一點當根。**
2. **狀態：** $DP[v][m] = \max w$
3. **轉移：**
   令 $u_1, u_2, \ldots, u_t$ 為 v 的兒子，稱以 u_i 為根的 sub-tree 為 T_i。若 T_i 中各砍了 m_i 條，有 w_i 個白點，那麼：

$$m = \sum_{i=1}^{t} (\|v \leftrightarrow u_i\| + m_i)$$

$$w = \|v : \mathrm{WHITE}\| + \sum_{i=1}^{k} w_i$$

例題：感染問題 | DP

1. **定根：隨意以一點當根。**
2. **狀態：** $DP[v][m] = \max w$
3. **轉移：**

令 $u_1, u_2, \ldots, u_t$ 為 v 的兒子，稱以 u_i 為根的 sub-tree 為 T_i。若 T_i 中各砍了 m_i 條，有 w_i 個白點，那麼：

$$m = \sum_{i=1}^{t} (\|v \leftrightarrow u_i\| + m_i)$$

$$w = \|v : \mathrm{WHITE}\| + \sum_{i=1}^{k} w_i$$

例題：感染問題 | DP

1. **定根：隨意以一點當根。**
2. **狀態：** $DP[v][m] = \max w$
3. **轉移：**
   令 $u_1, u_2, \ldots, u_t$ **為** v **的兒子，稱以** u_i **為根的** sub-tree **為** T_i**。若** T_i **中各砍了** m_i **條，有** w_i **個白點，那麼：**

$$m = \sum_{i=1}^{t} (\|v \leftrightarrow u_i\| + m_i)$$

$$w = \|v : \mathrm{WHITE}\| + \sum_{i=1}^{k} w_i$$

例題：感染問題 | DP

1. **定根：隨意以一點當根。**
2. **狀態：** $DP[v][m] = \max w$
3. **轉移：**
   令 $u_1, u_2, \ldots, u_t$ **為** v **的兒子，稱以** u_i **為根的** sub-tree **為** T_i**。若** T_i **中各砍了** m_i **條，有** w_i **個白點，那麼：**

$$m = \sum_{i=1}^{t} (\|v \leftrightarrow u_i\| + m_i)$$

$$w = \|v : \mathrm{WHITE}\| + \sum_{i=1}^{t} w_i$$

例題：感染問題 | DP

1. **定根：隨意以一點當根。**
2. **狀態：** $DP[v][m][c] = \max w$
3. **轉移：**
   令 $u_1, u_2, \ldots, u_t$ **為** v **的兒子，稱以** u_i **為根的** sub-tree **為** T_i**。若** T_i **中各砍了** m_i **條，有** w_i **個白點，那麼：**

$$m = \sum_{i=1}^{t} (\|v \leftrightarrow u_i\| + m_i)$$

$$w = \|v : \mathrm{WHITE}\| + \sum_{i=1}^{t} w_i$$

例題：感染問題 | DP

1. **定根：隨意以一點當根。**
2. **狀態：** $DP[v][m][c] = \max w$
3. **轉移：**
   令 $u_1, u_2, \ldots, u_t$ **為** v **的兒子，稱以** u_i **為根的** sub-tree **為** T_i**。若** T_i **中各砍了** m_i **條，有** w_i **個白點，那麼：**

$$m = \sum_{i=1}^{t} (\|c \neq c_i\| + m_i)$$

$$w = \|c = \mathrm{WHITE}\| + \sum_{i=1}^{t} w_i$$

例題：感染問題 | DP

$$DP[v][m][c] = \max_{m_i, c_i,\ \forall i=1\ldots t} \left\{ \|c = \text{WHITE}\| + \sum_{i=1}^{t} DP[u_i][m_i][c_i] \right\}$$

where

$$\sum_{i=1}^{t} (\|c \neq c_i\| + m_i) = m$$

- 枚舉 m_i, c_i 要嘗試的組合太多了？怎麼辦？
- DP again！背包 DP！

例題：感染問題 | DP

$$DP[v][m][c] = \max_{m_i, c_i,\ \forall i=1\ldots t} \left\{ \|c = \text{WHITE}\| + \sum_{i=1}^{t} DP[u_i][m_i][c_i] \right\}$$

where

$$\sum_{i=1}^{t} (\|c \neq c_i\| + m_i) = m$$

- **枚舉 m_i, c_i 要嘗試的組合太多了？怎麼辦？**
- DP again！背包 DP！

例題：感染問題 | DP

$$DP[v][m][c] = \max_{m_i, c_i,\ \forall i=1\ldots t} \left\{ \|c = \text{WHITE}\| + \sum_{i=1}^{t} DP[u_i][m_i][c_i] \right\}$$

where

$$\sum_{i=1}^{t} (\|c \neq c_i\| + m_i) = m$$

- 枚舉 m_i, c_i 要嘗試的組合太多了？怎麼辦？
- DP again！背包 DP！

例題：感染問題 | DP

- **背包問題：**
  你要從 t 個罐子裡面拿糖果。已知你從第 i 個罐子裡面拿 m_i 顆時，回得到 f_{i,m_i} 那麼多的滿足感。你總共只能拿 m 顆時，最多可以獲得多少滿足感？

- $DP2[i][s] =$
  拿到第 i 罐時，已經拿了 s 顆，可獲多少滿足感

- $DP2[i][s] = \max_p \{DP2[i-1][s-p] + f_{i,p}\}$

- **罐子 $\leftrightarrow$ 子樹**
  糖果 $\leftrightarrow$ 砍邊
  滿足 $\leftrightarrow$ 白點

例題：感染問題 | DP

- **背包問題：**
  你要從 t 個罐子裡面拿糖果。已知你從第 i 個罐子裡面拿 m_i 顆時，回得到 f_{i,m_i} 那麼多的滿足感。你總共只能拿 m 顆時，最多可以獲得多少滿足感？

- $DP2[i][s] =$
  拿到第 i 罐時，已經拿了 s 顆，可獲多少滿足感

- $DP2[i][s] = \max_p \{DP2[i-1][s-p] + f_{i,p}\}$

- 罐子 $\leftrightarrow$ 子樹
  糖果 $\leftrightarrow$ 砍邊
  滿足 $\leftrightarrow$ 白點

例題：感染問題 | DP

- **背包問題：**
  你要從 t 個罐子裡面拿糖果。已知你從第 i 個罐子裡面拿 m_i 顆時，回得到 f_{i,m_i} 那麼多的滿足感。你總共只能拿 m 顆時，最多可以獲得多少滿足感？
- $DP2[i][s] =$
  拿到第 i 罐時，已經拿了 s 顆，可獲多少滿足感
- $DP2[i][s] = \max_p \{DP2[i-1][s-p] + f_{i,p}\}$
- 罐子 $\leftrightarrow$ 子樹
  糖果 $\leftrightarrow$ 砍邊
  滿足 $\leftrightarrow$ 白點

例題：感染問題 | DP

- **背包問題：**
  你要從 t 個罐子裡面拿糖果。已知你從第 i 個罐子裡面拿 m_i 顆時，回得到 f_{i,m_i} 那麼多的滿足感。你總共只能拿 m 顆時，最多可以獲得多少滿足感？

- $DP2[i][s] =$
  拿到第 i 罐時，已經拿了 s 顆，可獲多少滿足感

- $DP2[i][s] = \max_p \{DP2[i-1][s-p] + f_{i,p}\}$

- **罐子 $\leftrightarrow$ 子樹**
  糖果 $\leftrightarrow$ 砍邊
  滿足 $\leftrightarrow$ 白點

例題：感染問題 | 時間複雜度

時間複雜度是多少呢？

- $DP[\cdot][\cdot][\cdot]$：可直接從 $DP2$ 獲得。
- $DP2[i][s] = \max_p \{DP2[i-1][s-p] + f_{i,p}\}$ ，$\forall i, s$
- $O(v \text{ 個點} \times t(\because i) \times N(\because s)) = O(N^3)$。
- 那麼爛?!

例題：感染問題 | 時間複雜度

時間複雜度是多少呢？

- $DP[\cdot][\cdot][\cdot]$：**可直接從** $DP2$ **獲得。**
- $DP2[i][s] = \max_p \{DP2[i-1][s-p] + f_{i,p}\}$ ，$\forall i, s$
- $O(v \text{ 個點} \times t(\because i) \times N(\because s)) = O(N^3)$。
- 那麼爛?!

例題：感染問題 | 時間複雜度

時間複雜度是多少呢？

- $DP[\cdot][\cdot][\cdot]$: **可直接從** $DP2$ **獲得。**
- $DP2[i][s] = \max_p \{DP2[i-1][s-p] + f_{i,p}\}$,
  $\forall i, s$
- $O(v \text{ 個點} \times t(\cdot\cdot i) \times N(\cdot\cdot s)) = O(N^3)$。
- 那麼爛?!

例題：感染問題 | 時間複雜度

時間複雜度是多少呢？

- $DP[\cdot][\cdot][\cdot]$：**可直接從** $DP2$ **獲得。**
- $DP2[i][s] = \max_{0 \le p \le N} \{DP2[i-1][s-p] + f_{i,p}\}$ ，
  $\forall 0 \le i \le t, 0 \le s \le N$
- $O(v \text{ 個點} \times t(\because i) \times N(\because s)) = O(N^3)$。
- 那麼爛?!

例題：感染問題 | 時間複雜度

時間複雜度是多少呢？

- $DP[\cdot][\cdot][\cdot]$：**可直接從** $DP2$ **獲得**。
- $DP2[i][s] = \max_{0 \le p \le N} \{DP2[i-1][s-p] + f_{i,p}\}$ ，
  $\forall 0 \le i \le t, 0 \le s \le N$
- $O(v$ **個點** $\times\ t(\because i) \times N(\because s)) = O(N^3)$。
- 那麼爛?!

例題：感染問題 | 時間複雜度

時間複雜度是多少呢？

- $DP[\cdot][\cdot][\cdot]$: **可直接從** $DP2$ **獲得。**
- $DP2[i][s] = \max_{0 \le p \le N} \{DP2[i-1][s-p] + f_{i,p}\}$,
  $\forall 0 \le i \le t, 0 \le s \le N$
- $O($**v 個點** $\times\ t(\because i) \times N(\because s)) = O(N^3)$ **。**
- **那麼爛?!**

例題：感染問題 | 時間複雜度

例題：感染問題 | 時間複雜度

胡扯!!!

例題：感染問題 | 時間複雜度

考慮各種 case . . .

- 直鏈：在每個點只需要決定顏色 (c)，$O(N^2)$。
- 平衡樹：每點 $DP2$ 只需要 $O(2N)$，N 點共 $O(N^2)$。
- 星狀樹：相當於只要對根做一次 $DP2$ ，$O(N^2)$。

例題：感染問題 | 時間複雜度

考慮各種 case . . .

- 直鏈：在每個點只需要決定顏色 (c)，$O(N^2)$。
- 平衡樹：每點 $DP2$ 只需要 $O(2N)$，N 點共 $O(N^2)$。
- 星狀樹：相當於只要對根做一次 $DP2$ ，$O(N^2)$。

例題：感染問題 | 時間複雜度

考慮各種 case . . .

- 直鏈：在每個點只需要決定顏色 (c)，$O(N^2)$。
- 平衡樹：每點 $DP2$ 只需要 $O(2N)$，N 點共 $O(N^2)$。
- 星狀樹：相當於只要對根做一次 $DP2$ ，$O(N^2)$。

例題：感染問題 | 時間複雜度

考慮各種 case . . .

- 直鏈：在每個點只需要決定顏色 (c)，$O(N^2)$。
- 平衡樹：每點 $DP2$ 只需要 $O(2N)$，N 點共 $O(N^2)$。
- 星狀樹：相當於只要對根做一次 $DP2$ ，$O(N^2)$。

例題：感染問題 | 時間複雜度

$$DP2[i][s] = \max_{p_i} \{DP2[i-1][s-p_i] + f_{i,p_i}\}$$

$$O(v \text{ 個點} \times t(\because i) \times N(\because s))$$

- 該拿出均攤分析果菜機了！
- 令 $n(v)$ 代表以 v 為根的子樹有多少個點。
- 上式中 $s - p_i$ 只有 $\sum_{j=1}^{i-1} n(j)$ 那麼多選擇。
- 上式中 p_i 只有 $n(i)$ 那麼多選擇。
- 因此對於點 v 來說，要做的事情的量為：

$$\sum_{i=1}^{t} \left[n(i) \cdot \left(\sum_{j=1}^{i-1} n(j) \right) \right]$$

例題：感染問題 | 時間複雜度

$$DP2[i][s] = \max_{p_i} \{DP2[i-1][s-p_i] + f_{i,p_i}\}$$

$$O(v \text{ 個點} \times t(\because i) \times N(\because s))$$

- **該拿出均攤分析果菜機了！**
- 令 $n(v)$ 代表以 v 為根的子樹有多少個點。
- 上式中 $s - p_i$ 只有 $\sum_{j=1}^{i-1} n(j)$ 那麼多選擇。
- 上式中 p_i 只有 $n(i)$ 那麼多選擇。
- 因此對於點 v 來說，要做的事情的量為：

$$\sum_{i=1}^{t} \left[n(i) \cdot \left(\sum_{j=1}^{i-1} n(j) \right) \right]$$

例題：感染問題 | 時間複雜度

$$DP2[i][s] = \max_{p_i} \{DP2[i-1][s-p_i] + f_{i,p_i}\}$$

$$O(v \text{ 個點} \times t(\because i) \times N(\because s))$$

- 該拿出均攤分析果菜機了！
- 令 $n(v)$ 代表以 v 為根的子樹有多少個點。
- 上式中 $s - p_i$ 只有 $\sum_{j=1}^{i-1} n(j)$ 那麼多選擇。
- 上式中 p_i 只有 $n(i)$ 那麼多選擇。
- 因此對於點 v 來說，要做的事情的量為：

$$\sum_{i=1}^{t} \left[n(i) \cdot \left(\sum_{j=1}^{i-1} n(j) \right) \right]$$

例題：感染問題 | 時間複雜度

$$DP2[i][s] = \max_{p_i} \{DP2[i-1][s-p_i] + f_{i,p_i}\}$$

$$O(v \text{ 個點} \times t(\because i) \times N(\because s))$$

- 該拿出均攤分析果菜機了！
- 令 $n(v)$ 代表以 v 為根的子樹有多少個點。
- 上式中 $s - p_i$ 只有 $\sum_{j=1}^{i-1} n(j)$ 那麼多選擇。
- 上式中 p_i 只有 $n(i)$ 那麼多選擇。
- 因此對於點 v 來說，要做的事情的量為：

$$\sum_{i=1}^{t} \left[n(i) \cdot \left(\sum_{j=1}^{i-1} n(j) \right) \right]$$

例題：感染問題 | 時間複雜度

$$DP2[i][s] = \max_{p_i} \{DP2[i-1][s-p_i] + f_{i,p_i}\}$$

$$O(v \text{ 個點} \times t(\because i) \times N(\because s))$$

- 該拿出均攤分析果菜機了！
- 令 $n(v)$ 代表以 v 為根的子樹有多少個點。
- 上式中 $s - p_i$ 只有 $\sum_{j=1}^{i-1} n(j)$ 那麼多選擇。
- 上式中 p_i 只有 $n(i)$ 那麼多選擇。
- 因此對於點 v 來說，要做的事情的量為：

$$\sum_{i=1}^{t} \left[n(i) \cdot \left(\sum_{j=1}^{i-1} n(j) \right) \right]$$

例題：感染問題 | 時間複雜度

$$DP2[i][s] = \max_{p_i} \{DP2[i-1][s-p_i] + f_{i,p_i}\}$$

$$O(v \text{ 個點} \times t(\because i) \times N(\because s))$$

- 該拿出均攤分析果菜機了！
- 令 $n(v)$ 代表以 v 為根的子樹有多少個點。
- 上式中 $s - p_i$ 只有 $\sum_{j=1}^{i-1} n(j)$ 那麼多選擇。
- 上式中 p_i 只有 $n(i)$ 那麼多選擇。
- 因此對於點 v 來說，要做的事情的量為：

$$\sum_{i=1}^{t} \left[n(i) \cdot \left(\sum_{j=1}^{i-1} n(j) \right) \right]$$

例題：感染問題 | 時間複雜度

- 對於點 v 來說，要做的事情的量：

$$\sum_{i=1}^{t} \left[n(u_i) \cdot \left(\sum_{j=1}^{i-1} n(u_j) \right) \right]$$
$$= \sum_{1 \le i < j \le t} n(u_i) n(u_j)$$

- 那加上處理完 v 之下所有點所需的時間呢？
(我們需要知道對應 $u_1, u_2, \ldots, u_t$ 子問題的解)
- 不知道 ...

例題：感染問題 | 時間複雜度

- 對於點 v 來說，要做的事情的量：

$$\sum_{i=1}^{t}\left[n(u_i)\cdot\left(\sum_{j=1}^{i-1}n(u_j)\right)\right]$$
$$=\sum_{1\le i<j\le t}n(u_i)n(u_j)$$

- 那加上處理完 v 之下所有點所需的時間呢？
(我們需要知道對應 $u_1, u_2, \ldots, u_t$ 子問題的解)
- 不知道 . . .

例題：感染問題 | 時間複雜度

- 對於點 v 來說，要做的事情的量：

$$\sum_{i=1}^{t}\left[n(u_i)\cdot\left(\sum_{j=1}^{i-1}n(u_j)\right)\right]$$
$$=\sum_{1\le i<j\le t}n(u_i)n(u_j)$$

- 那加上處理完 v 之下所有點所需的時間呢？
(我們需要知道對應 $u_1, u_2, \ldots, u_t$ 子問題的解)
- 不知道 ...

例題：感染問題 | 時間複雜度

- **最後一步鴻溝，我們請數學歸納法幫我們跨過去。**
- 假設做完以 u 為根整個子問題只需要 $n(u)^2$。
- 做完 $u_1, u_2, \ldots, u_t$ 子問題所需的時間：
- 所以總時間：
- 上面的分析為求簡明，刻意省去操作常數上的分析。比較正式的做法中，歸納假設中的複雜度應要乘上一常數 c，並去說明取合適大的 c 即可把歸納好好做完。

例題：感染問題 | 時間複雜度

- 最後一步鴻溝，我們請數學歸納法幫我們跨過去。
- 假設做完以 u 為根整個子問題只需要 $n(u)^2$。
- 做完 $u_1, u_2, \ldots, u_t$ 子問題所需的時間：
- 所以總時間：
- 上面的分析為求簡明，刻意省去操作常數上的分析。比較正式的做法中，歸納假設中的複雜度應要乘上一常數 c，並去說明取合適大的 c 即可把歸納好好做完。

例題：感染問題 | 時間複雜度

- 最後一步鴻溝，我們請數學歸納法幫我們跨過去。
- 假設做完以 u 為根整個子問題只需要 $n(u)^2$。
- 做完 $u_1, u_2, \ldots, u_t$ 子問題所需的時間：

$$n(u_1)^2 + n(u_2)^2 + \cdots + n(u_t)^2$$

- 所以總時間：
- 上面的分析為求簡明，刻意省去操作常數上的分析。比較正式的做法中，歸納假設中的複雜度應要乘上一常數 c，並去說明取合適大的 c 即可把歸納好好做完。

例題：感染問題 | 時間複雜度

- 最後一步鴻溝，我們請數學歸納法幫我們跨過去。
- 假設做完以 u 為根整個子問題只需要 $n(u)^2$。
- 做完 $u_1, u_2, \ldots, u_t$ 子問題所需的時間：

$$n(u_1)^2 + n(u_2)^2 + \cdots + n(u_t)^2$$

- 所以總時間：

$$\sum_{1 \le i < j \le t} n(u_i)n(u_j) + \sum_{1 \le i \le t} n(u_i)^2 = n(v)^2$$

- 上面的分析為求簡明，刻意省去操作常數上的分析。比較正式的做法中，歸納假設中的複雜度應要乘上一常數 c，並去說明取合適大的 c 即可把歸納好好做完。

例題：感染問題 | 時間複雜度

- 最後一步鴻溝，我們請數學歸納法幫我們跨過去。
- 假設做完以 u 為根整個子問題只需要 $n(u)^2$。
- 做完 $u_1, u_2, \ldots, u_t$ 子問題所需的時間：

$$n(u_1)^2 + n(u_2)^2 + \cdots + n(u_t)^2$$

- 所以總時間：

$$\sum_{1 \le i < j \le t} n(u_i)n(u_j) + \sum_{1 \le i \le t} n(u_i)^2 = n(v)^2$$

- 上面的分析為求簡明，刻意省去操作常數上的分析。比較正式的做法中，歸納假設中的複雜度應要乘上一常數 c，並去說明取合適大的 c 即可把歸納好好做完。

例題：感染問題 | 時間複雜度

$O(N^2)$ 圓滿解決。

例題：送貨配給問題

給一個 N 點的樹，每條邊長 l_i，每個點的貨物需求量 r_i，以及在點上設倉庫的花費 c_i。

現在你想要設計一個物流網：在若干個點設置倉庫，對於每個沒有設倉庫的點，則由最近的倉庫提供貨物。於是總花費將由倉庫設置費與貨物運送費決定；其中倉庫設置費就是所有有設置倉庫的點的設置費總和，貨物運費則是(起點倉庫到終點距離 $\times$ 貨物需求量)。

若你可以任意選擇要在哪些點設置倉庫，問最少總花費是多少？(當然，你至少需要在某處設置一個倉庫)

$(1 \leq N \leq 3000, 1 \leq c_i, r_i, l_i \leq 10^6)$

例題：送貨配給問題

分析：

- 花費 c、邊長 l、貨物量 r 都非常大，要儘量限制狀態只用到點集 V。
- 狀態設計複雜得多。一個點的貨物要運到那裡難以在局部決定。
- 子樹的狀態難以合併。

例題：送貨配給問題

分析：

- 花費 c、邊長 l、貨物量 r 都非常大，要儘量限制狀態只用到點集 V。
- 狀態設計複雜得多。一個點的貨物要運到那裡難以在局部決定。
- 子樹的狀態難以合併。

例題：送貨配給問題

分析：

- 花費 c、邊長 l、貨物量 r 都非常大，要儘量限制狀態只用到點集 V。
- 狀態設計複雜得多。一個點的貨物要運到那裡難以在局部決定。
- 子樹的狀態難以合併。

例題：送貨配給問題 | 第一次嘗試

- $DP[v][u] =$ 處理以v為根的子問題，其下最近倉庫為u。
- 難以解決貨物要運到那裡難以在局部決定的問題。
- 沒辦法不遺失狀態地轉移。

例題：送貨配給問題 | 第一次嘗試

- $DP[v][u] =$ 處理以v為根的子問題，其下最近倉庫為u。
- 難以解決**貨物要運到那裡難以在局部決定**的問題。
- 沒辦法不遺失狀態地轉移。

例題：送貨配給問題 | 第一次嘗試

- $DP[v][u] =$ 處理以v為根的子問題，其下最近倉庫為u。
- 難以解決貨物要運到那裡難以在局部決定的問題。
- 沒辦法不遺失狀態地轉移。

例題：送貨配給問題 | 第二次嘗試

- **難以事後合併，那我就預先決定！**
- 關鍵的觀察：假如在最佳解中，v 的貨物需運到 u，則 $v \to u$ 路徑上所有點的貨物肯定也是運到 u。
- 在處理點 v 的當下就決定 v 的所有貨物要運到哪！
- 轉移 (合併子樹) 的時候確保東西都有好好的運道同一個目的地。
- $DP[v][u] =$ 以v為根，當前所有貨物運到u。

例題：送貨配給問題 | 第二次嘗試

- 難以事後合併，那我就**預先決定**！
- 關鍵的觀察：假如在最佳解中，v 的貨物需運到 u，則 $v \to u$ **路徑上所有點的貨物肯定也是運到** u。
- 在處理點 v 的當下就決定 v 的所有貨物要運到哪！
- 轉移 (合併子樹) 的時候確保東西都有好好的運道同一個目的地。
- $DP[v][u] =$ 以v為根，當前所有貨物運到u。

例題：送貨配給問題 | 第二次嘗試

- 難以事後合併，那我就**預先決定**！
- 關鍵的觀察：假如在最佳解中，v 的貨物需運到 u，則 $v \to u$ **路徑上所有點的貨物肯定也是運到** u。
- 在處理點 v 的當下就決定 v 的所有貨物要運到哪！
- 轉移 (合併子樹) 的時候確保東西都有好好的運道同一個目的地。
- $DP[v][u] =$ 以v為根，當前所有貨物運到u。

例題：送貨配給問題 | 第二次嘗試

- 難以事後合併，那我就**預先決定**！
- 關鍵的觀察：假如在最佳解中，v 的貨物需運到 u，則 $v \to u$ **路徑上所有點的貨物肯定也是運到** u。
- 在處理點 v 的當下就決定 v 的所有貨物要運到哪！
- 轉移 (合併子樹) 的時候確保東西都有好好的運道同一個目的地。
- $DP[v][u] =$ 以v為根，當前所有貨物運到u。

例題：送貨配給問題 | 第二次嘗試

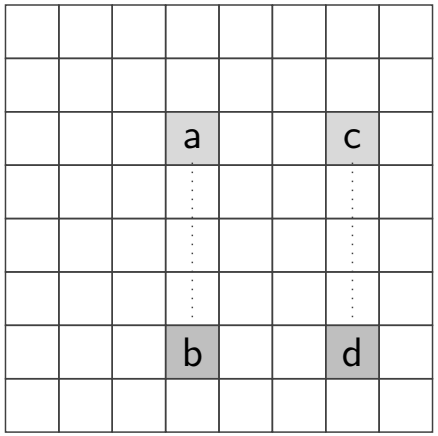
- 難以事後合併，那我就預先決定！
- 關鍵的觀察：假如在最佳解中，v 的貨物需運到 u，則 $v \to u$ 路徑上所有點的貨物肯定也是運到 u。
- 在處理點 v 的當下就決定 v 的所有貨物要運到哪！
- 轉移 (合併子樹) 的時候確保東西都有好好的運道同一個目的地。
- $DP[v][u] =$ 以v為根，當前所有貨物運到u。

Monotonicity

Chapter II

MONOTONICITY

矩陣的單調性



凹單調 (concave totally monotone)：

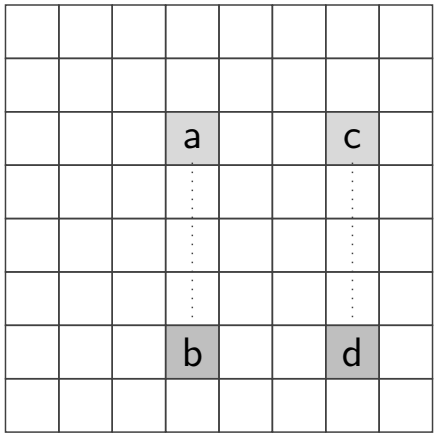
$$a \leq b \Rightarrow c \leq d$$

凸單調 (convex totally monotone)：

$$a \geq b \Rightarrow c \geq d$$

- 找矩陣列最小元素，有必勝法！

矩陣的單調性



凹單調 (concave totally monotone)：

$$a \leq b \Rightarrow c \leq d$$

凸單調 (convex totally monotone)：

$$a \geq b \Rightarrow c \geq d$$

- **找矩陣列最小元素，有必勝法！**

The tD-eD Problem

- 狀態：$DP[\cdot][\cdot][\cdot]$：一個 t 維表格。
- 轉移：$\min_{\forall i,j} DP[i][j][z] + f(i,j)$：仰賴 $O(N^e)$ 格。
- $\Rightarrow$ tD-eD 問題。
- 存在無腦 $O(N^{d+e})$ 算法。

The tD-eD Problem

- **狀態：** $DP[\cdot][\cdot][\cdot]$ **：一個** t **維表格。**
- **轉移：** $\min_{\forall i,j} DP[i][j][z] + f(i,j)$ **：仰賴** $O(N^e)$ **格。**
- $\Rightarrow$ tD-eD 問題。
- 存在無腦 $O(N^{d+e})$ 算法。

The tD-eD Problem

- **狀態：** $DP[\cdot][\cdot][\cdot]$ **：一個** t **維表格。**
- **轉移：** $\min_{\forall i,j} DP[i][j][z] + f(i,j)$ **：仰賴** $O(N^e)$ **格。**
- $\Rightarrow$ tD-eD **問題。**
- 存在無腦 $O(N^{d+e})$ 算法。

The tD-eD Problem

- **狀態：** $DP[\cdot][\cdot][\cdot]$ **：一個** t **維表格。**
- **轉移：** $\min_{\forall i,j} DP[i][j][z] + f(i,j)$ **：仰賴** $O(N^e)$ **格。**
- $\Rightarrow$ tD-eD **問題。**
- **存在無腦** $O(N^{d+e})$ **算法。**

1D-1D Problems

Definition (1D-1D 問題)

- **目標函數**：$E[j],\ \forall\ 1 \leq j \leq n$。
- Input：
  - **轉移代價** $f(i, j, E[i]),\ \forall\ 0 \leq i < j \leq n$
  - **初始化邊界** $E[0]$
- Output：

$$E[j] = \min_{0 \leq i < j} \{f(i, j, E[i])\},\ \forall\ 1 \leq i \leq n$$

1D-1D Problems

$$E[j] = \min_{0 \le i < j} \{ \text{f(i,j,E[i])} \},$$

$$\forall\ 1 \le i \le n$$

- $E[j]$ 是 矩陣 B 第 j 列的列最小元素。
- 注意：B 只在 對角線之上 有意義。
- 注意：B 是 在線 (online) 的。

1D-1D Problems

$$E[j] = \min_{0 \le i < j} \{ \quad \text{B[i][j]} \quad \},$$

$$\forall\, 1 \le j \le n, B[i][j] = f(i, j, E[i])$$

- $E[j]$ 是 矩陣 B 第 j 列的列最小元素。
- 注意：B 只在 對角線之上 有意義。
- 注意：B 是 在線 (online) 的。

1D-1D Problems

$$E[j] = \min_{0 \le i < j} \{ \quad \text{B[i][j]} \quad \},$$

$$\forall\, 1 \le j \le n, B[i][j] = f(i, j, E[i])$$

- $E[j]$ 是 矩陣 B 第 j 列的列最小元素。
- 注意：B 只在 對角線之上 有意義。
- 注意：B 是 在線 (online) 的。

1D-1D Problems

$$E[j] = \min_{0 \le i < j} \{ \quad \text{B[i][j]} \quad \},$$

$$\forall\, 1 \le j \le n, B[i][j] = f(i, j, E[i])$$

- $E[j]$ 是 矩陣 B 第 j 列的列最小元素。
- 注意：B 只在 對角線之上 有意義。
- 注意：B 是 在線 (online) 的。

1D-1D Problems

$$E[j] = \min_{0 \le i < j} \{ \quad \text{B[i][j]} \quad \},$$

$$\forall\, 1 \le j \le n, B[i][j] = f(i, j, E[i])$$

- $E[j]$ 是 矩陣 B 第 j 列的列最小元素。
- 注意：B 只在 對角線之上 有意義。
- 注意：B 是 在線 (online) 的。

1D-1D Problems | 凹單調

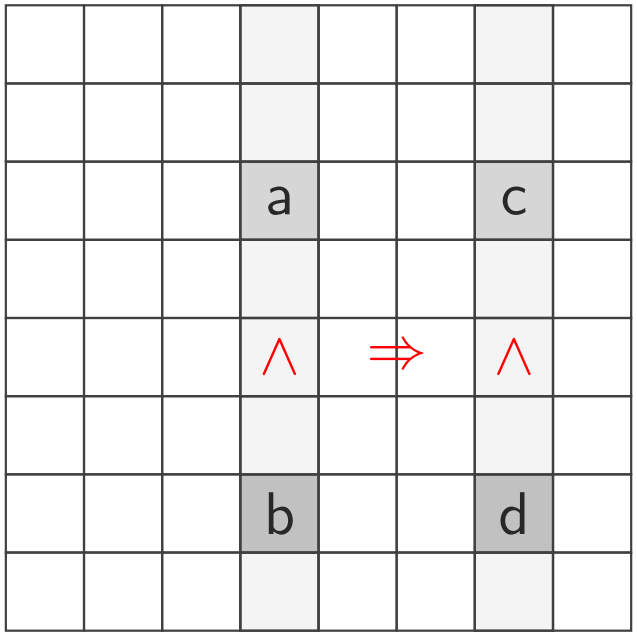


Figure: 凹單調

1D-1D Problems | 凹單調

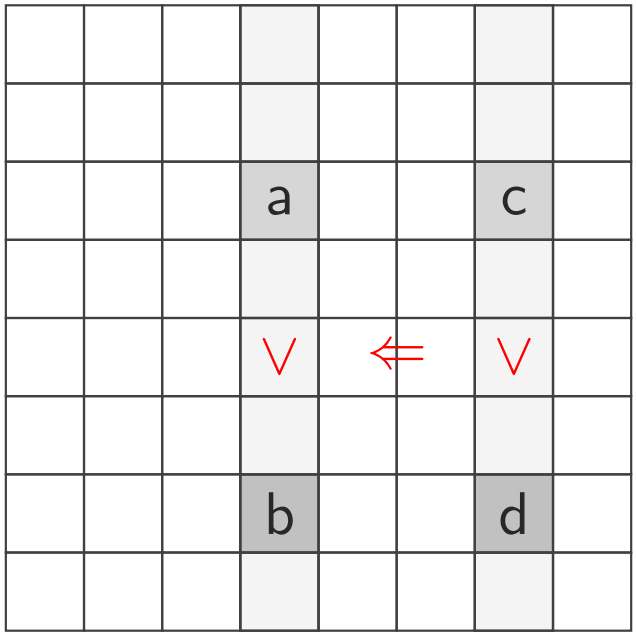


Figure: 凹單調

1D-1D Problems | 凹單調

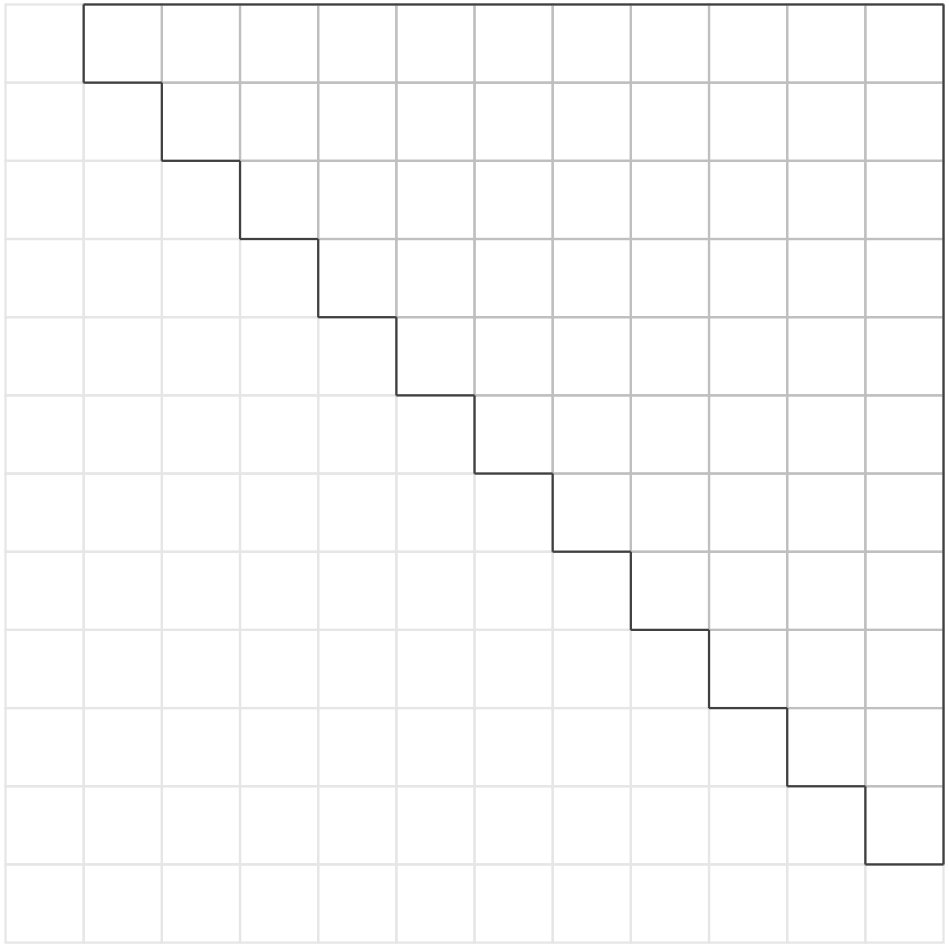


Figure: 凹單調對 B 的影響

1D-1D Problems | 凹單調

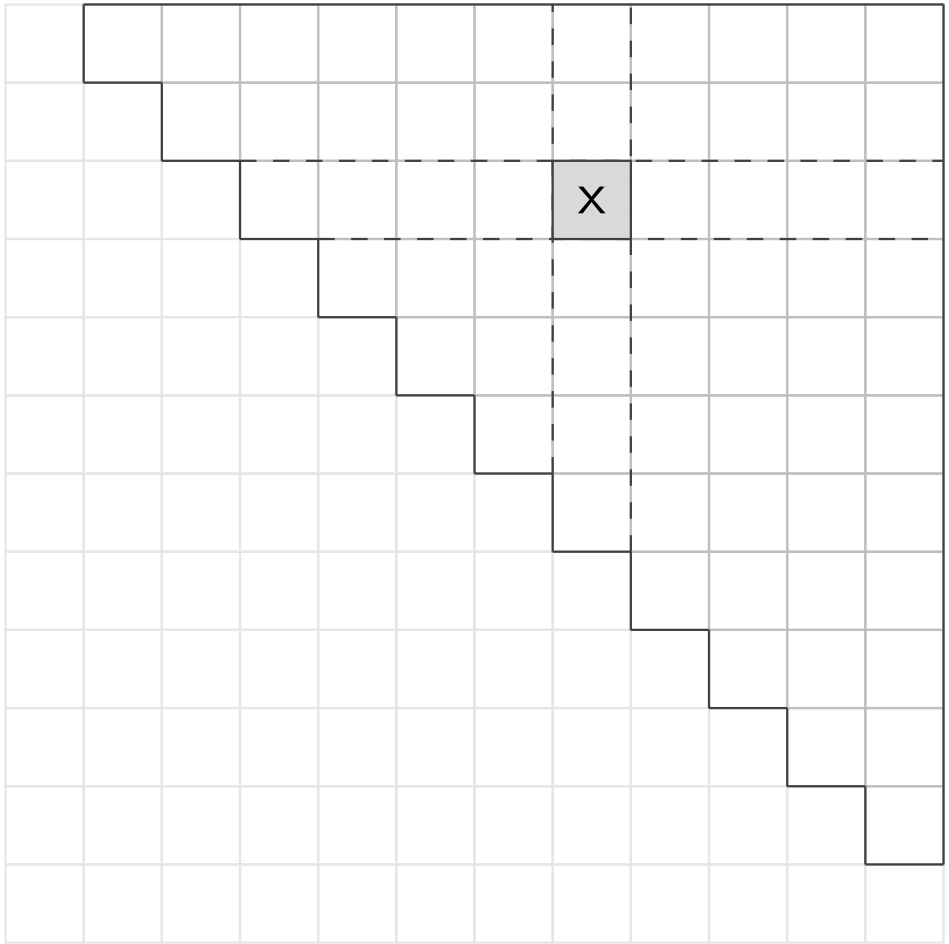


Figure: 凹單調對 B 的影響

1D-1D Problems | 凹單調

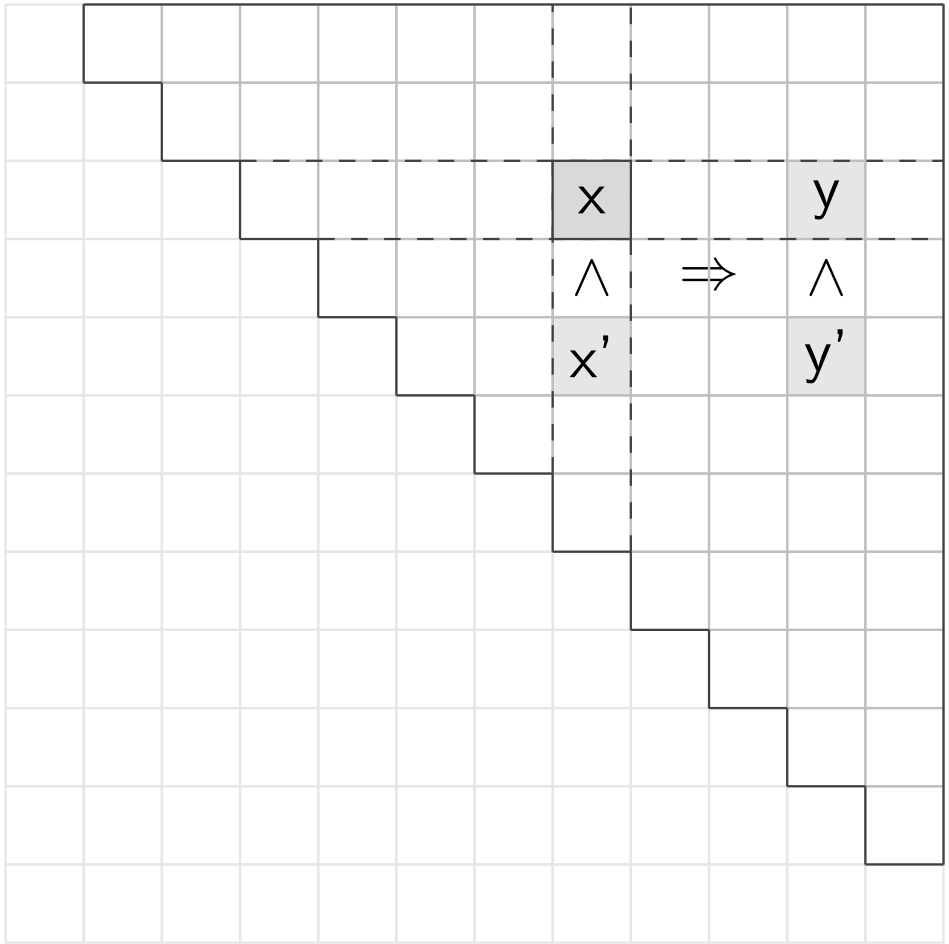


Figure: 凹單調對 B 的影響

1D-1D Problems | 凹單調

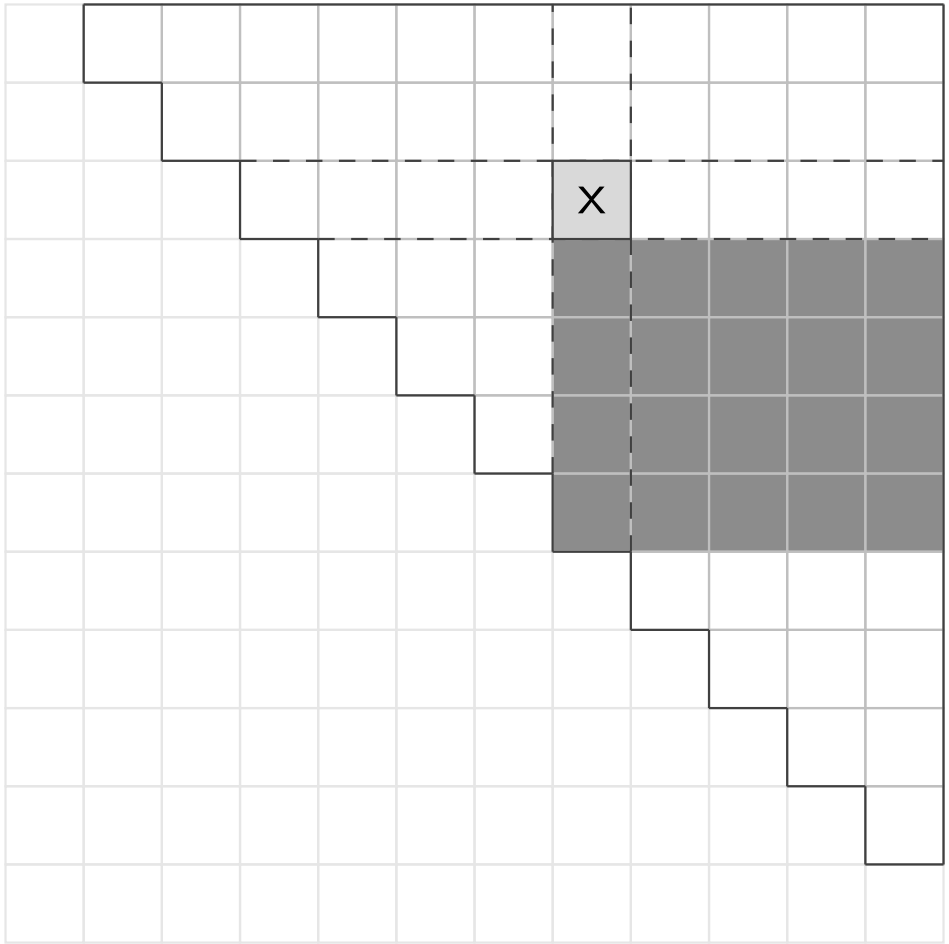


Figure: 凹單調對 B 的影響

1D-1D Problems | 凹單調

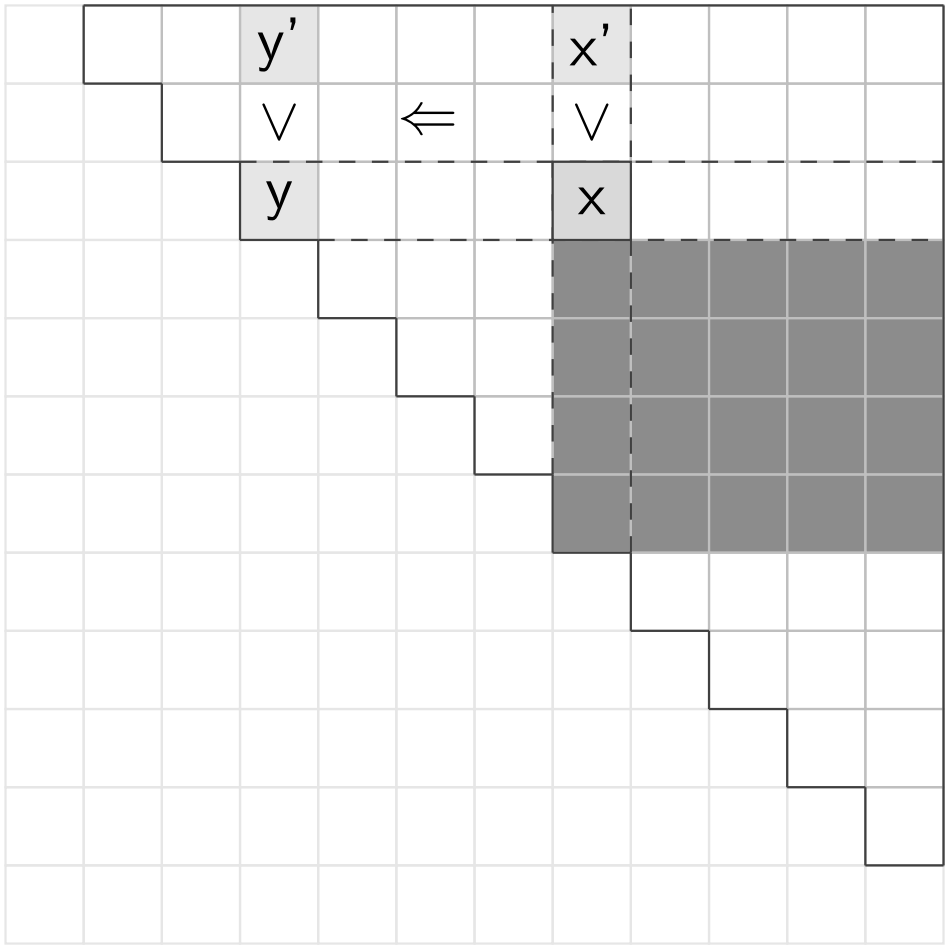


Figure: 凹單調對 B 的影響

1D-1D Problems | 凹單調

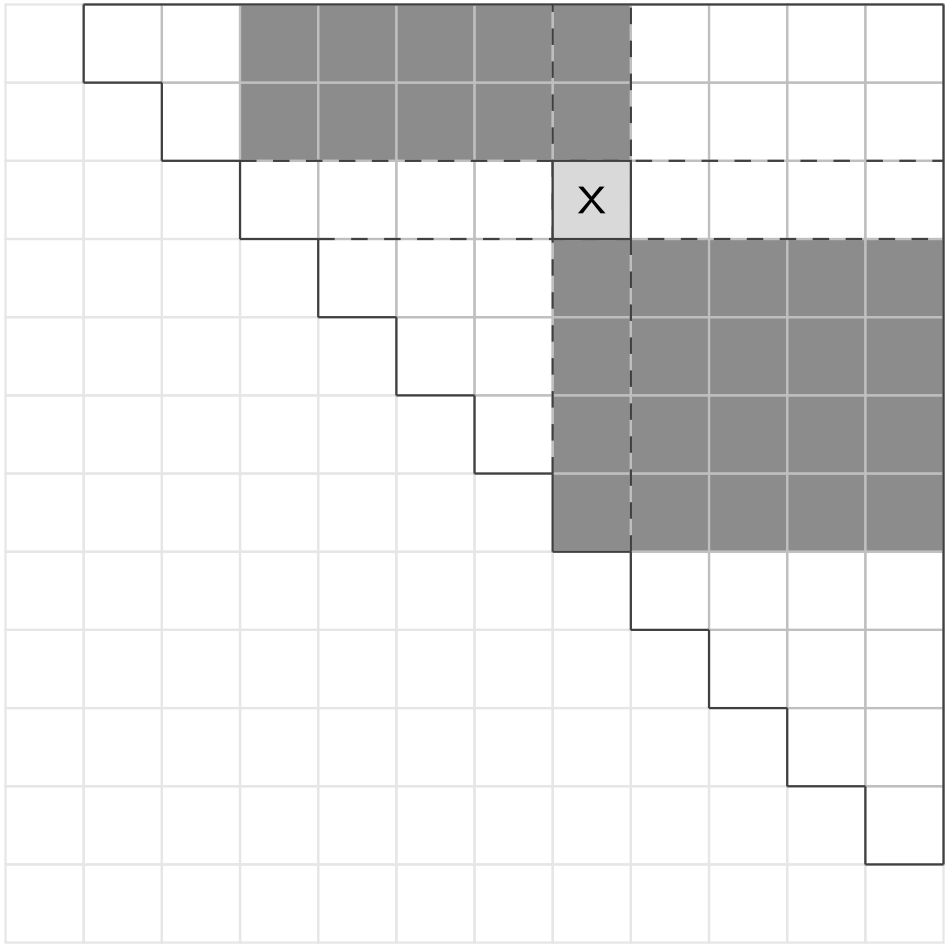


Figure: 凹單調對 B 的影響

1D-1D Problems | 凹單調

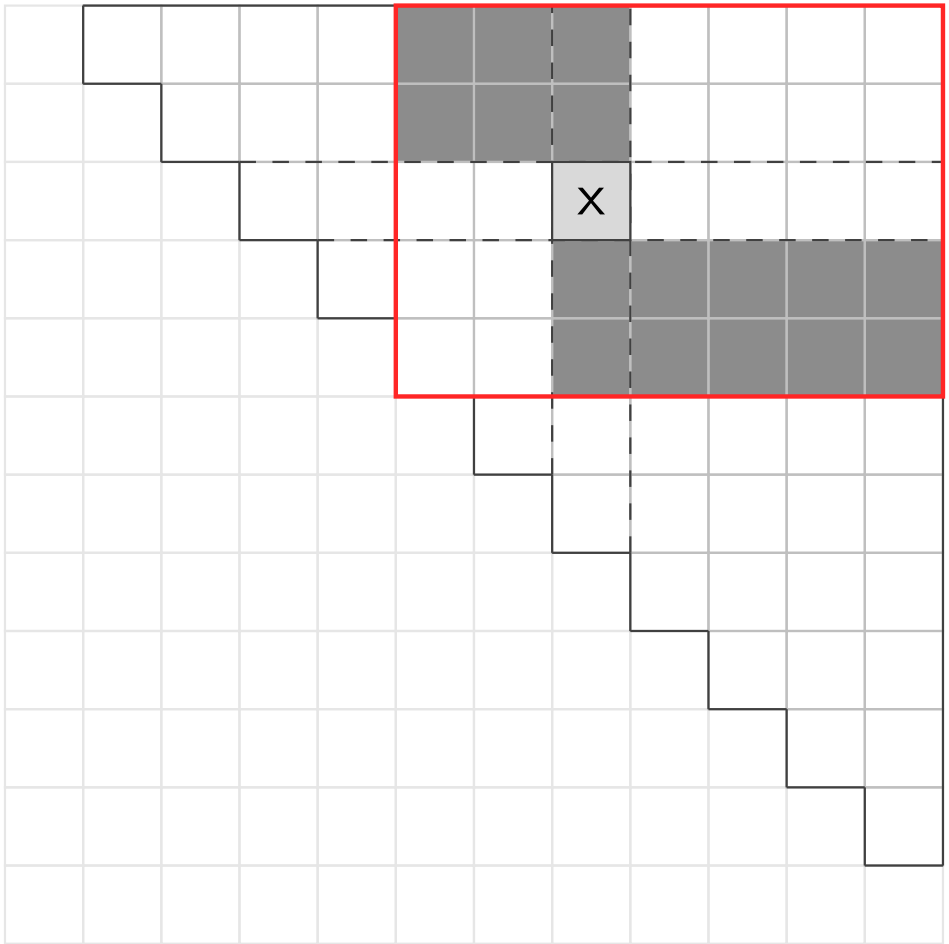


Figure: 凹單調對 B 的影響

1D-1D Problems | 凹單調

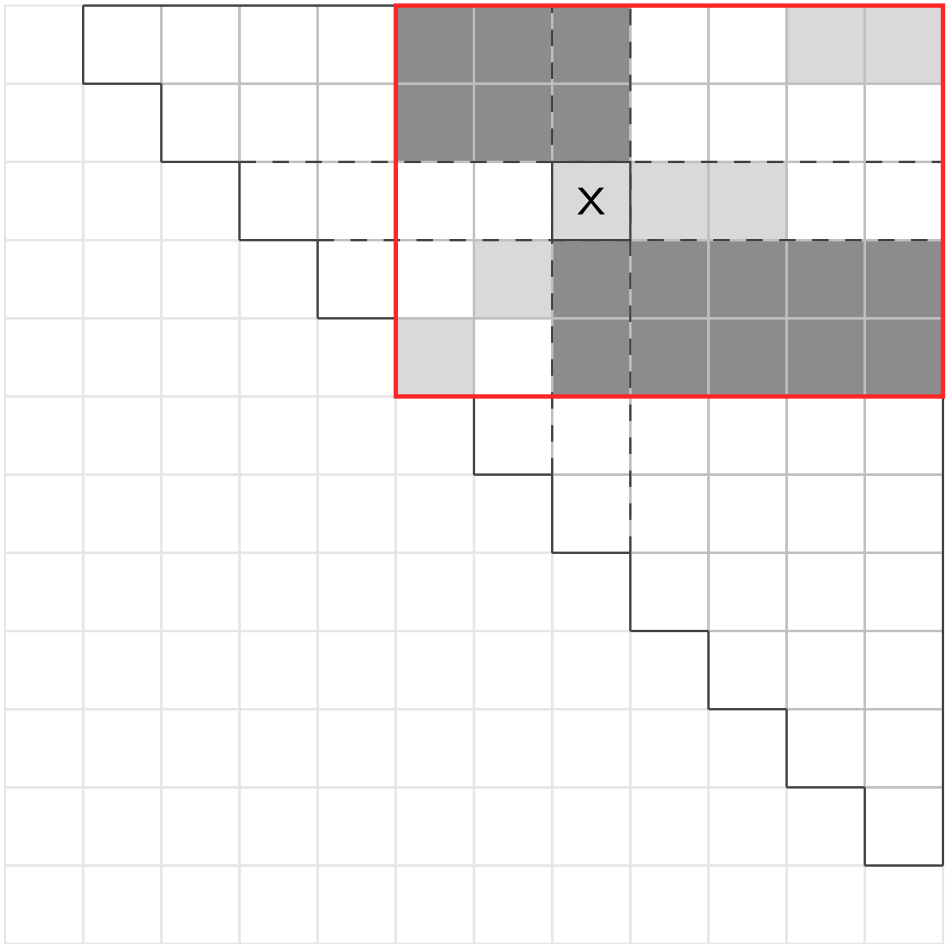


Figure: 凹單調對 B 的影響

1D-1D Problems | 凹單調

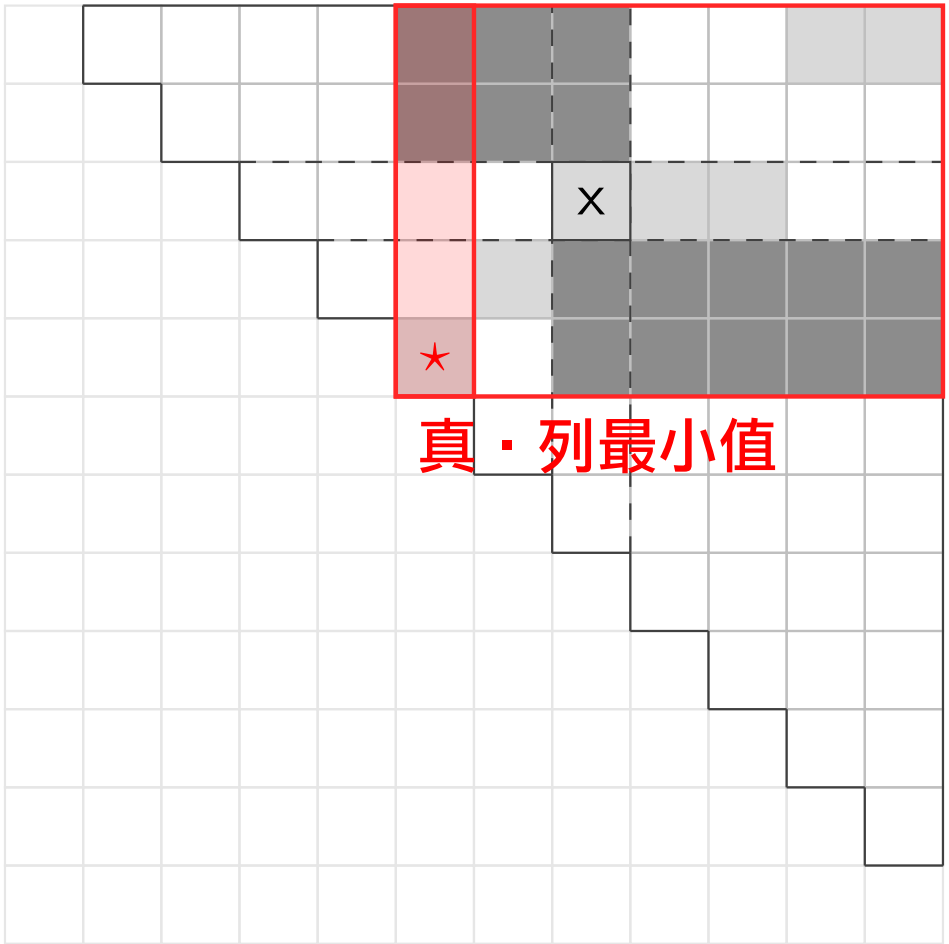


Figure: 凹單調對 B 的影響

1D-1D Problems | 凹單調

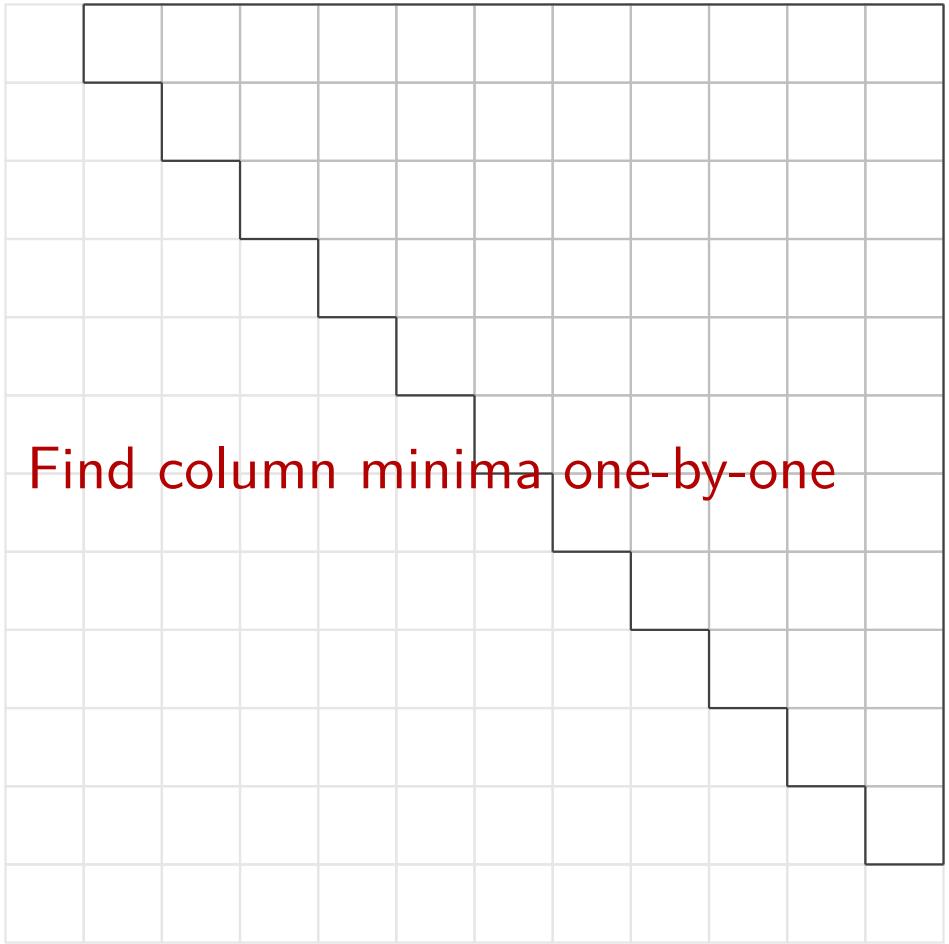


Figure: Finding column minima

1D-1D Problems | 凹單調

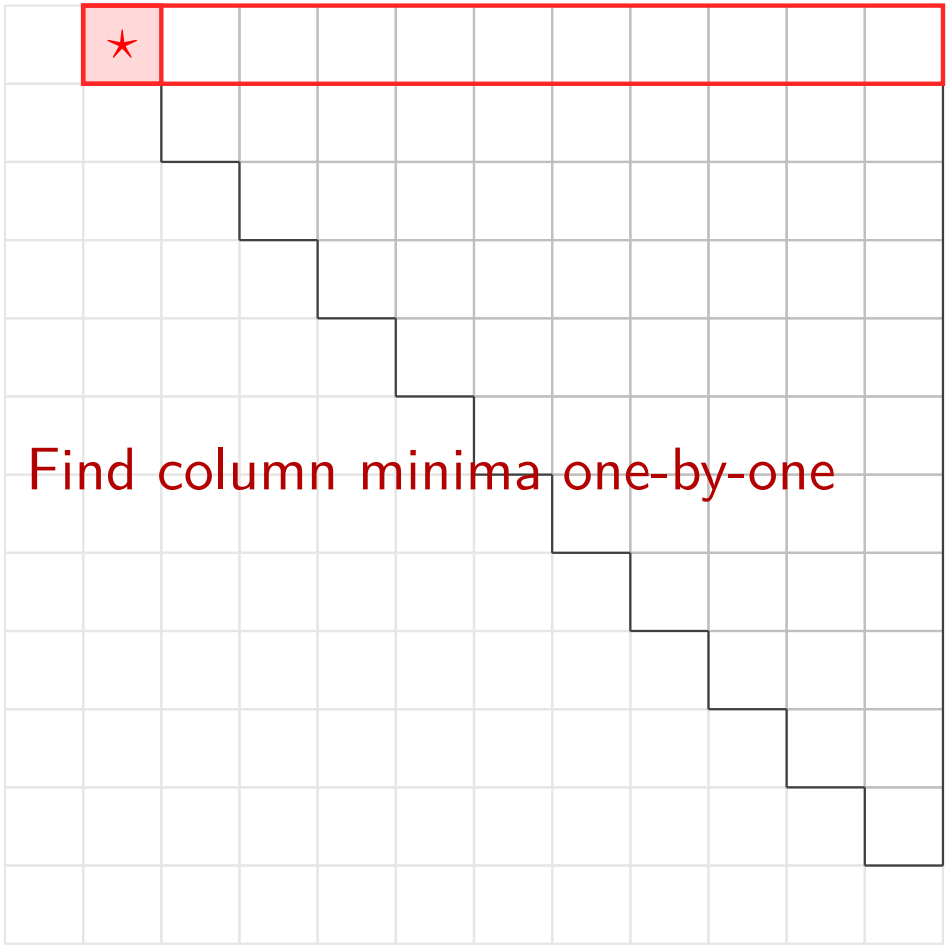


Figure: Finding column minima

1D-1D Problems | 凹單調

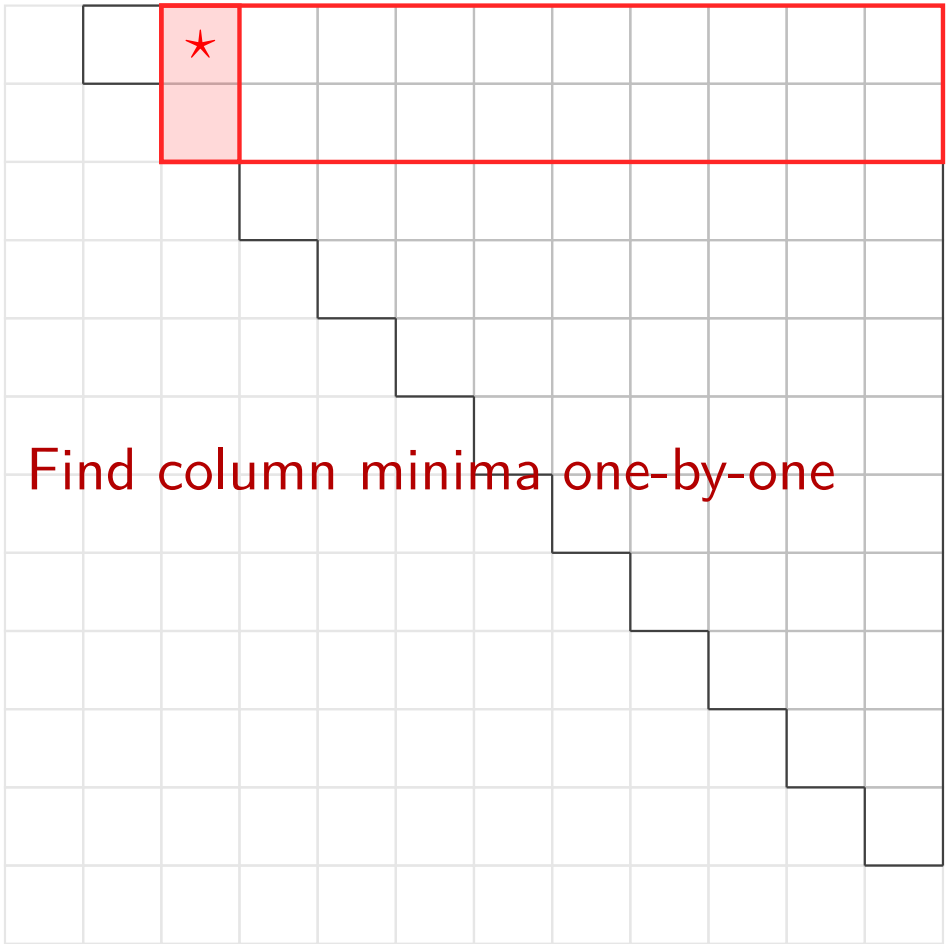


Figure: Finding column minima

1D-1D Problems | 凹單調

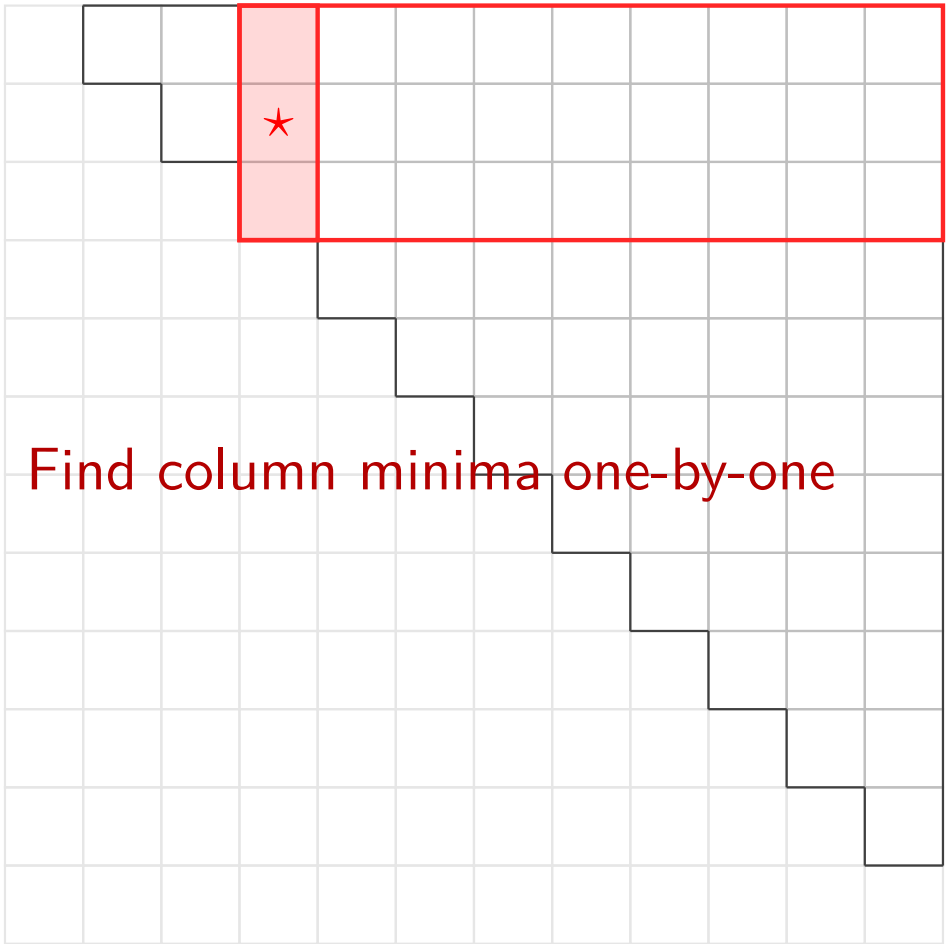


Figure: Finding column minima

1D-1D Problems | 凹單調

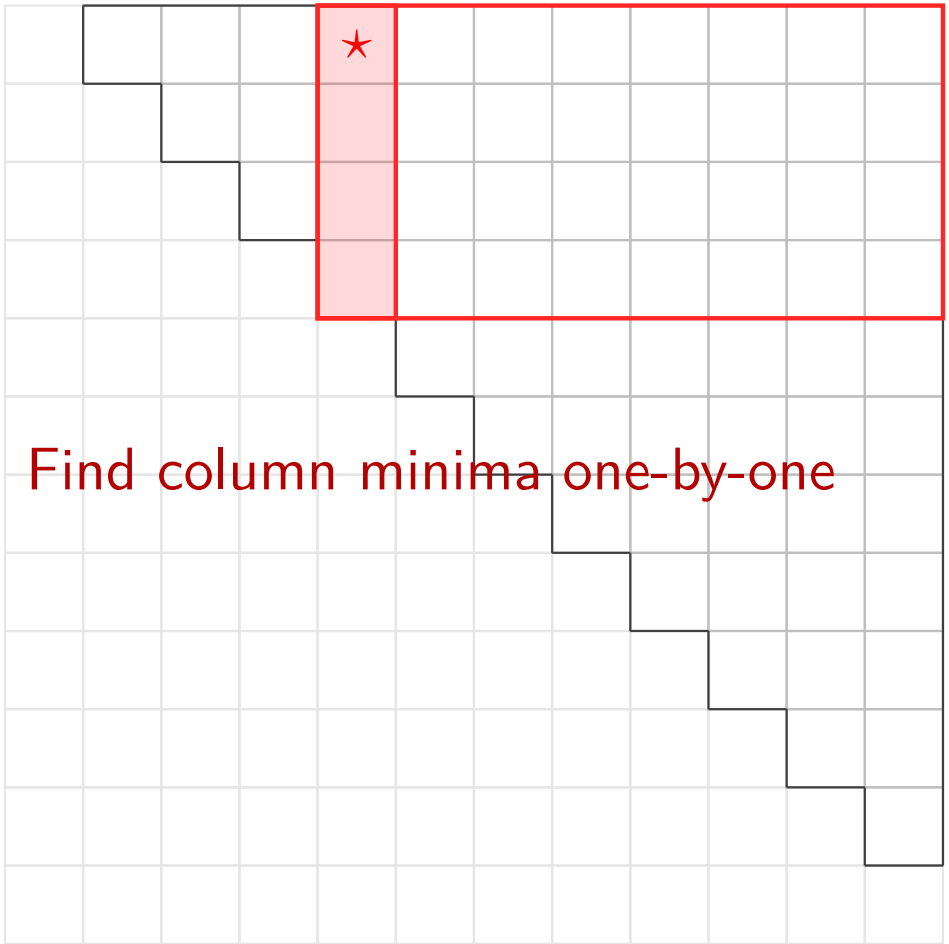


Figure: Finding column minima

1D-1D Problems | 凹單調

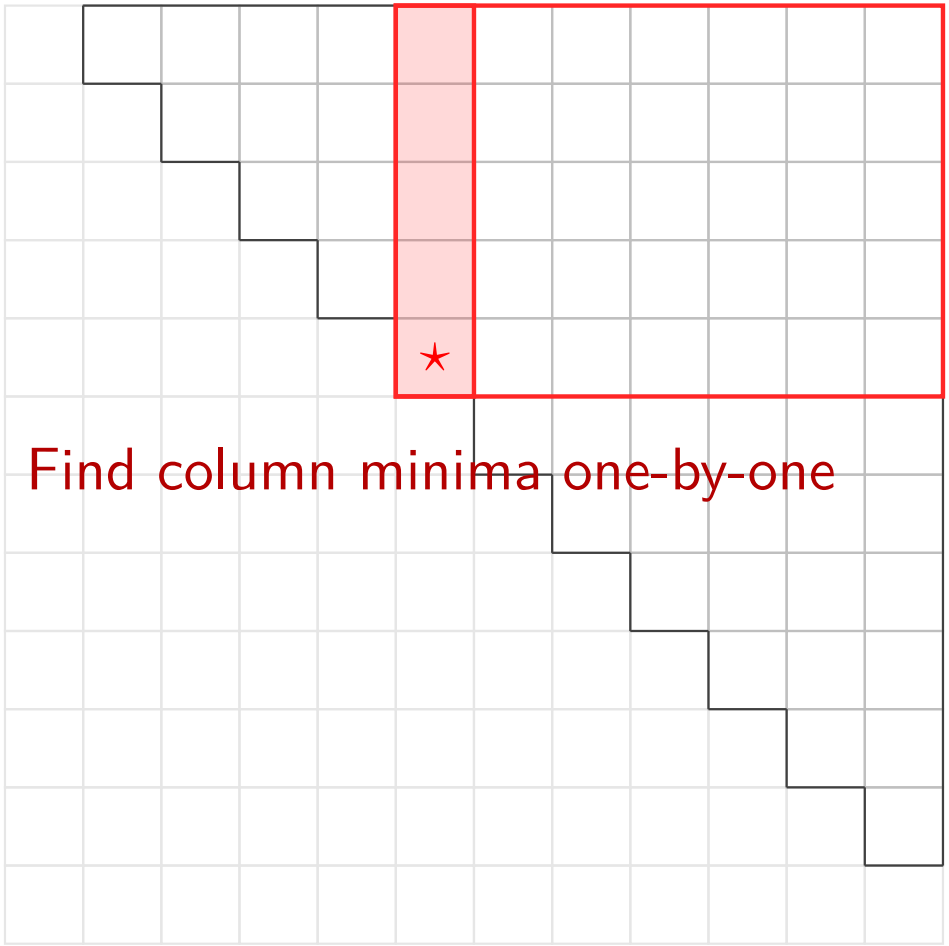


Figure: Finding column minima

1D-1D Problems | 凹單調

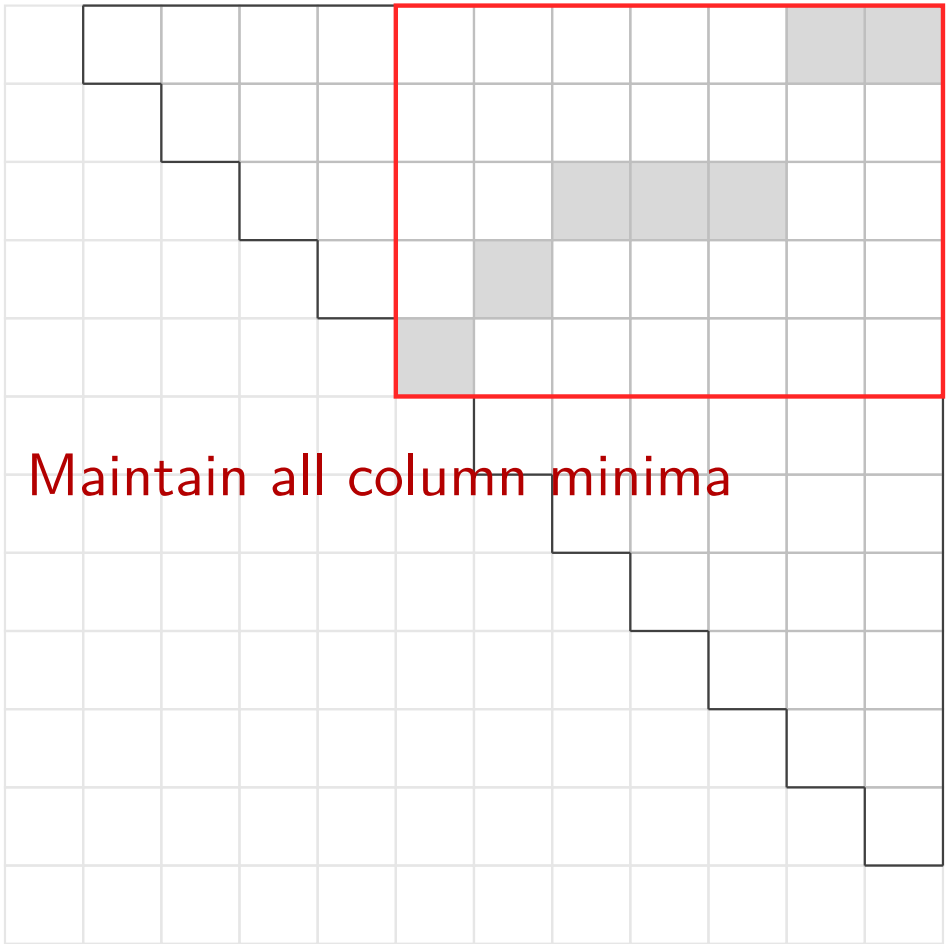


Figure: Finding column minima

1D-1D Problems | 凹單調

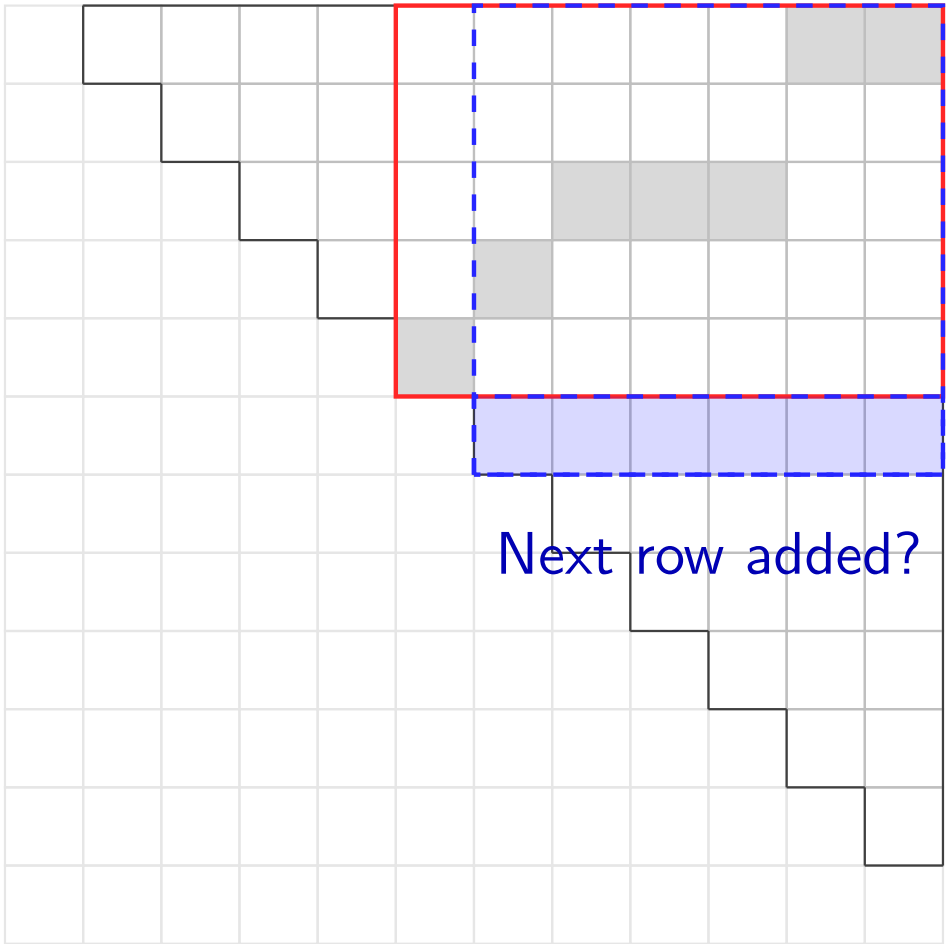


Figure: Finding column minima

1D-1D Problems | 凹單調

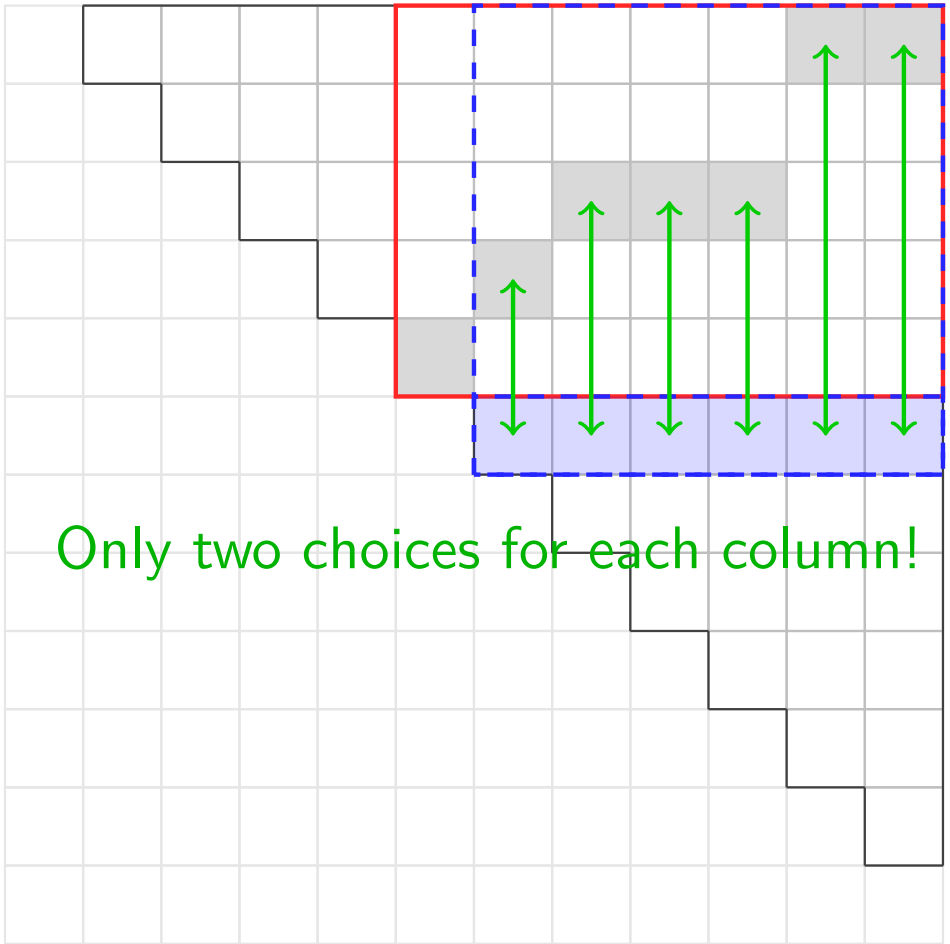


Figure: Finding column minima

1D-1D Problems | 凹單調

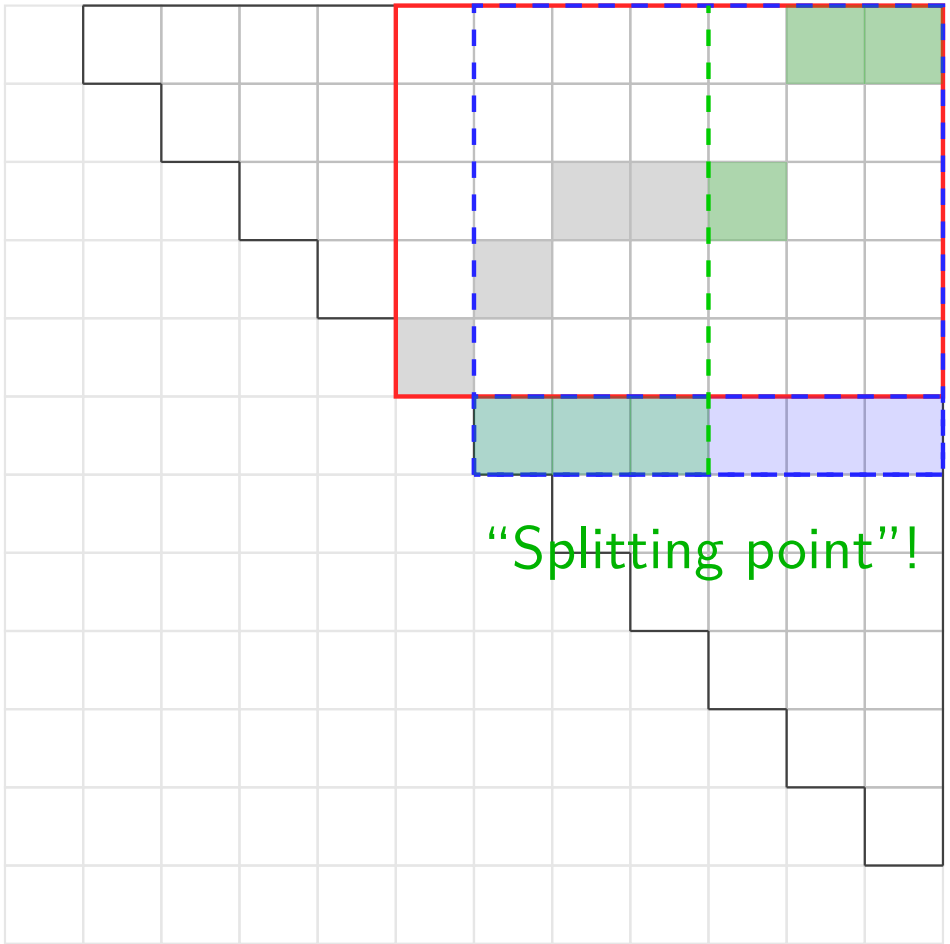


Figure: Finding column minima

1D-1D Problems | 凹單調

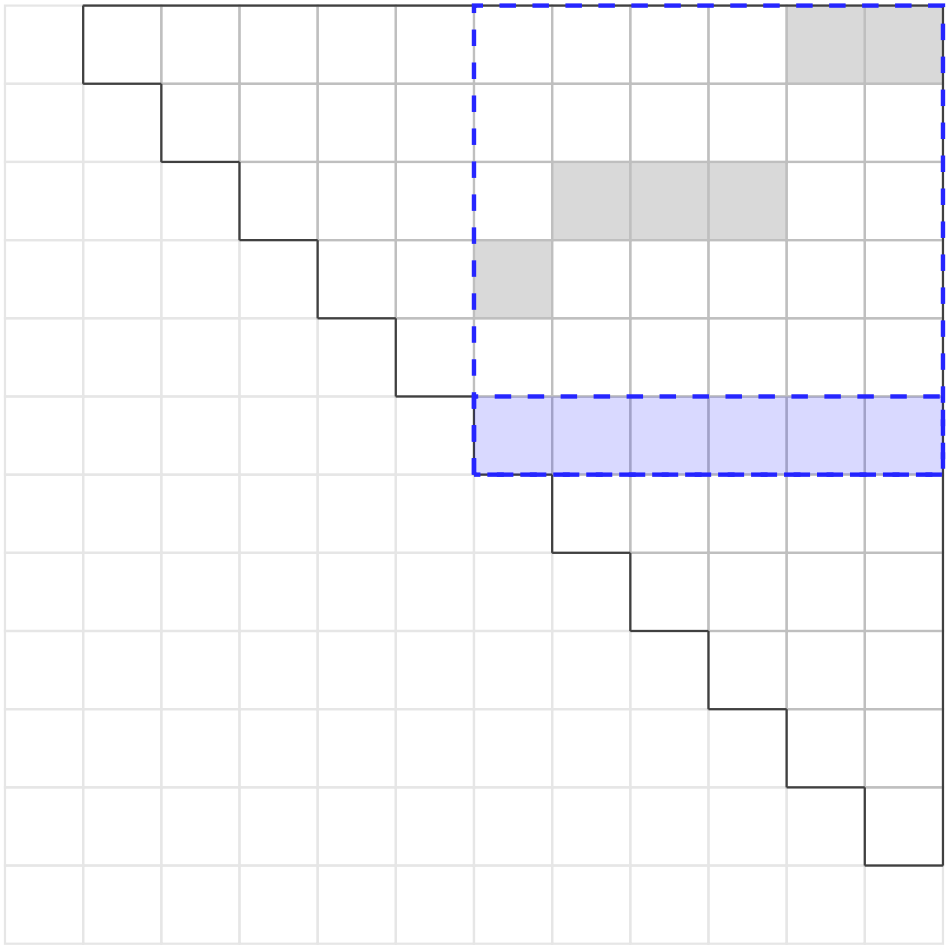


Figure: Finding column minima

1D-1D Problems | 凹單調

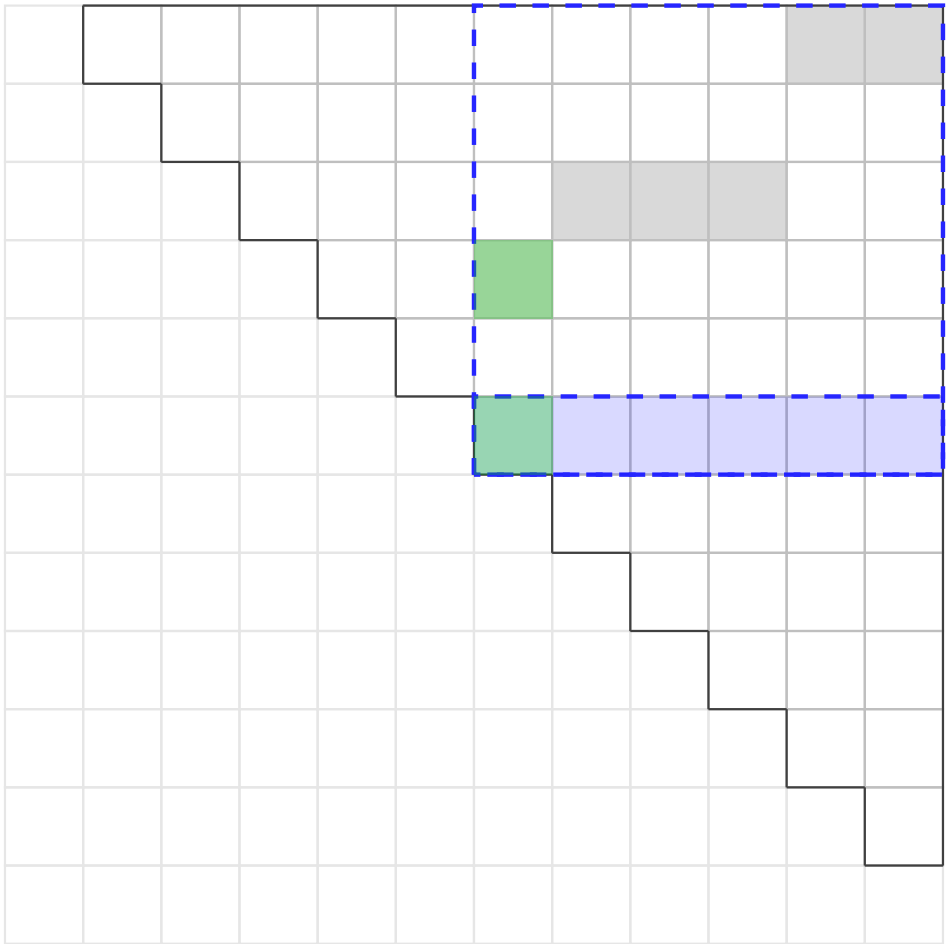


Figure: Finding column minima

1D-1D Problems | 凹單調

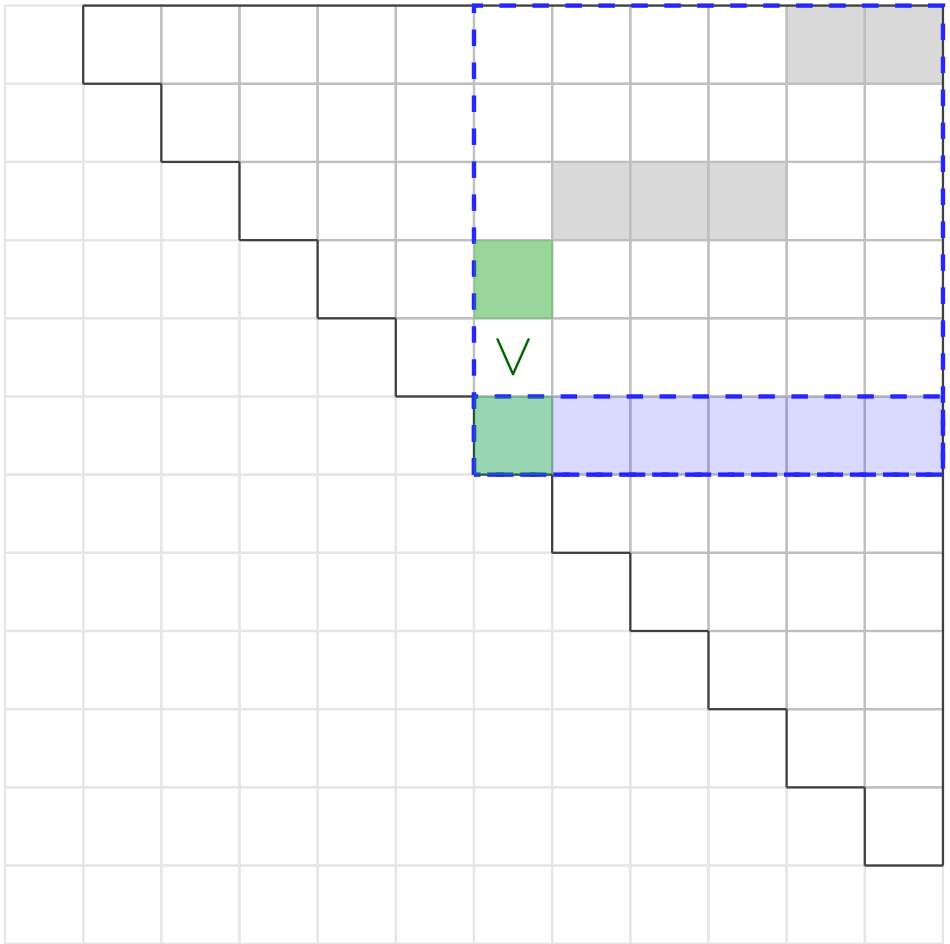


Figure: Finding column minima

1D-1D Problems | 凹單調

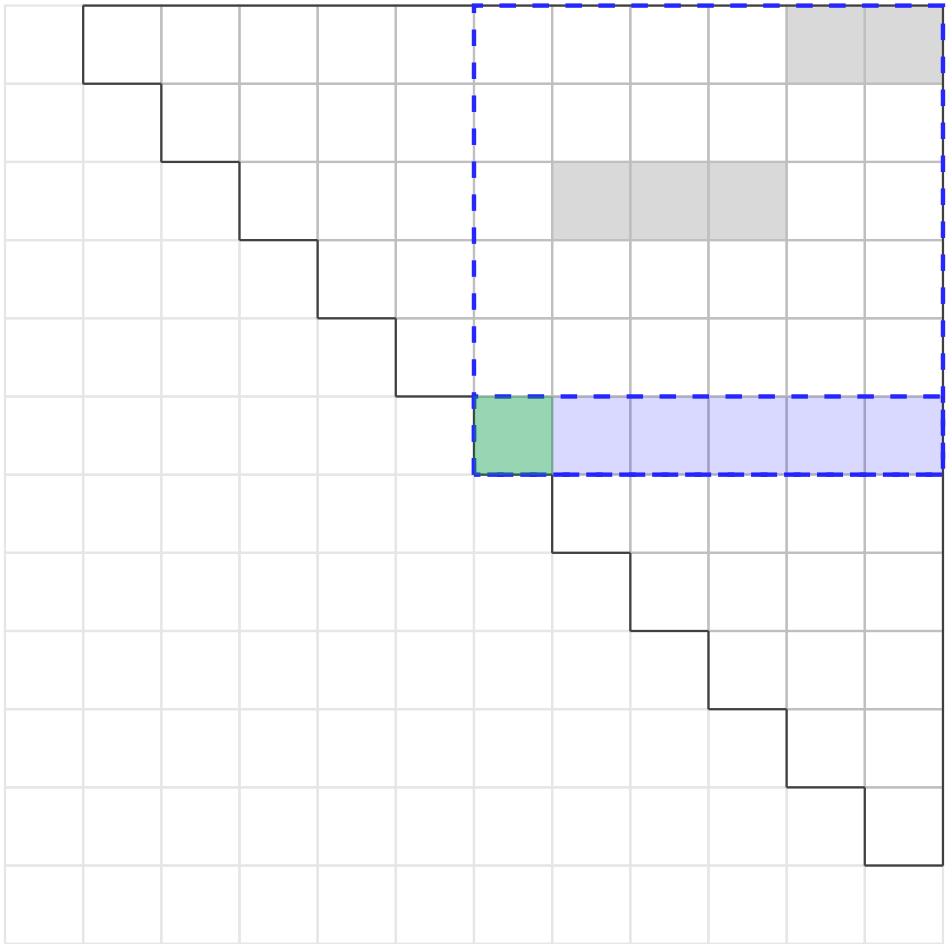


Figure: Finding column minima

1D-1D Problems | 凹單調

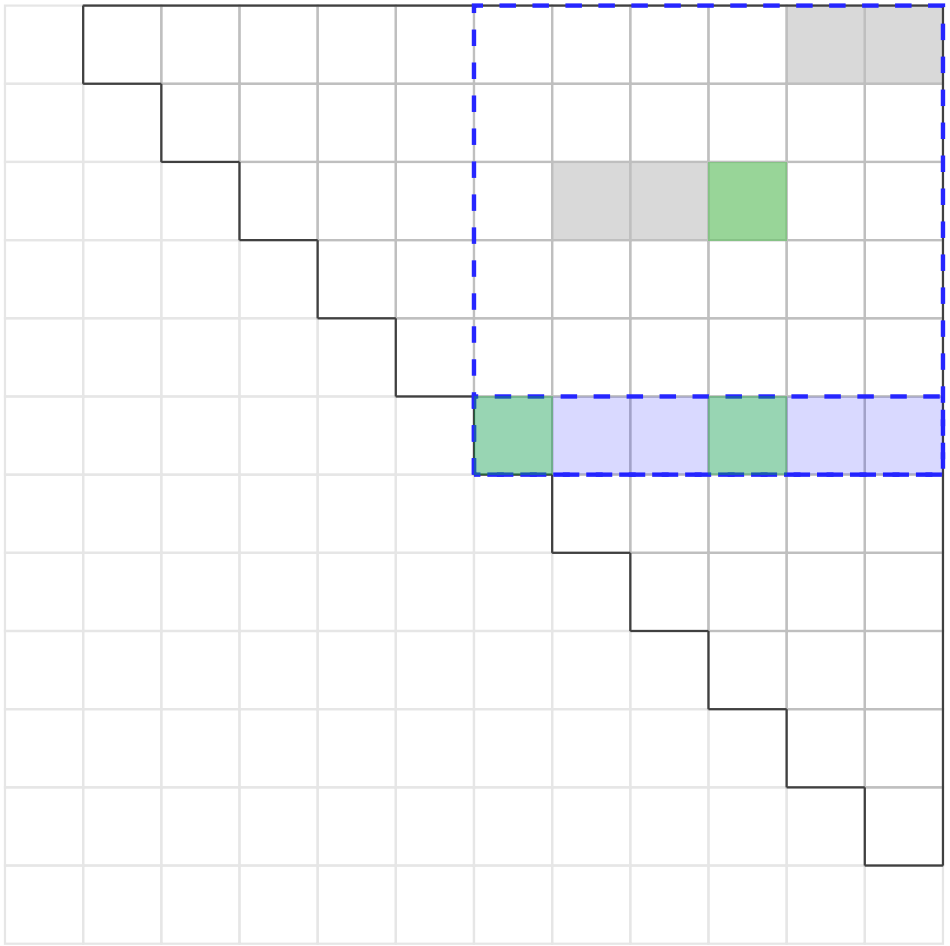


Figure: Finding column minima

1D-1D Problems | 凹單調

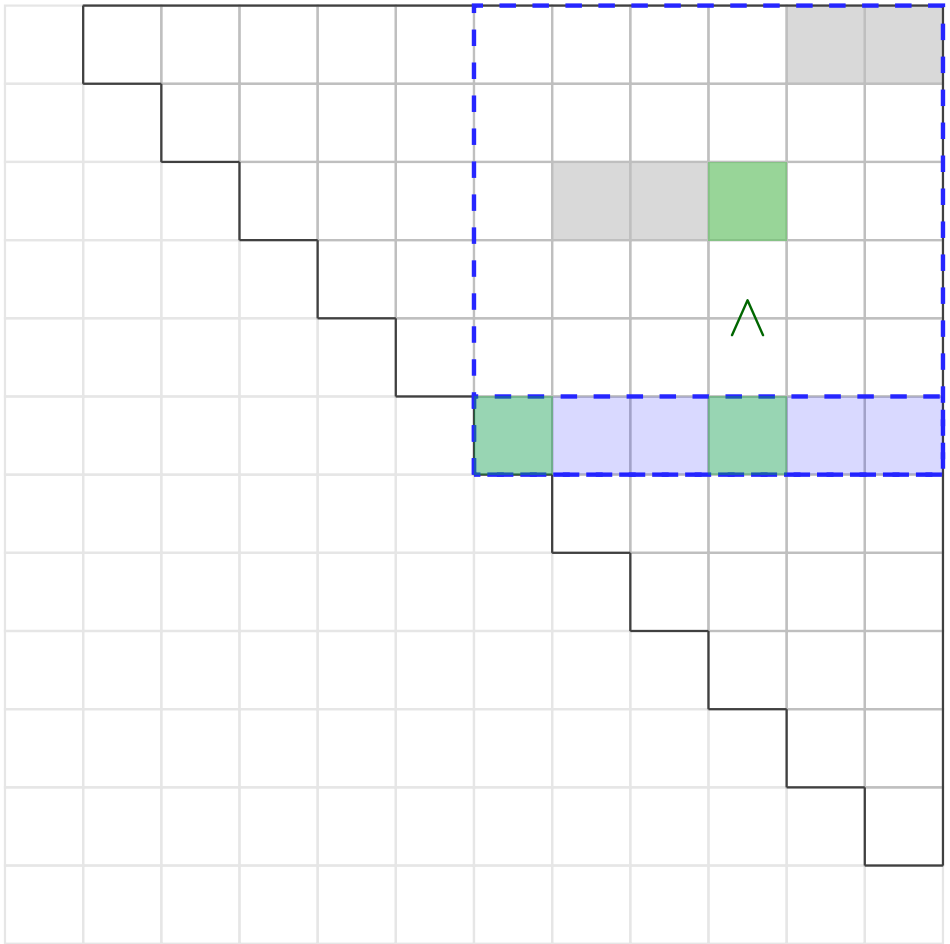


Figure: Finding column minima

1D-1D Problems | 凹單調

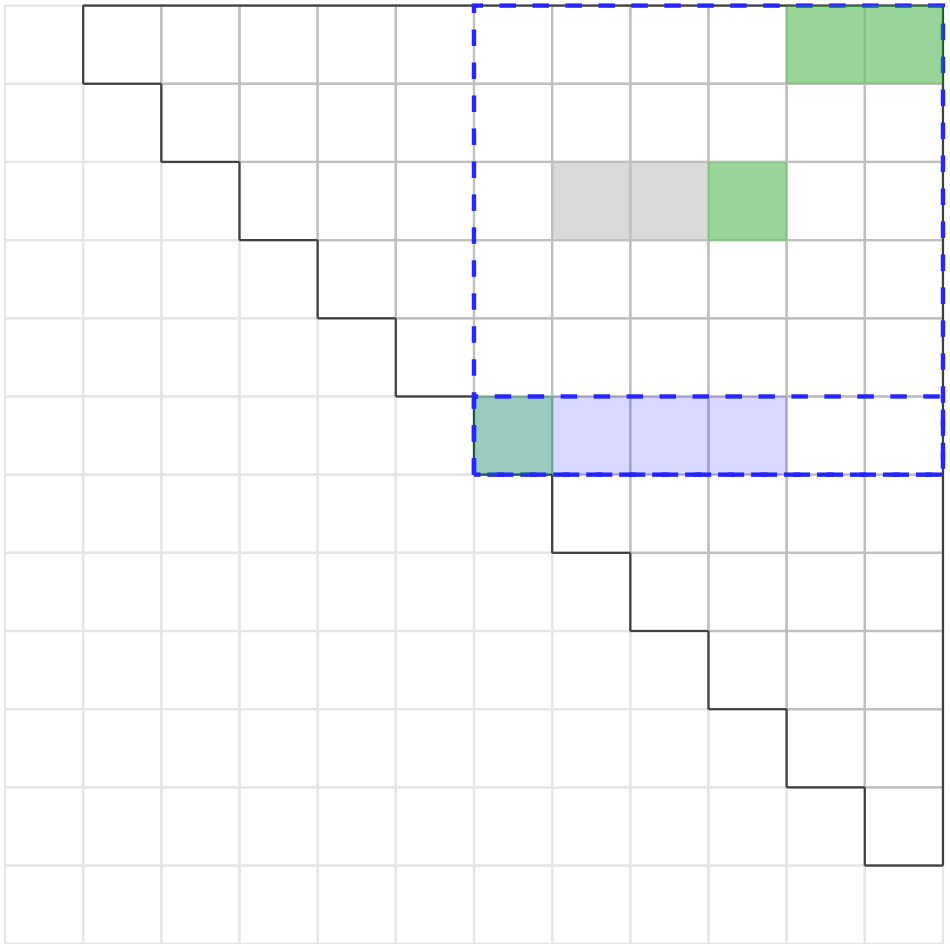


Figure: Finding column minima

1D-1D Problems | 凹單調

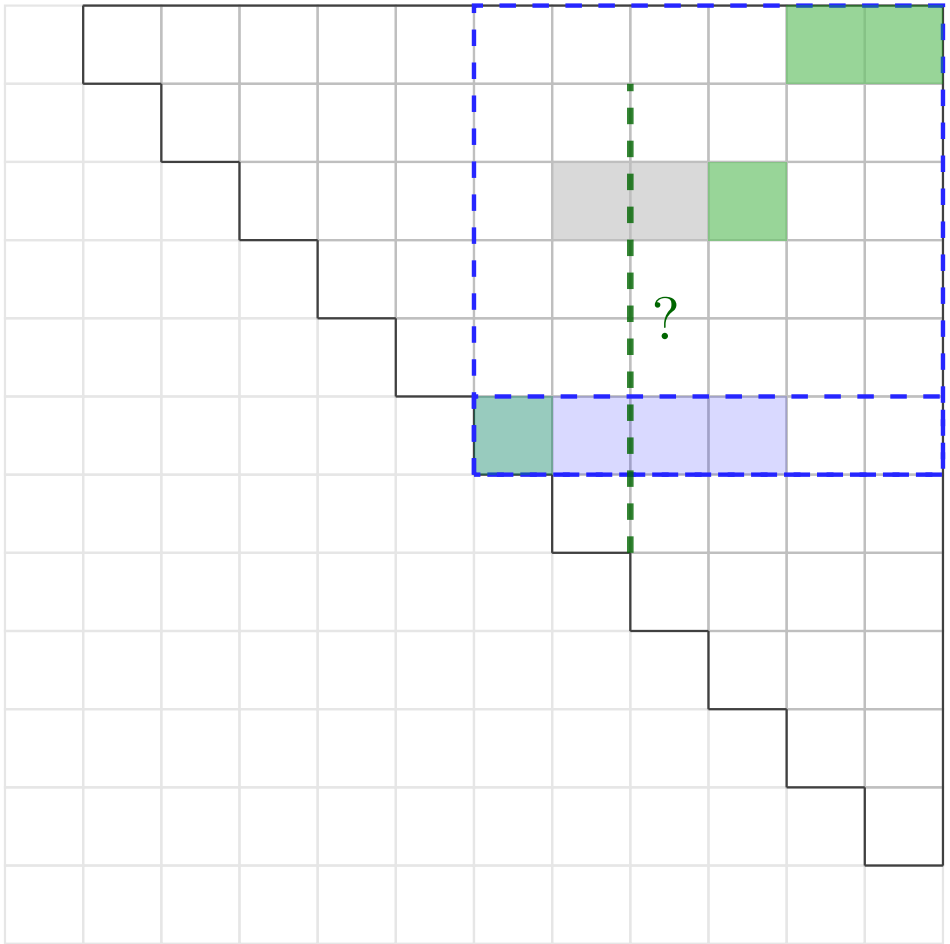


Figure: Finding column minima

1D-1D Problems | 凹單調

Algorithm 2: Concave 1D/1D DP

```
1  function Concave-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← min{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; c ... h_r]
9          S.push([j − 1; j ... c − 1])
10         E[j] ← S.top().first-element
11     end
12 end
```

1D-1D Problems | 凹單調

Algorithm 2: Concave 1D/1D DP

```
1  function Concave-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← min{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; c ... h_r]
9          S.push([j − 1; j ... c − 1])
10         E[j] ← S.top().first-element
11     end
12 end
```

找到第一個 (最左) 不完全比新的一行差的行區間 r 。

1D-1D Problems | 凹單調

Algorithm 2: Concave 1D/1D DP

```
1  function Concave-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← min{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; c ... h_r]
9          S.push([j − 1; j ... c − 1])
10         E[j] ← S.top().first-element
11     end
12 end
```

在行區間 r 以前的所有區間都「掰了」。

1D-1D Problems | 凹單調

Algorithm 2: Concave 1D/1D DP

```
1  function Concave-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j-1][h_t]}
6          pop from S all segments before r-th
7          c ← min{t | B[i_r][t] < B[j-1][t]}
8          S.top() ← [i_r; c ... h_r]
9          S.push([j-1; j ... c-1])
10         E[j] ← S.top().first-element
11     end
12 end
```

對於行區間 r，二分搜找出新區間比較好的分界。

1D-1D Problems | 凹單調

Algorithm 2: Concave 1D/1D DP

```
1  function Concave-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← min{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; c ... h_r]
9          S.push([j − 1; j ... c − 1])
10         E[j] ← S.top().first-element
11     end
12 end
```

更新 stack 。

1D-1D Problems | 凹單調

Algorithm 2: Concave 1D/1D DP

```
1  function Concave-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← min{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; c ... h_r]
9          S.push([j − 1; j ... c − 1])
10         E[j] ← S.top().first-element
11     end
12 end
```

紀錄列最小值：$E[j]$。

1D-1D Problems | 凹單調

回顧一下我們做了什麼：

- **給一個單調 1D-1D 的 DP 問題，將其轉化為找尋線上 (online) 單調二維矩陣 B 的列最小值。**
- 若一個元素 x 是 B 某列的列最小值，那麼其左上與右下的元素都「掰了」。
- 因此，一個 B 的子矩形中列最小值由左下往右上分布。
- 我們 (以行區間形式) 用 stack 維護子矩形中所有列最小值：起始只有 B 的第一行，之後依序加入新的行並更新 stack。
- 加入更新的第一步：找到第一個需被保留的行區間。
- 加入更新的第二步：二分搜該區間與新的行區間的分界。

1D-1D Problems | 凹單調

回顧一下我們做了什麼：

- 給一個單調 1D-1D 的 DP 問題，將其轉化為找尋線上 (online) 單調二維矩陣 B 的列最小值。
- 若一個元素 x 是 B 某列的列最小值，那麼其左上與右下的元素都「掰了」。
- 因此，一個 B 的子矩形中列最小值由左下往右上分布。
- 我們 (以行區間形式) 用 stack 維護子矩形中所有列最小值：起始只有 B 的第一行，之後依序加入新的行並更新 stack。
- 加入更新的第一步：找到第一個需被保留的行區間。
- 加入更新的第二步：二分搜該區間與新的行區間的分界。

1D-1D Problems | 凹單調

回顧一下我們做了什麼：

- 給一個單調 1D-1D 的 DP 問題，將其轉化為找尋線上 (online) 單調二維矩陣 B 的列最小值。
- 若一個元素 x 是 B 某列的列最小值，那麼其左上與右下的元素都「掰了」。
- 因此，一個 B 的子矩形中列最小值由左下往右上分布。
- 我們 (以行區間形式) 用 stack 維護子矩形中所有列最小值：起始只有 B 的第一行，之後依序加入新的行並更新 stack。
- 加入更新的第一步：找到第一個需被保留的行區間。
- 加入更新的第二步：二分搜該區間與新的行區間的分界。

1D-1D Problems | 凹單調

回顧一下我們做了什麼：

- 給一個單調 1D-1D 的 DP 問題，將其轉化為找尋線上 (online) 單調二維矩陣 B 的列最小值。
- 若一個元素 x 是 B 某列的列最小值，那麼其左上與右下的元素都「掰了」。
- 因此，一個 B 的子矩形中列最小值由左下往右上分布。
- 我們 (以行區間形式) 用 stack 維護子矩形中所有列最小值：起始只有 B 的第一行，之後依序加入新的行並更新 stack。
- 加入更新的第一步：找到第一個需被保留的行區間。
- 加入更新的第二步：二分搜該區間與新的行區間的分界。

1D-1D Problems | 凹單調

回顧一下我們做了什麼：

- 給一個單調 1D-1D 的 DP 問題，將其轉化為找尋線上 (online) 單調二維矩陣 B 的列最小值。
- 若一個元素 x 是 B 某列的列最小值，那麼其左上與右下的元素都「掰了」。
- 因此，一個 B 的子矩形中列最小值由左下往右上分布。
- 我們 (以行區間形式) 用 stack 維護子矩形中所有列最小值：起始只有 B 的第一行，之後依序加入新的行並更新 stack。
- 加入更新的第一步：找到第一個需被保留的行區間。
- 加入更新的第二步：二分搜該區間與新的行區間的分界。

1D-1D Problems | 凹單調

回顧一下我們做了什麼：

- 給一個單調 1D-1D 的 DP 問題，將其轉化為找尋線上 (online) 單調二維矩陣 B 的列最小值。
- 若一個元素 x 是 B 某列的列最小值，那麼其左上與右下的元素都「掰了」。
- 因此，一個 B 的子矩形中列最小值由左下往右上分布。
- 我們 (以行區間形式) 用 stack 維護子矩形中所有列最小值：起始只有 B 的第一行，之後依序加入新的行並更新 stack。
- 加入更新的第一步：找到第一個需被保留的行區間。
- 加入更新的第二步：二分搜該區間與新的行區間的分界。

1D-1D Problems | 凹單調

時間複雜度？

- **加入更新的第一步：找到第一個需被保留的行區間。**
  $\Rightarrow$ AYCP-stack – 均攤 $O(N)$。
- **加入更新的第二步：二分搜該區間與新的行區間的分界。**
  $\Rightarrow$ 共 $O(\lg N)$ 次決策點比較。
- **因此，我們在 $O(N \lg N)$ 的時間解決了原問題！**

1D-1D Problems | 凹單調

時間複雜度？

- **加入更新的第一步：找到第一個需被保留的行區間。**
  $\Rightarrow$ AYCP-stack – **均攤** $O(N)$。
- **加入更新的第二步：二分搜該區間與新的行區間的分界。**
  $\Rightarrow$ 共 $O(\lg N) = O(\log N)$ 次。
- **因此，我們在** $O(N \lg N)$ **的時間解決了原問題！**

1D-1D Problems | 凹單調

時間複雜度？

- **加入更新的第一步：找到第一個需被保留的行區間。**
  $\Rightarrow$ AYCP-stack – **均攤** $O(N)$。
- **加入更新的第二步：二分搜該區間與新的行區間的分界。**
  $\Rightarrow N \cdot O(\lg N) = O(N \lg N)$。
- **因此，我們在** $O(N \lg N)$ **的時間解決了原問題！**

1D-1D Problems | 凹單調

時間複雜度？

- **加入更新的第一步：找到第一個需被保留的行區間。**
  $\Rightarrow$ AYCP-stack – **均攤** $O(N)$。
- **加入更新的第二步：二分搜該區間與新的行區間的分界。**
  $\Rightarrow N \cdot O(\lg N) = O(N \lg N)$。
- **因此，我們在** $O(N \lg N)$ **的時間解決了原問題！**

1D-1D Problems | 凹單調

時間複雜度？

- **加入更新的第一步：找到第一個需被保留的行區間。**
  $\Rightarrow$ AYCP-stack – **均攤** $O(N)$。
- **加入更新的第二步：二分搜該區間與新的行區間的分界。**
  $\Rightarrow N \cdot O(\lg N) = O(N \lg N)$。
- **因此，我們在** $O(N \lg N)$ **的時間解決了原問題！**

1D-1D Problems | 凸單調

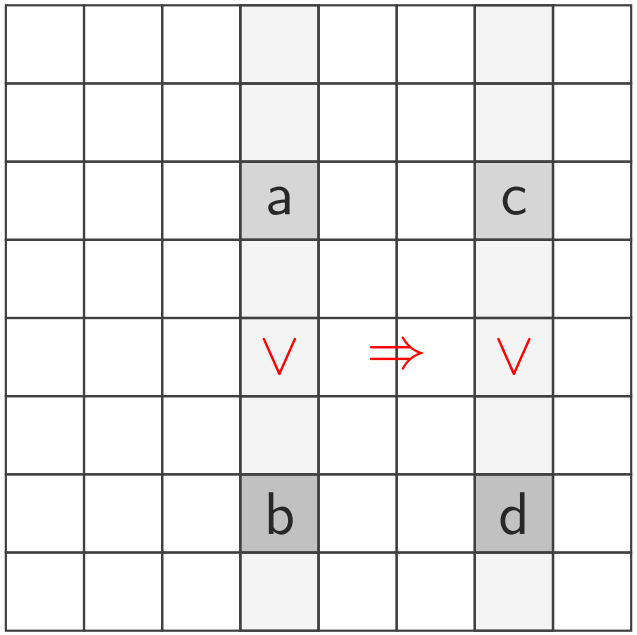


Figure: 凸單調

1D-1D Problems | 凸單調

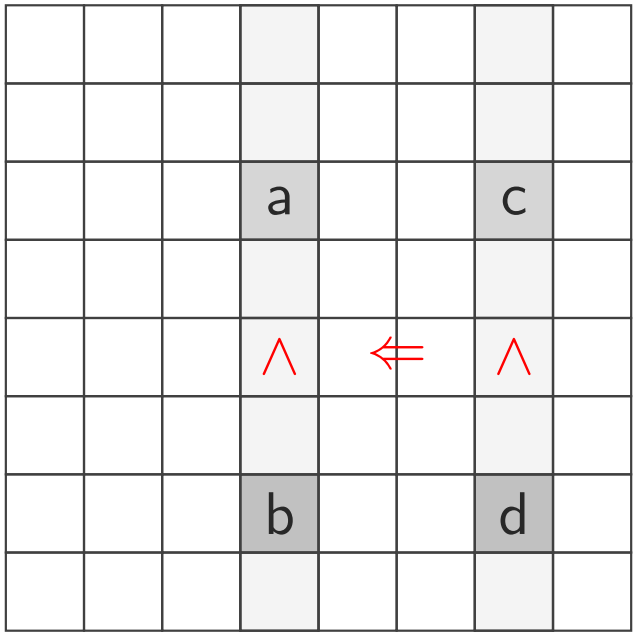


Figure: 凸單調

1D-1D Problems | 凸單調

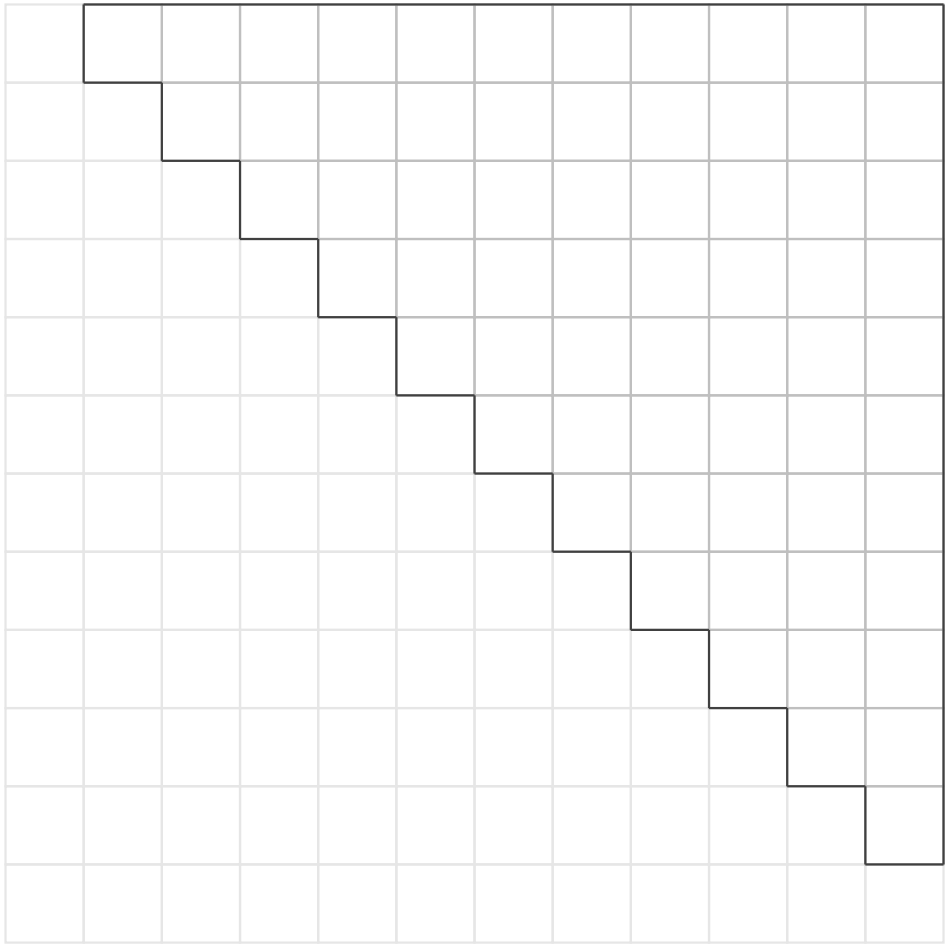


Figure: 凸單調對 B 的影響

1D-1D Problems | 凸單調

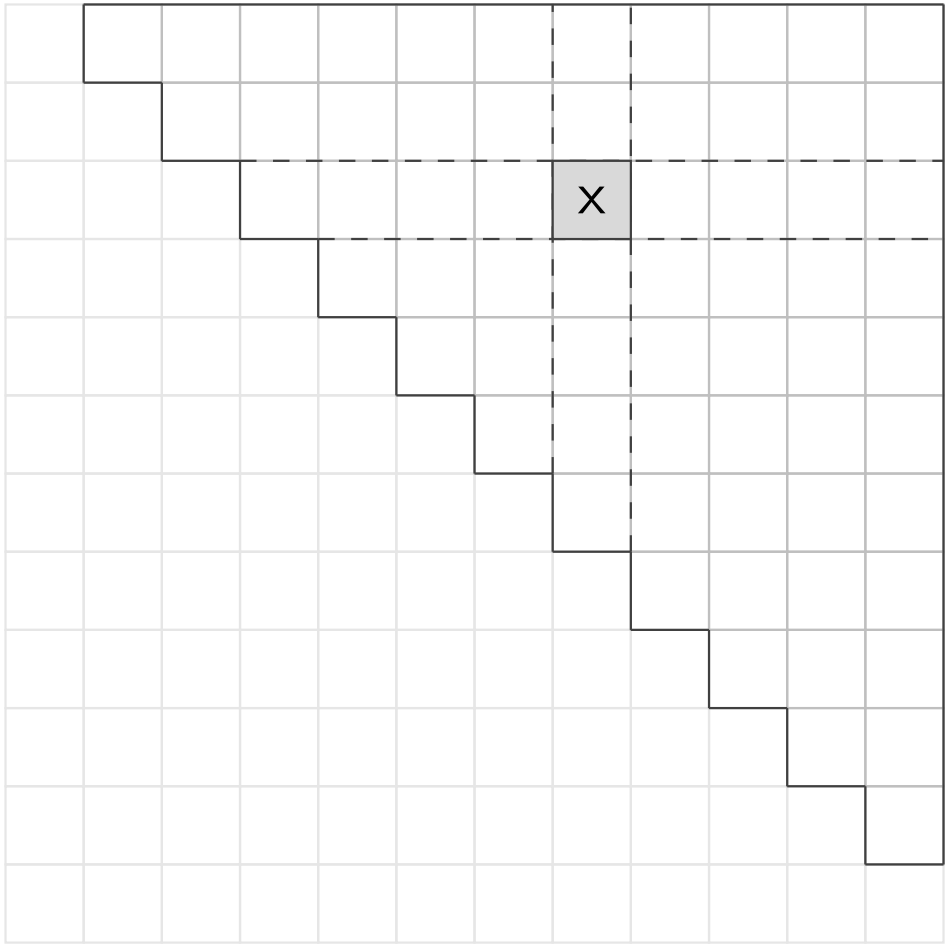


Figure: 凸單調對 B 的影響

1D-1D Problems | 凸單調

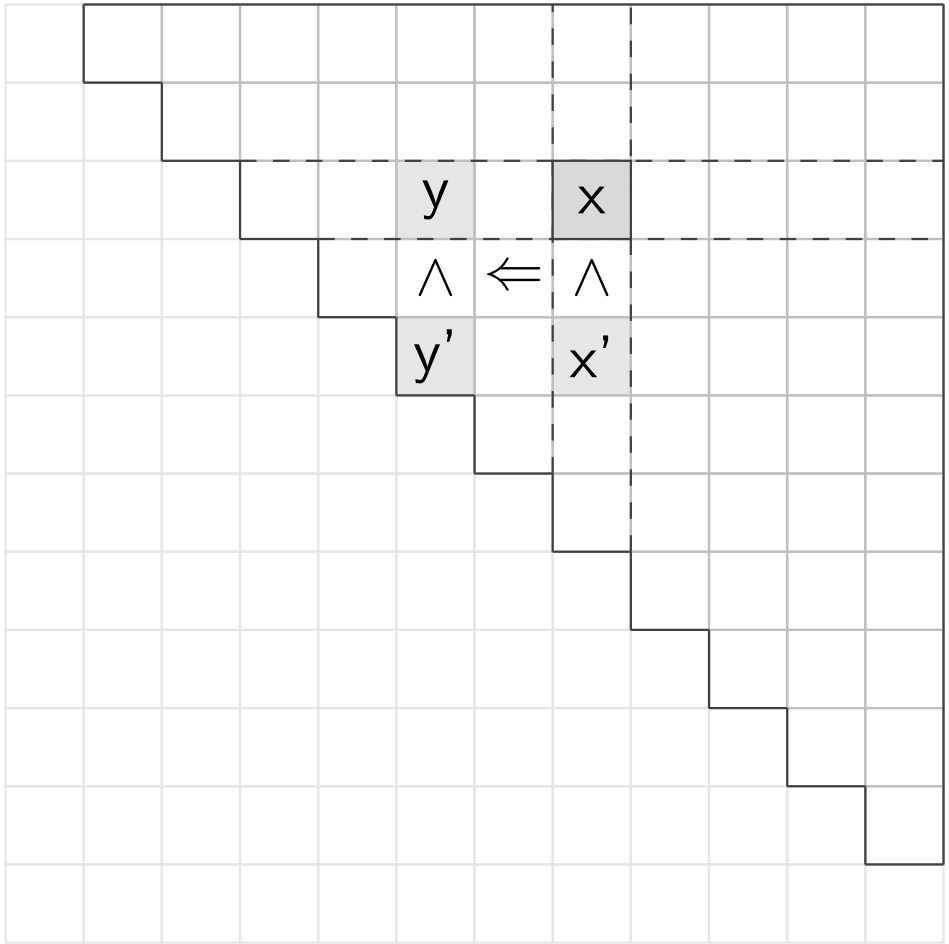


Figure: 凸單調對 B 的影響

1D-1D Problems | 凸單調

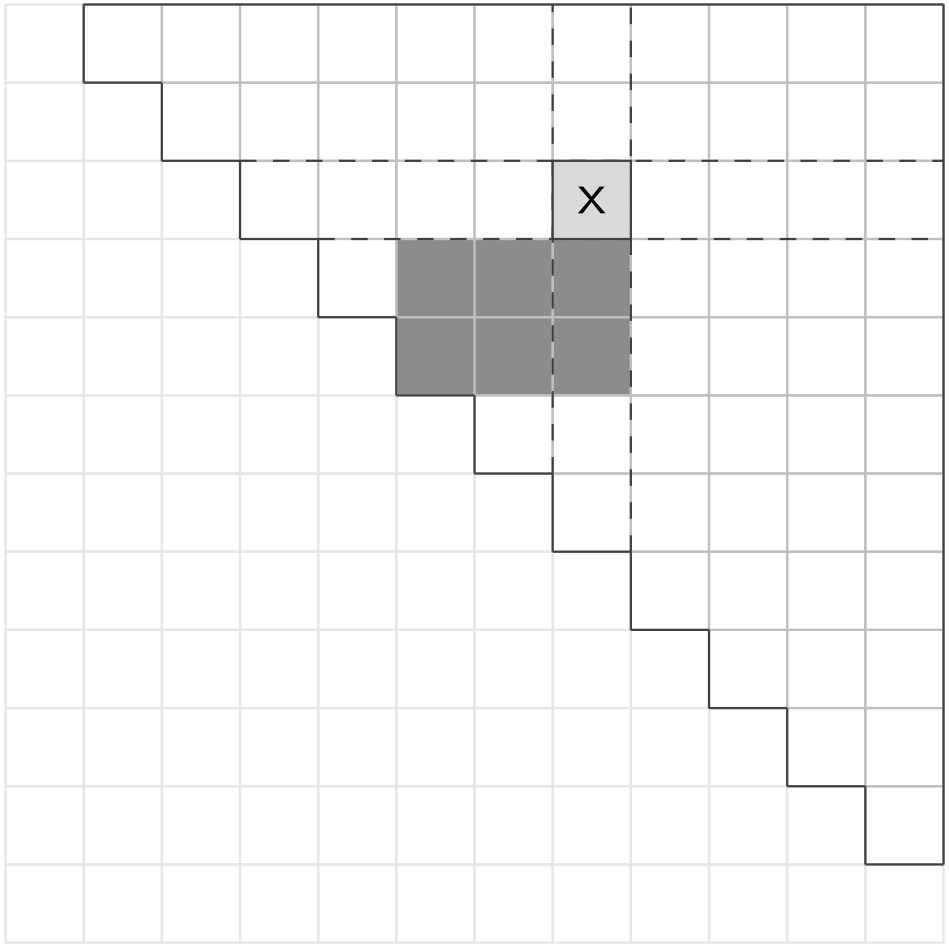


Figure: 凸單調對 B 的影響

1D-1D Problems | 凸單調

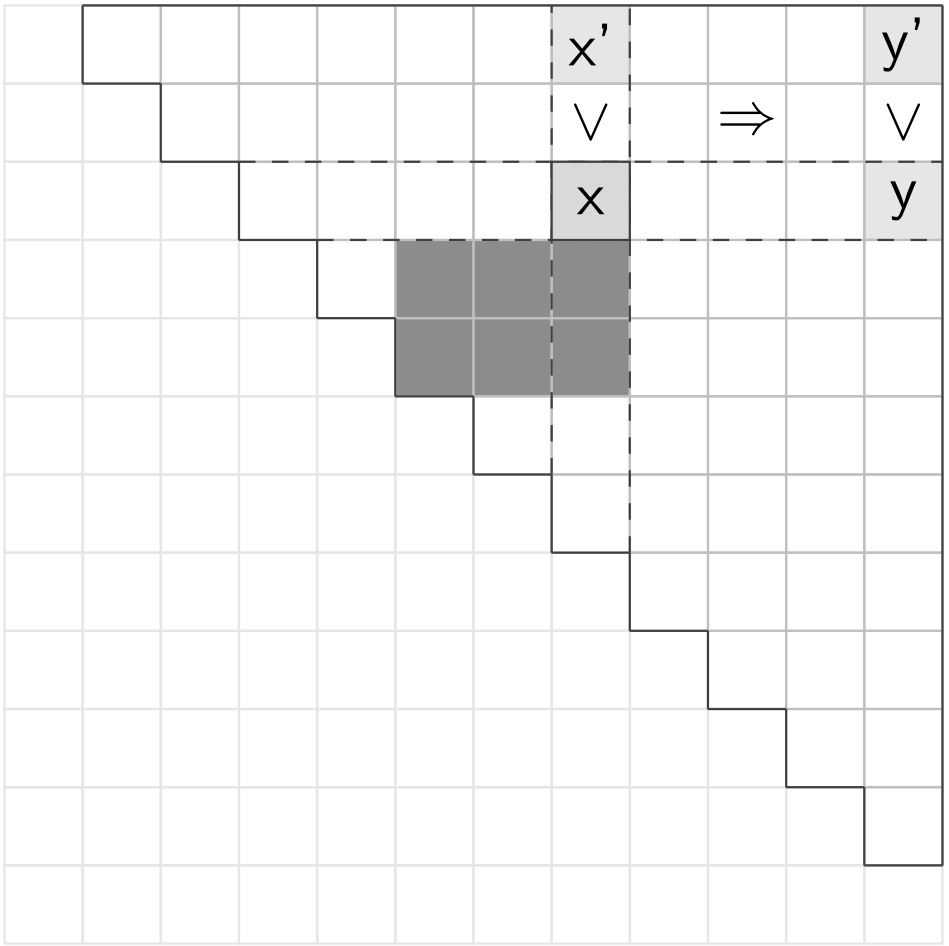


Figure: 凸單調對 B 的影響

1D-1D Problems | 凸單調

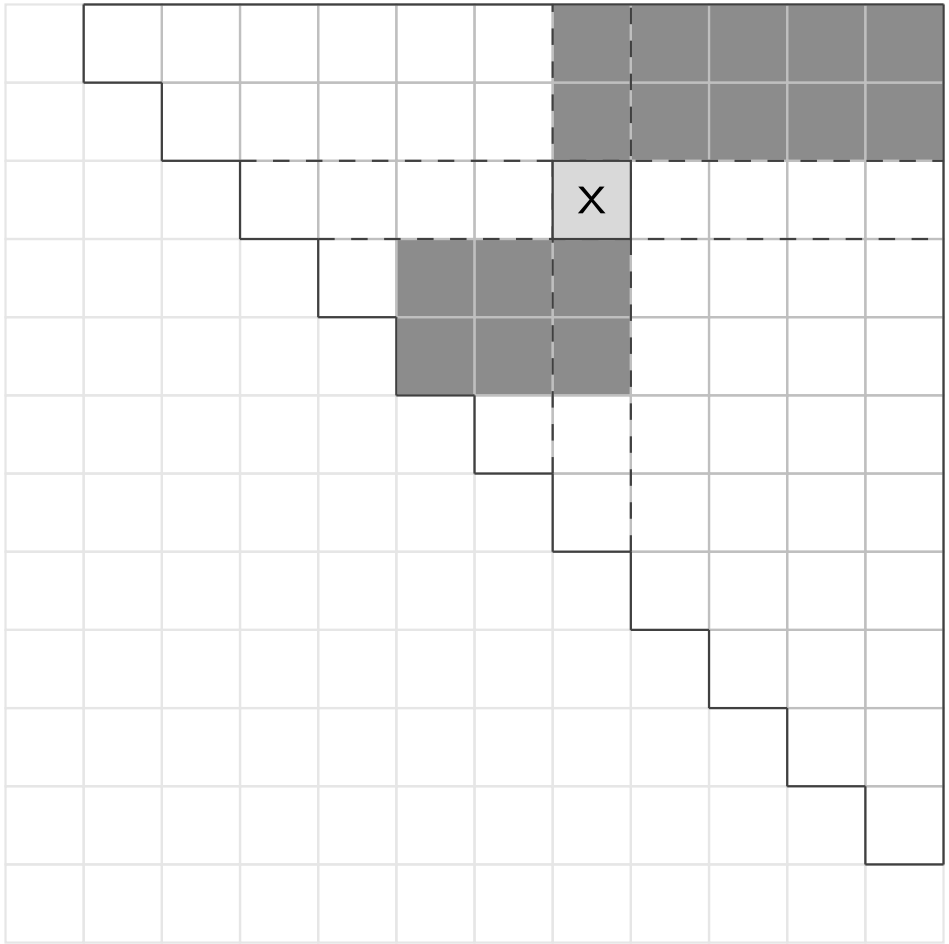


Figure: 凸單調對 B 的影響

1D-1D Problems | 凸單調

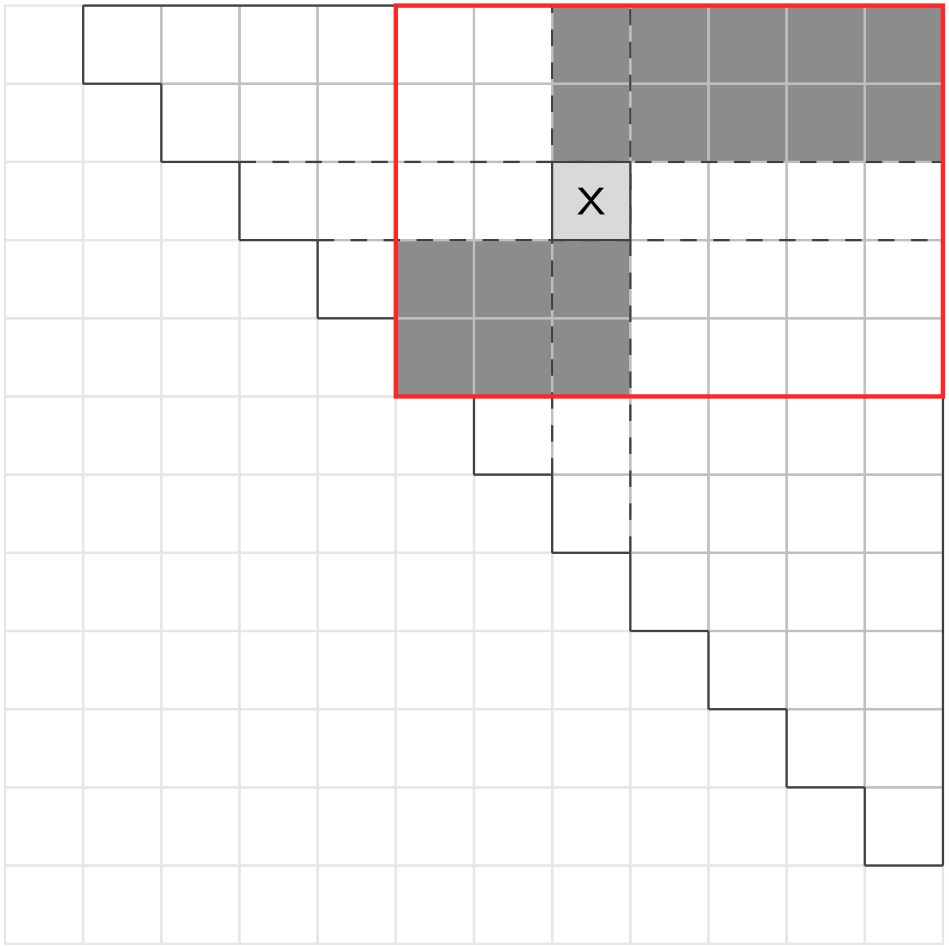


Figure: 凸單調對 B 的影響

1D-1D Problems | 凸單調

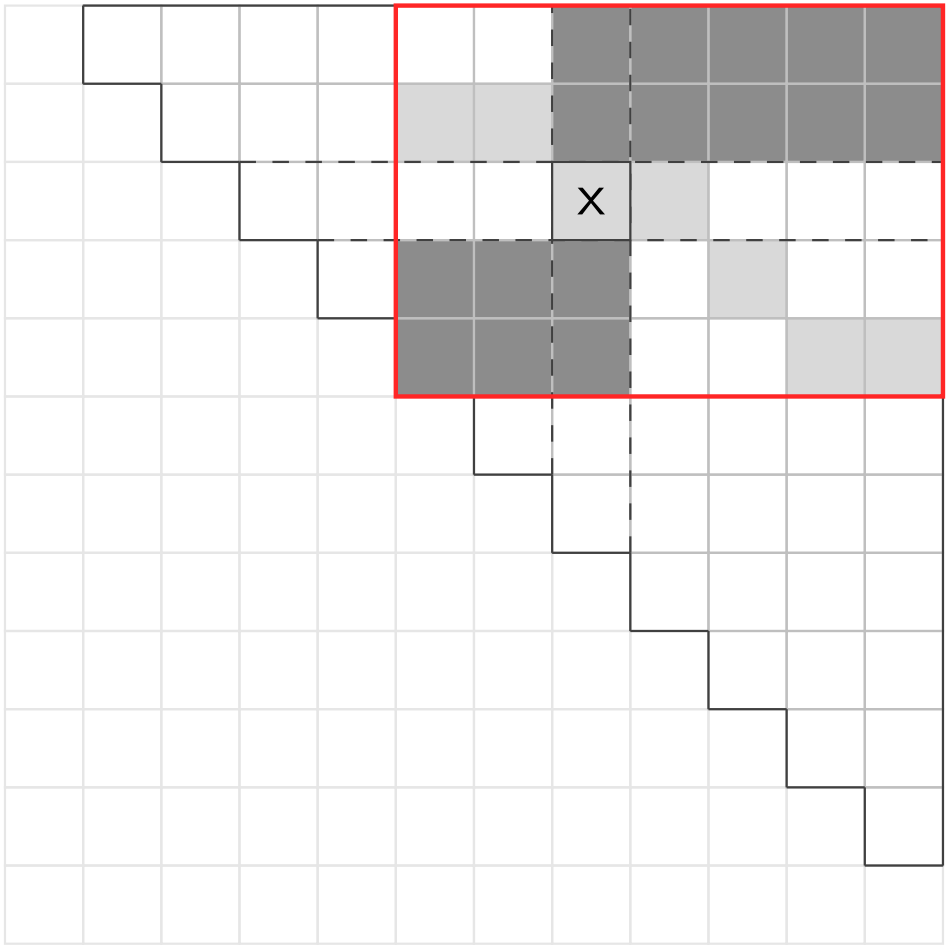


Figure: 凸單調對 B 的影響

1D-1D Problems | 凸單調

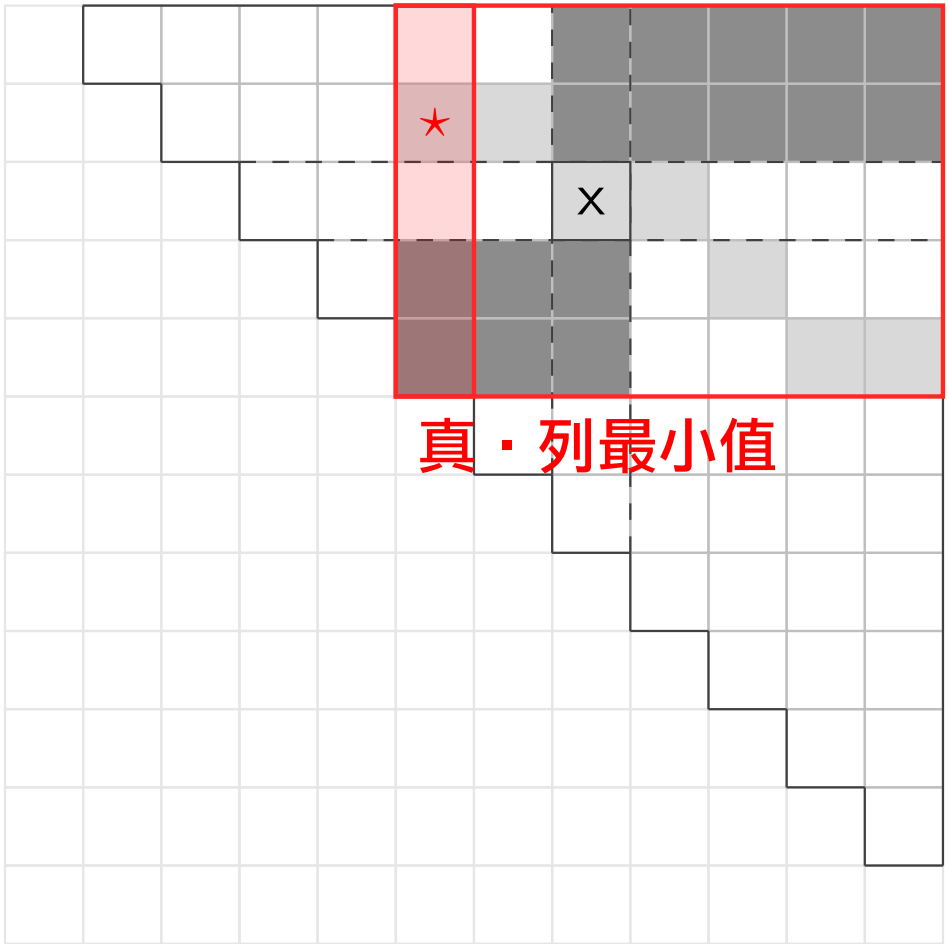


Figure: 凸單調對 B 的影響

1D-1D Problems | 凸單調

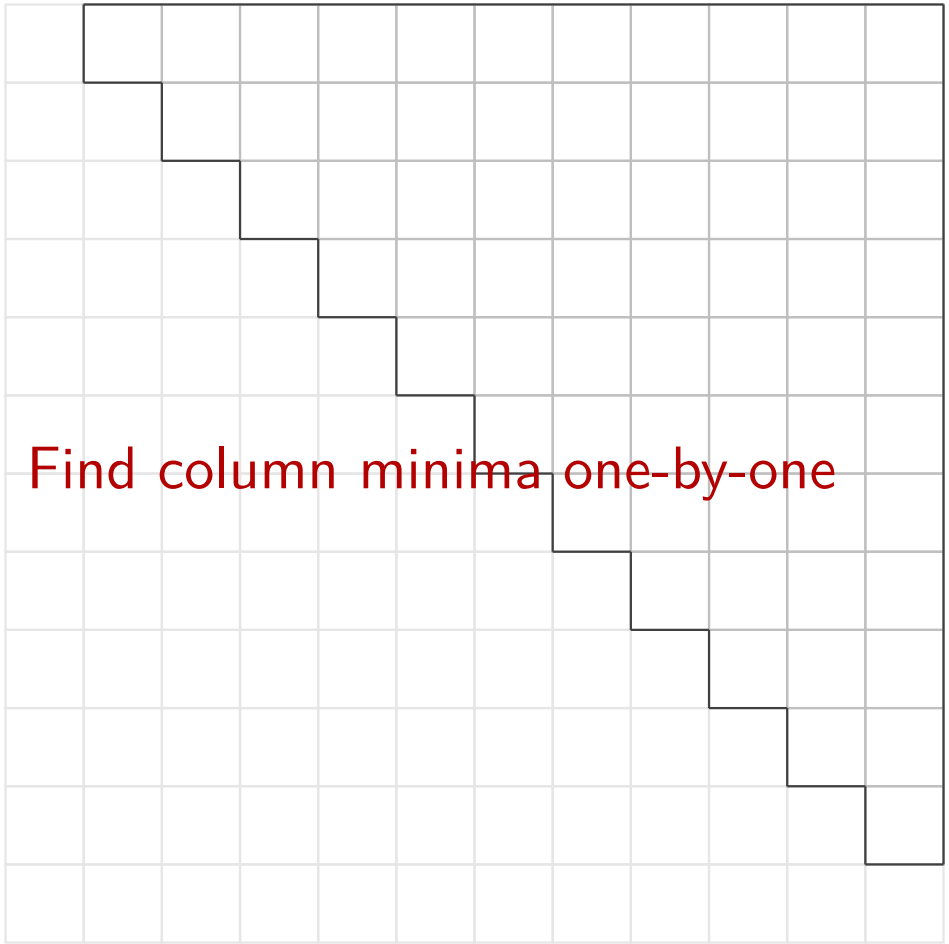


Figure: Finding column minima

1D-1D Problems | 凸單調

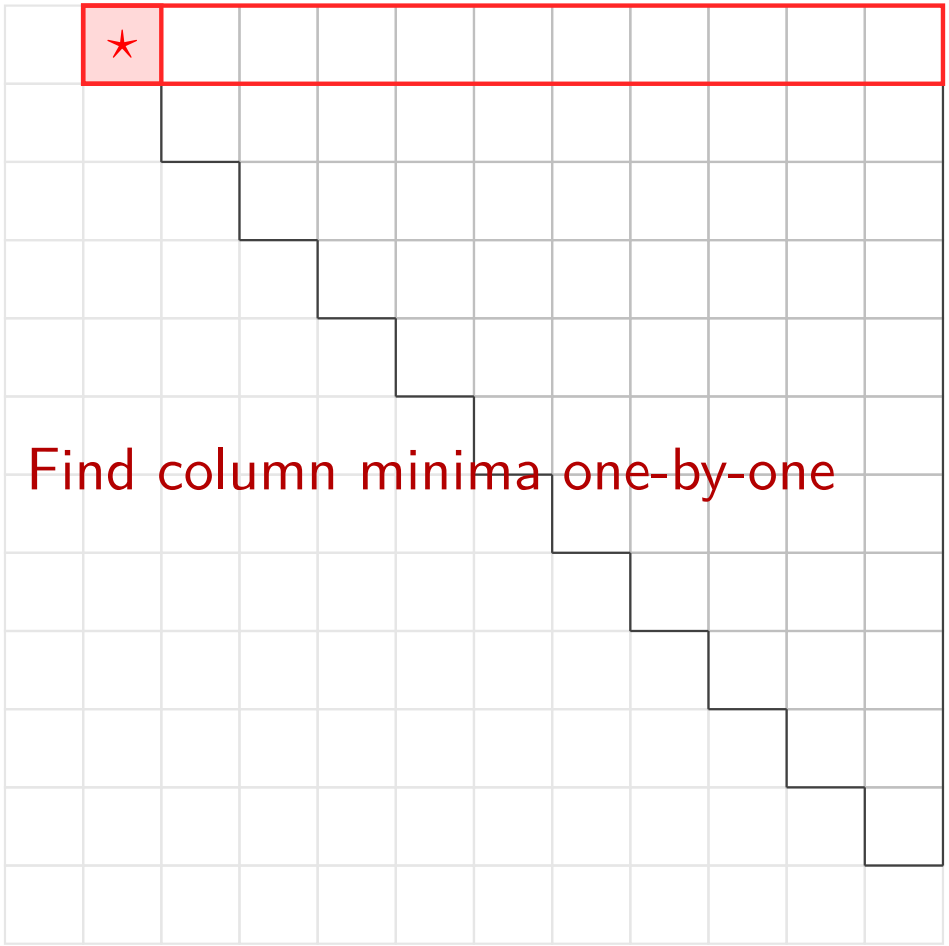


Figure: Finding column minima

1D-1D Problems | 凸單調

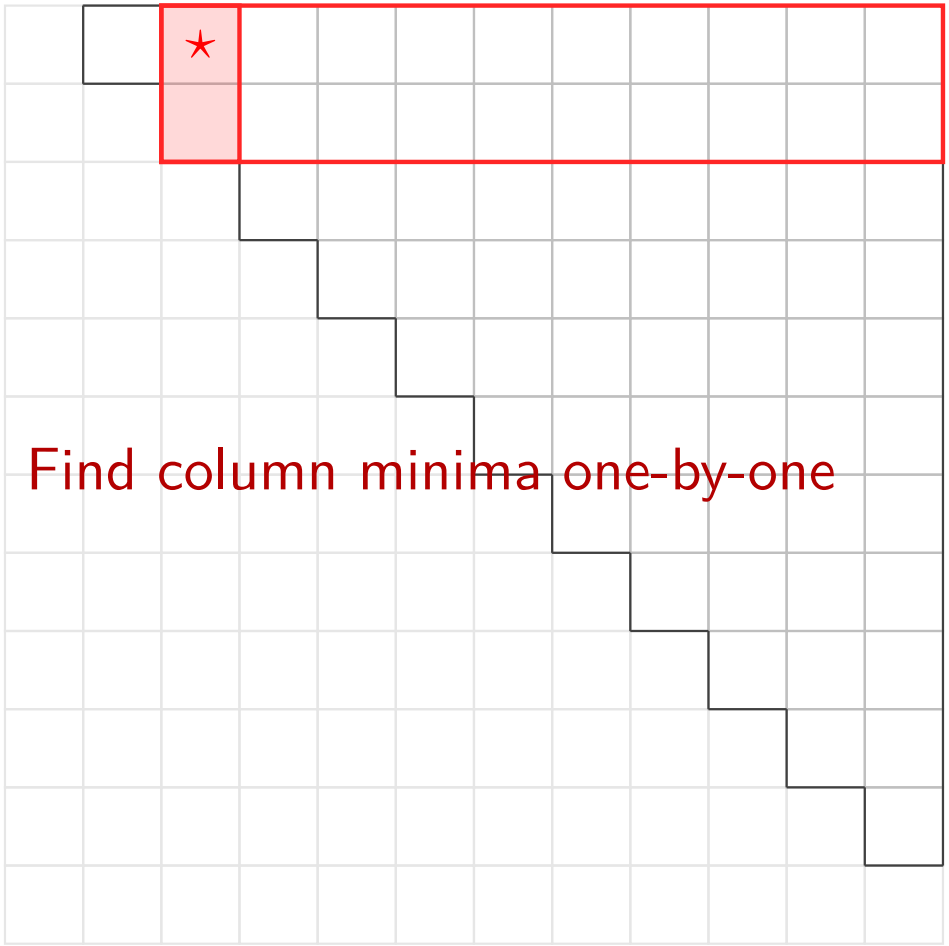


Figure: Finding column minima

1D-1D Problems | 凸單調

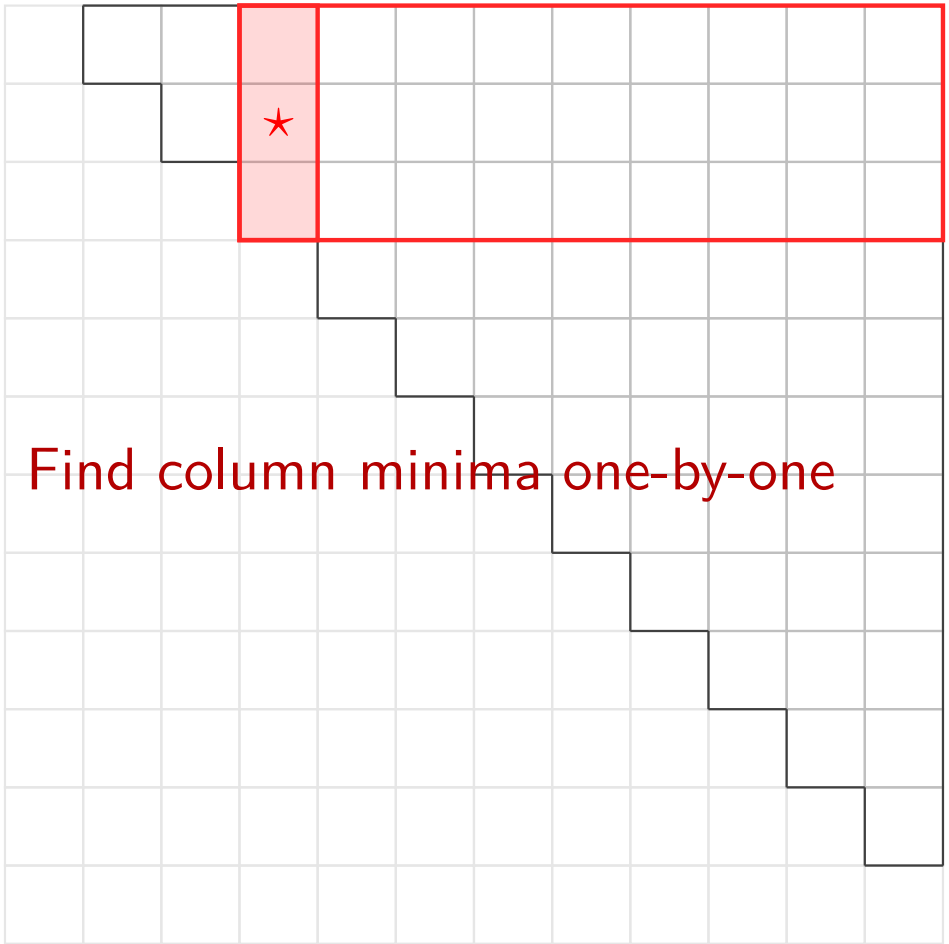


Figure: Finding column minima

1D-1D Problems | 凸單調

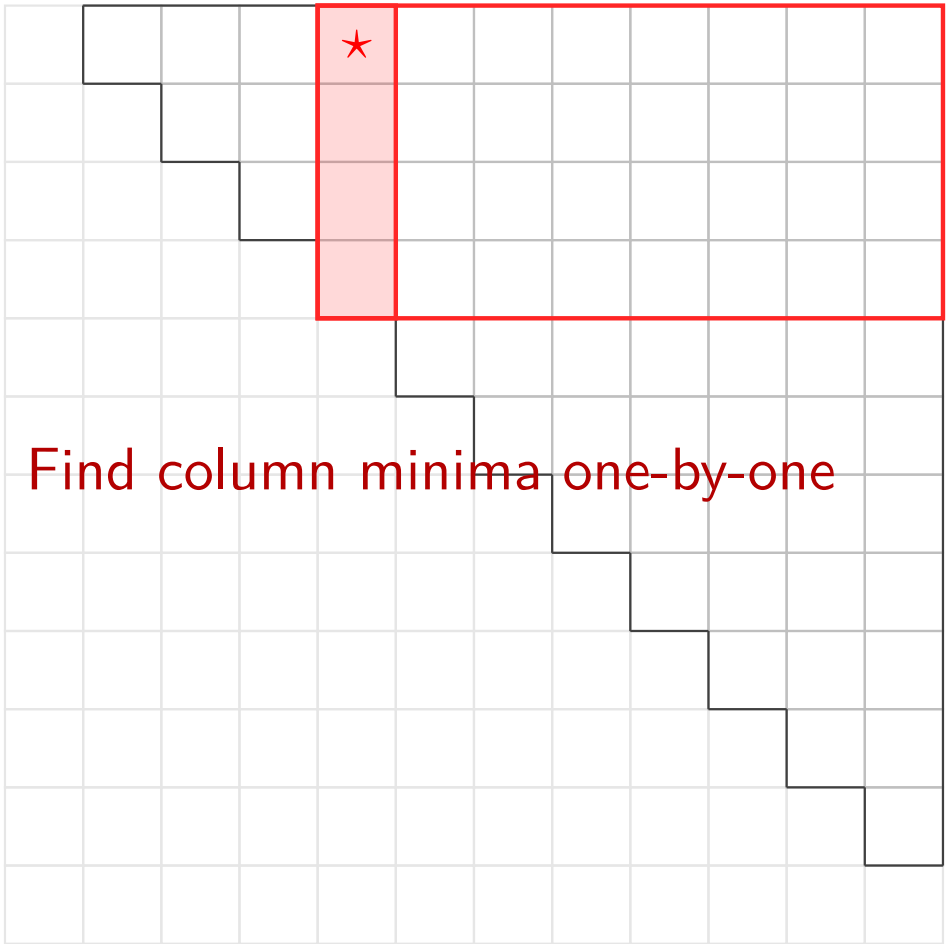


Figure: Finding column minima

1D-1D Problems | 凸單調

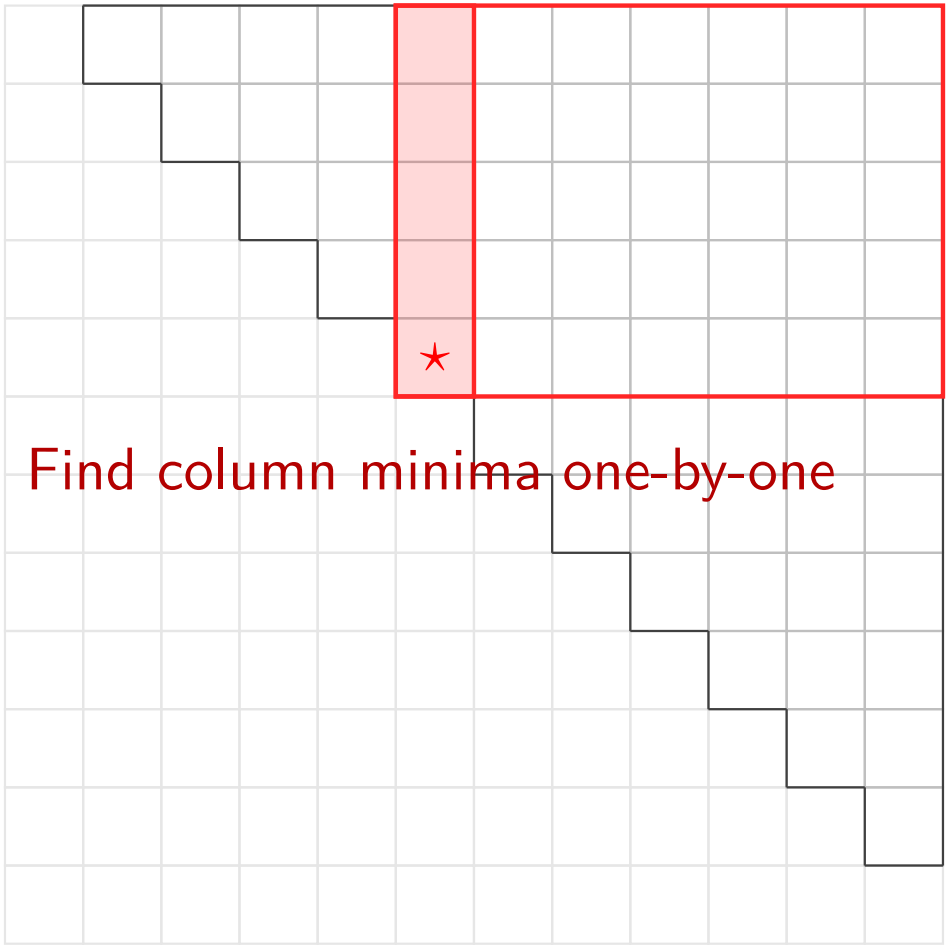


Figure: Finding column minima

1D-1D Problems | 凸單調

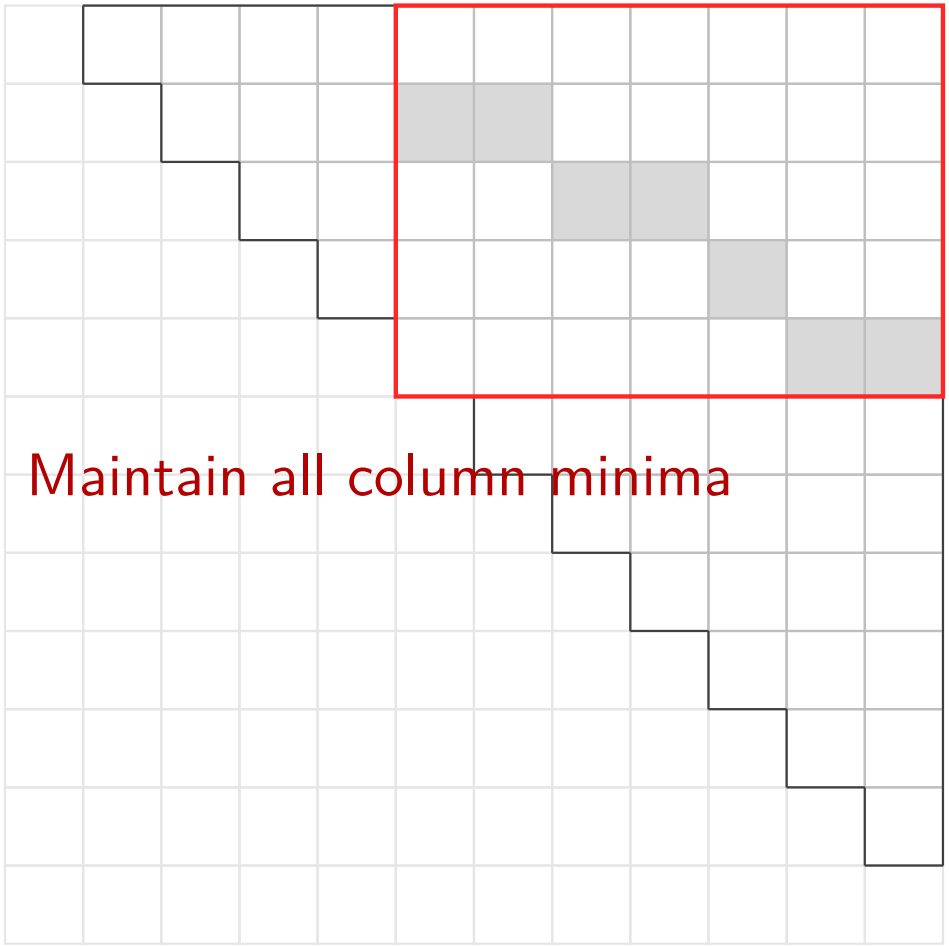


Figure: Finding column minima

1D-1D Problems | 凸單調

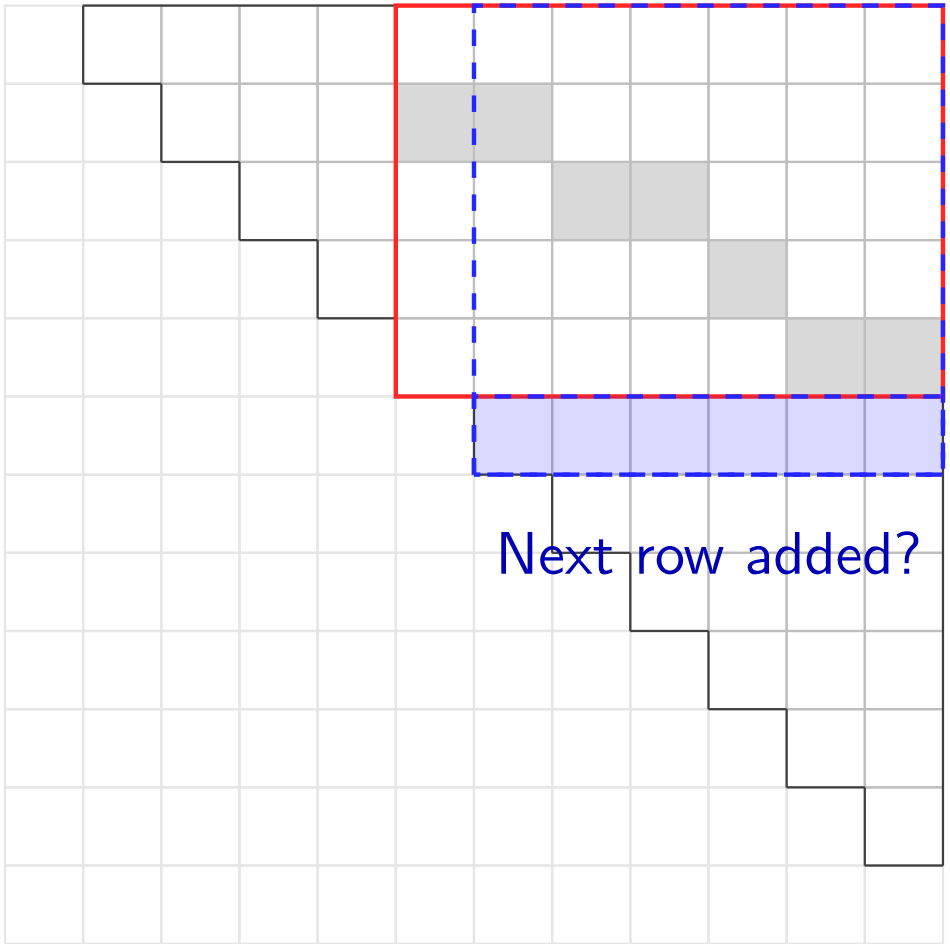


Figure: Finding column minima

1D-1D Problems | 凸單調

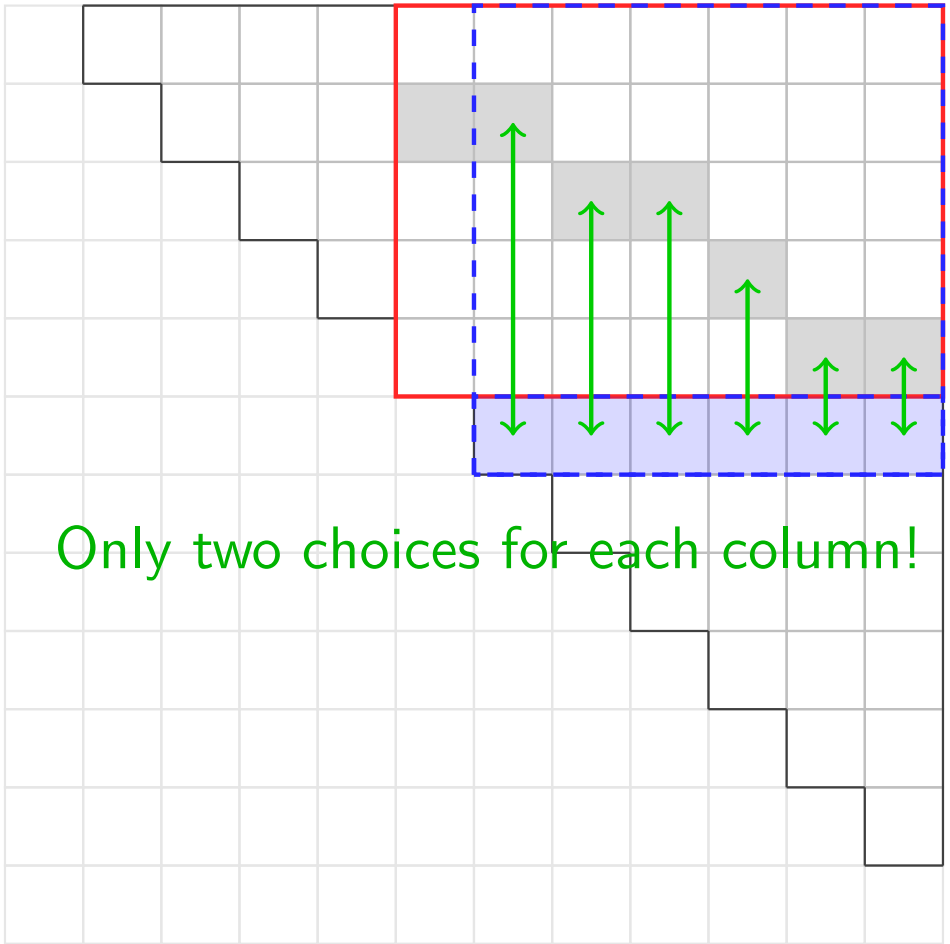


Figure: Finding column minima

1D-1D Problems | 凸單調

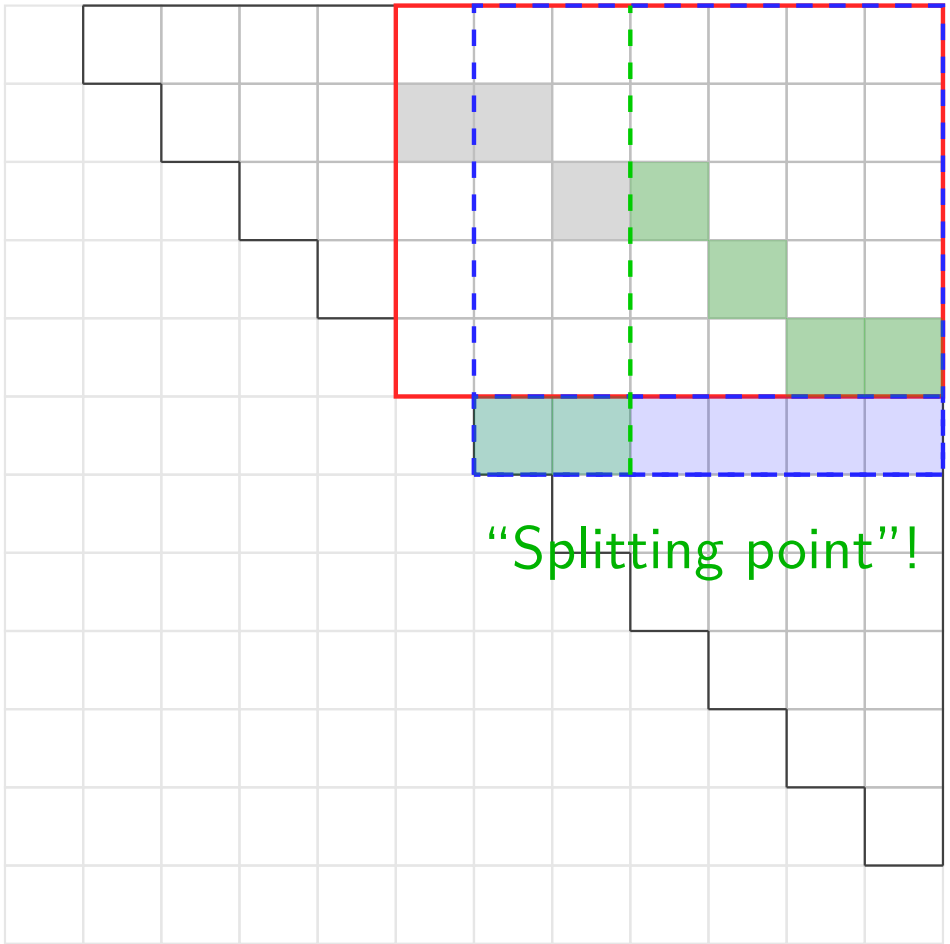


Figure: Finding column minima

1D-1D Problems | 凸單調

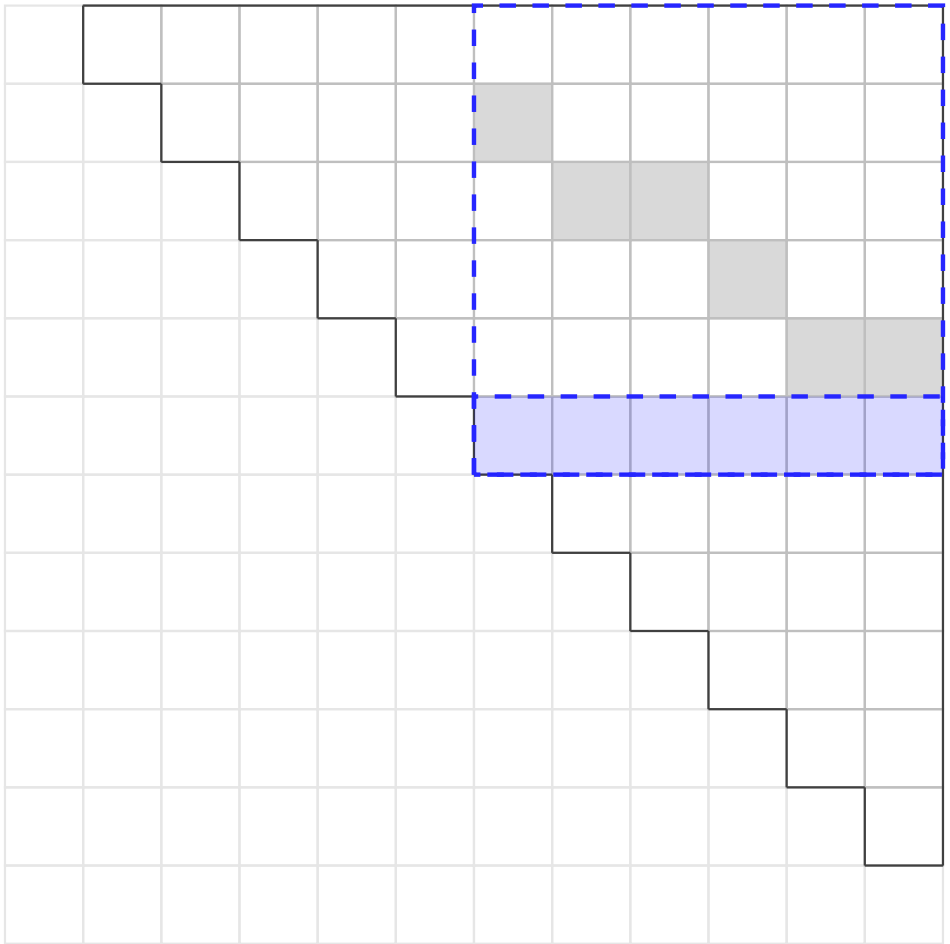


Figure: Finding column minima

1D-1D Problems | 凸單調

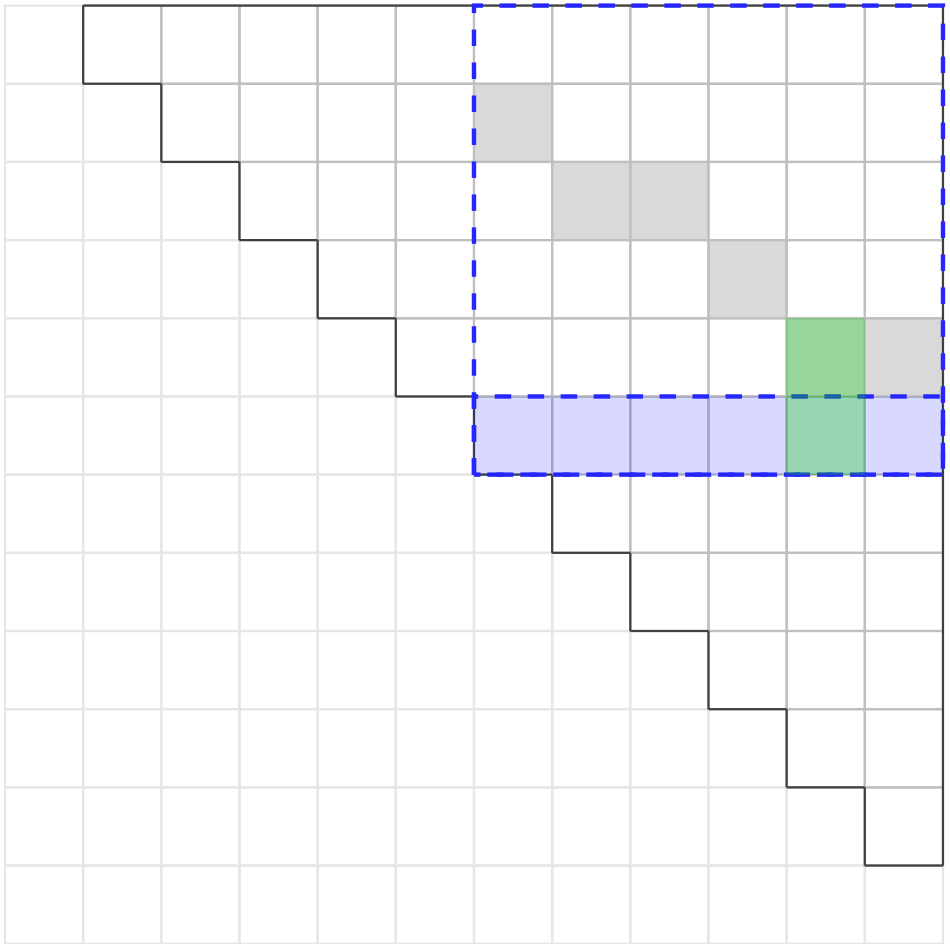


Figure: Finding column minima

1D-1D Problems | 凸單調

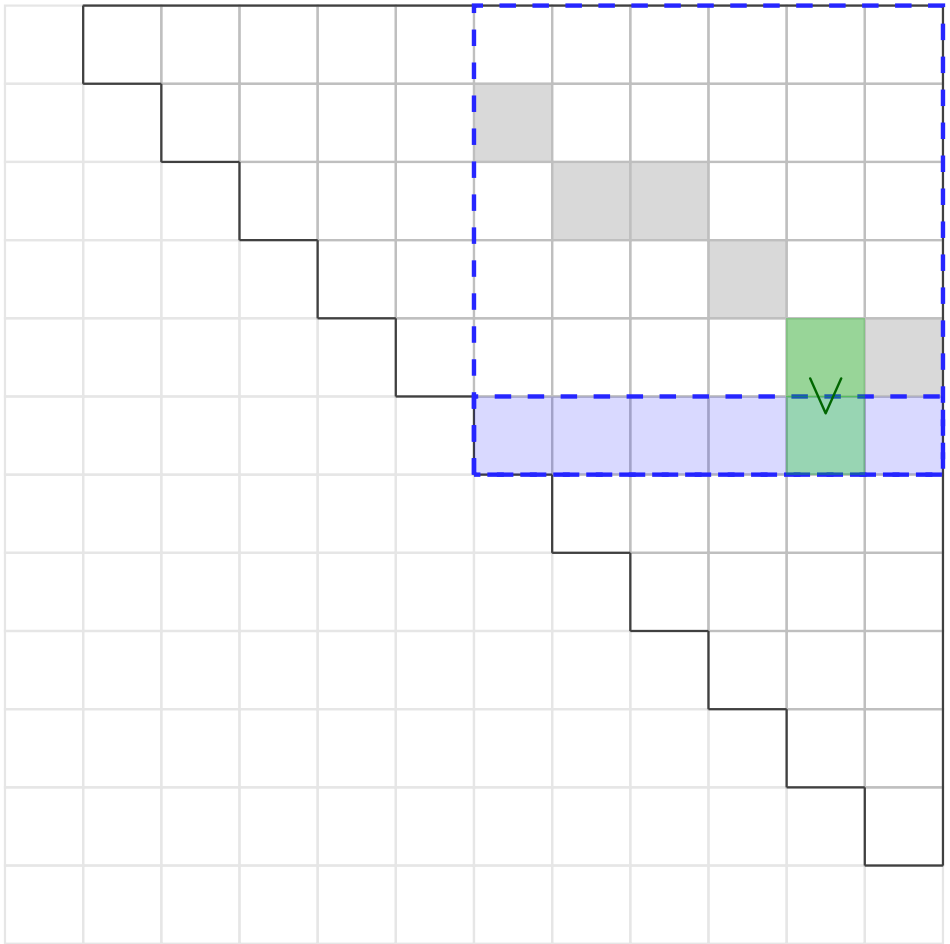


Figure: Finding column minima

1D-1D Problems | 凸單調

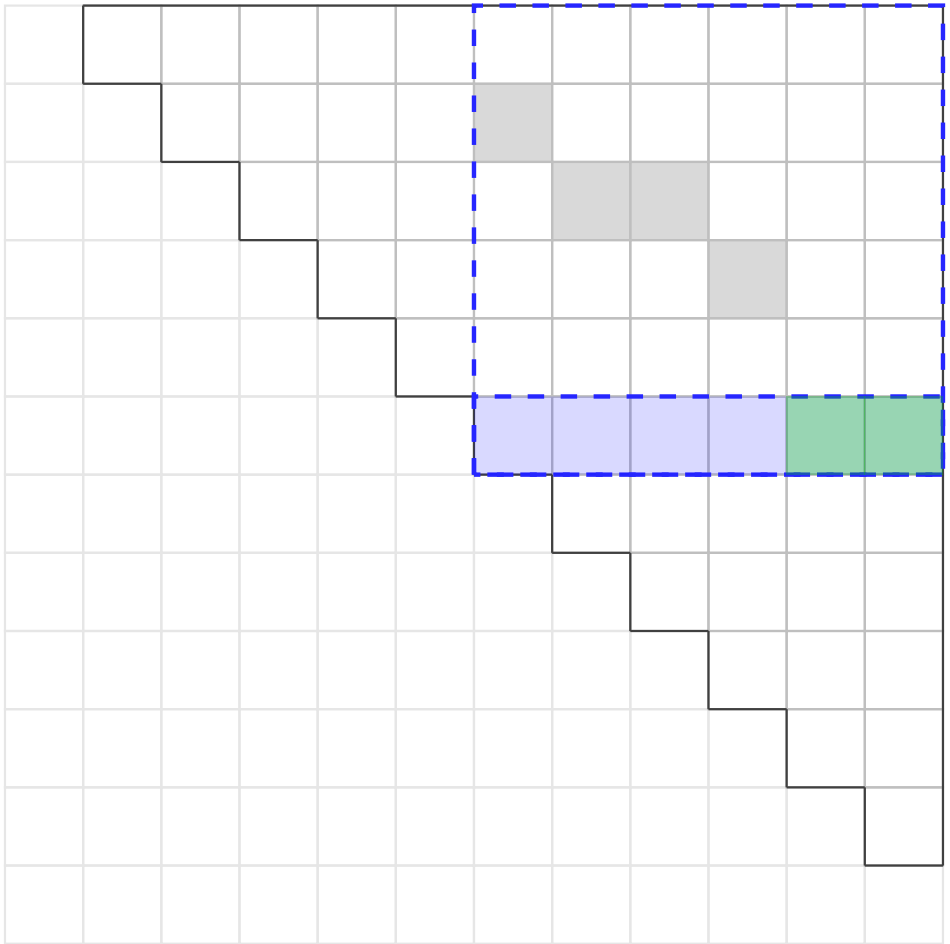


Figure: Finding column minima

1D-1D Problems | 凸單調

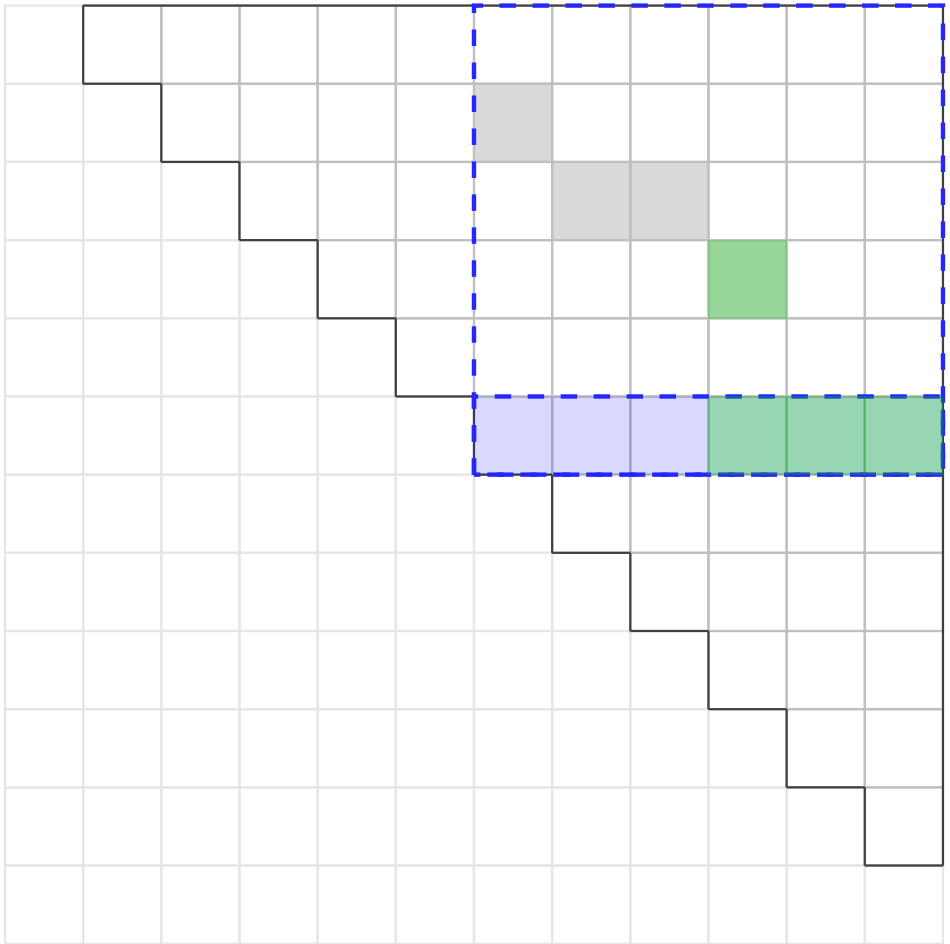


Figure: Finding column minima

1D-1D Problems | 凸單調

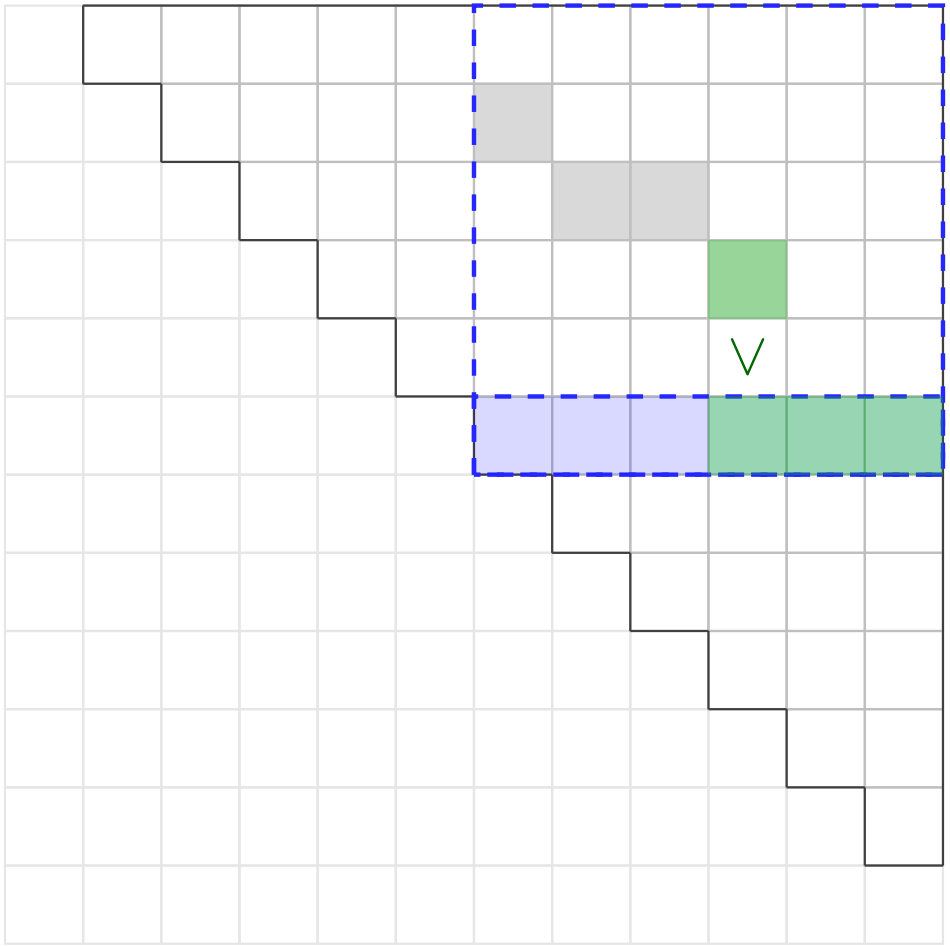


Figure: Finding column minima

1D-1D Problems | 凸單調

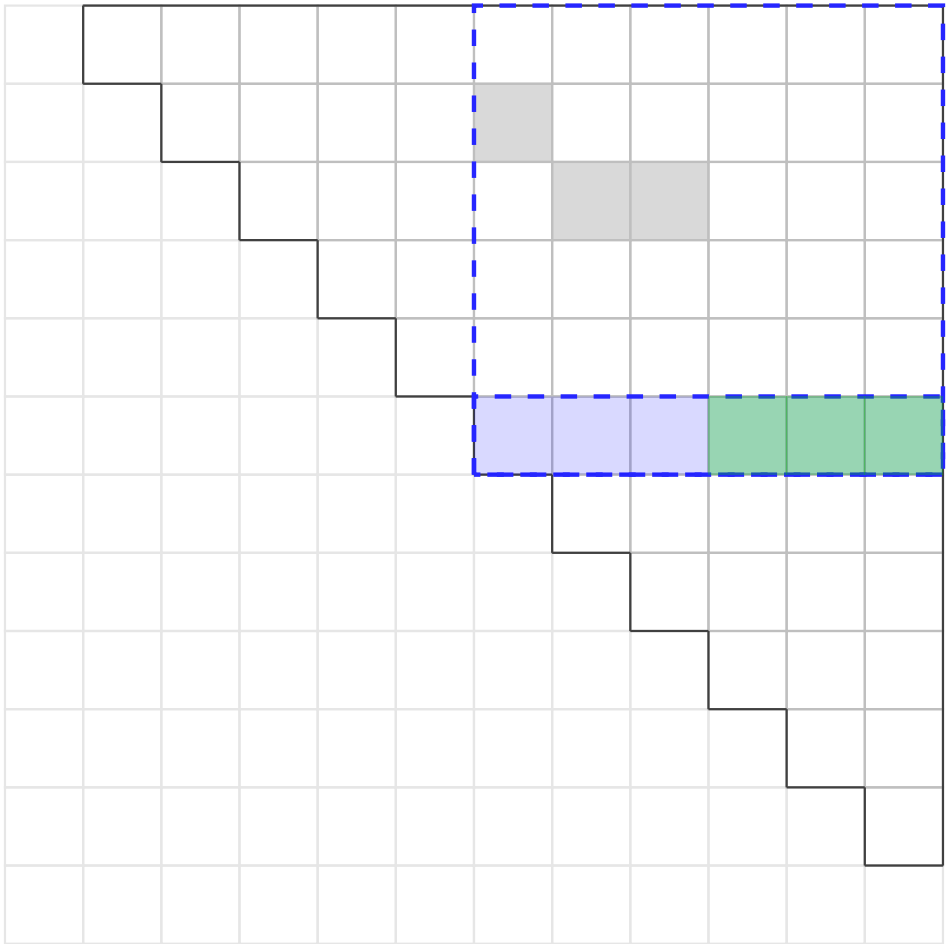


Figure: Finding column minima

1D-1D Problems | 凸單調

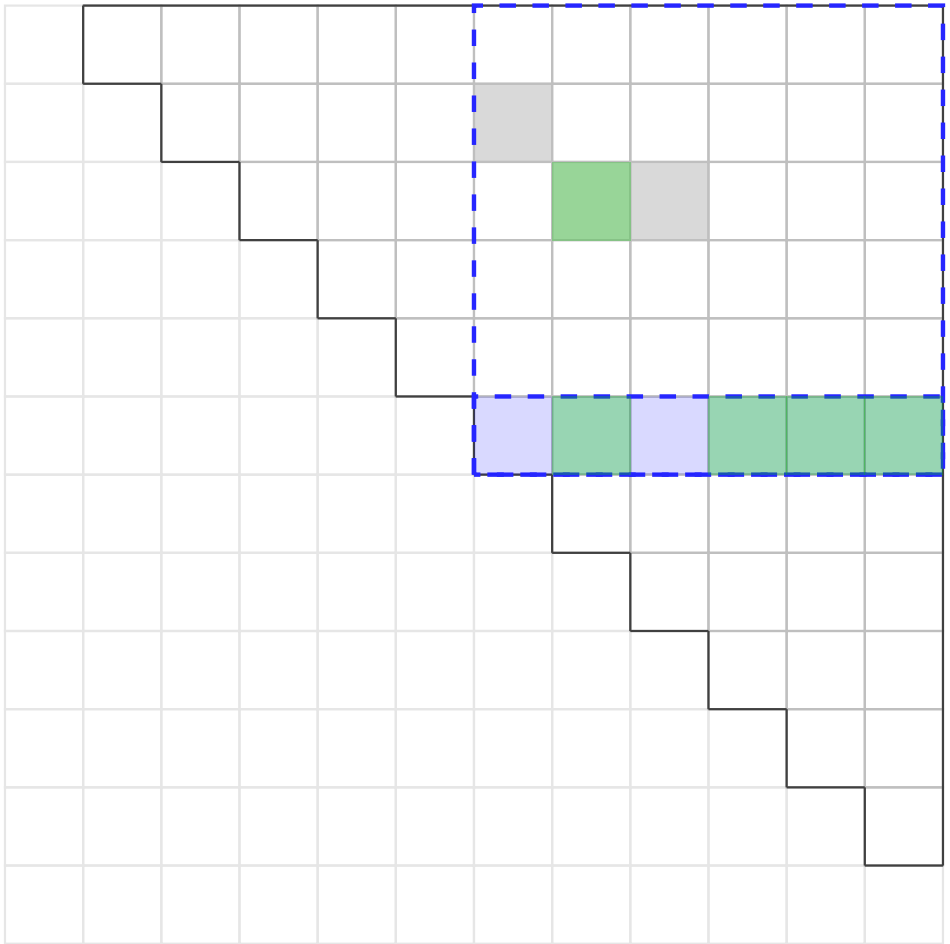


Figure: Finding column minima

1D-1D Problems | 凸單調

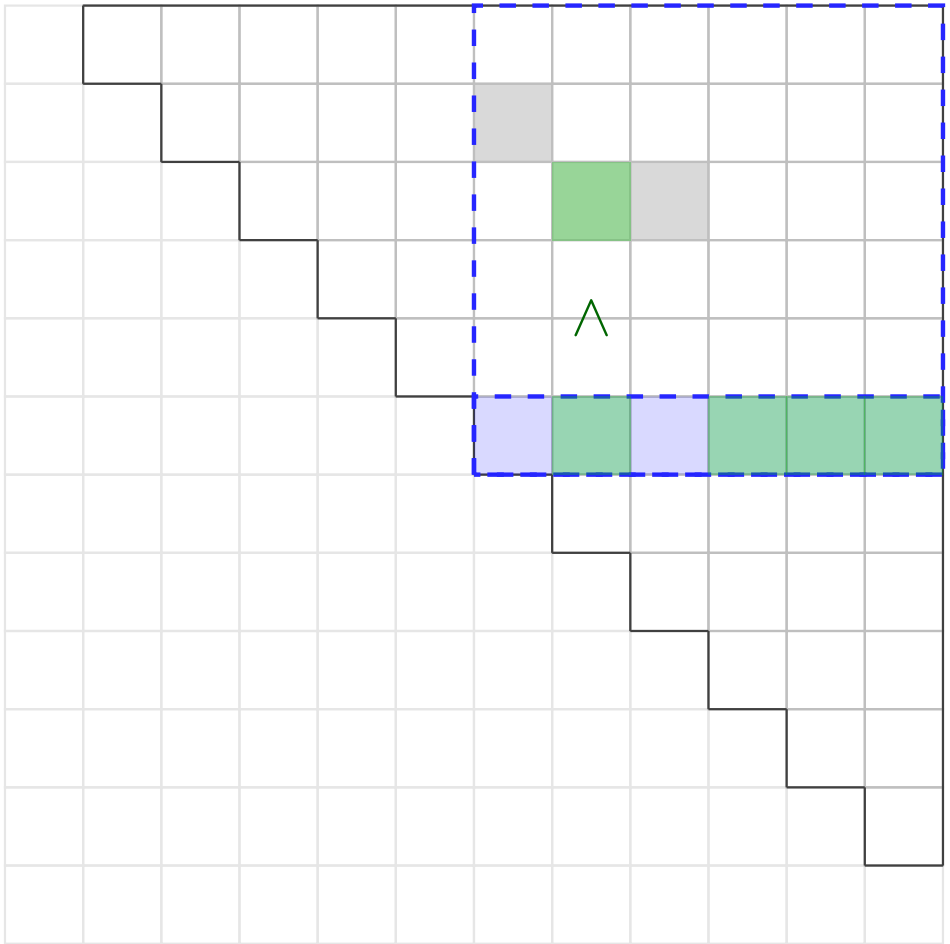


Figure: Finding column minima

1D-1D Problems | 凸單調

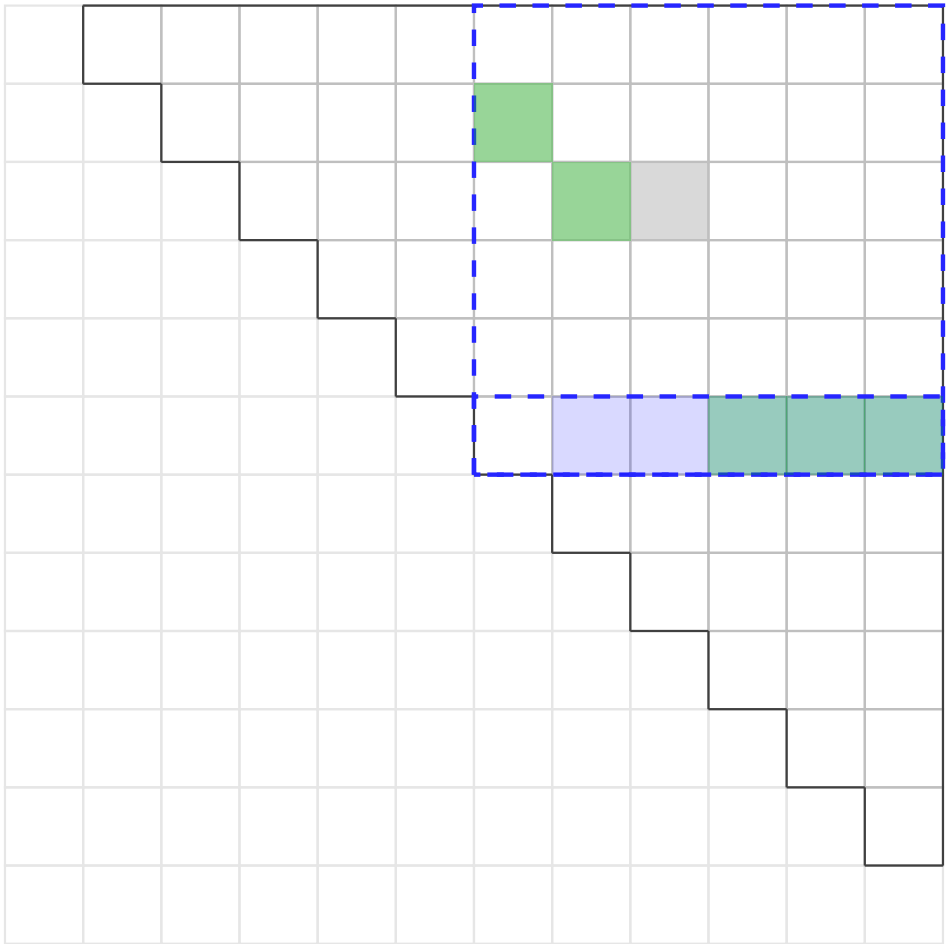


Figure: Finding column minima

1D-1D Problems | 凸單調

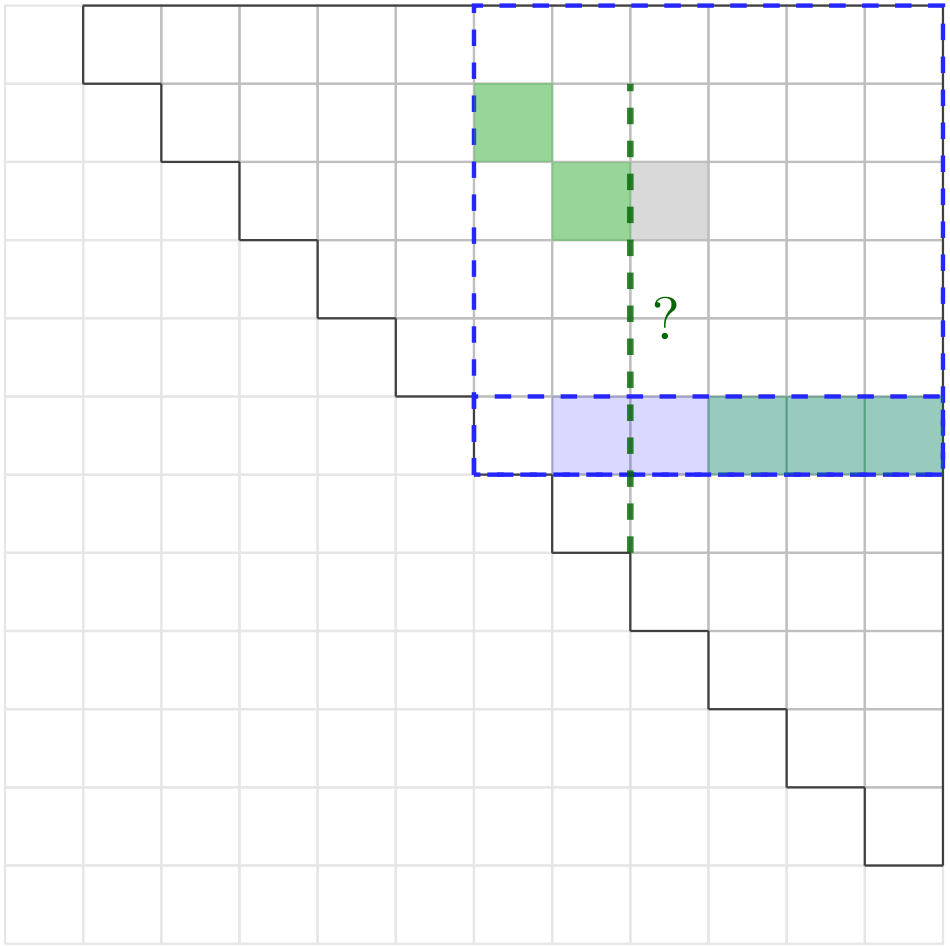


Figure: Finding column minima

1D-1D Problems | 凸單調

Algorithm 3: Convex 1D/1D DP

```
1  function Convex-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← max{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; h_r ... c]
9          S.push([j − 1; c + 1 ... n])
10         E[j] ← S.bottom().head
11     end
12 end
```

1D-1D Problems | 凸單調

Algorithm 3: Convex 1D/1D DP

```
1  function Convex-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← max{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; h_r ... c]
9          S.push([j − 1; c + 1 ... n])
10         E[j] ← S.bottom().head
11     end
12 end
```

找到第一個 (最右) 不完全比新的一行差的行區間 r 。

1D-1D Problems | 凸單調

Algorithm 3: Convex 1D/1D DP

```
1  function Convex-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← max{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; h_r ... c]
9          S.push([j − 1; c + 1 ... n])
10         E[j] ← S.bottom().head
11     end
12 end
```

在行區間 r 以前的所有區間都「掰了」。

1D-1D Problems | 凸單調

Algorithm 3: Convex 1D/1D DP

```
1  function Convex-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← max{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; h_r ... c]
9          S.push([j − 1; c + 1 ... n])
10         E[j] ← S.bottom().head
11     end
12 end
```

對於行區間 r，二分搜找出新區間比較好的分界。

1D-1D Problems | 凸單調

Algorithm 3: Convex 1D/1D DP

```
1  function Convex-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← max{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; h_r ... c]
9          S.push([j − 1; c + 1 ... n])
10         E[j] ← S.bottom().head
11     end
12 end
```

更新 stack 。

1D-1D Problems | 凸單調

Algorithm 3: Convex 1D/1D DP

```
1  function Convex-1D1D
2      Initialize stack S with ([0; 1 ... n])
3      for j ← 1 to n do
4          // S : ([i_1; j ... h_1], [i_2; h_1+1 ... h_2], ..., [i_k; h_{k-1}+1 ... h_k])
5          r ← min{t | B[i_t][h_t] < B[j − 1][h_t]}
6          pop from S all segments before r-th
7          c ← max{t | B[i_r][t] < B[j − 1][t]}
8          S.top() ← [i_r; h_r ... c]
9          S.push([j − 1; c + 1 ... n])
10         E[j] ← S.bottom().head
11     end
12 end
```

紀錄列最小值：$E[j]$。

1D-1D Problems | 凸單調

時間複雜度？

- **加入更新的第一步：找到第一個需被保留的行區間。**
  $\Rightarrow$ AYCP-stack – 均攤 $O(N)$。
- **加入更新的第二步：二分搜該區間與新的行區間的分界。**
  $\Rightarrow$ 共 $O(\lg N)$ 次決策點比較。
- **同樣地是 $O(N \lg N)$。**

1D-1D Problems | 凸單調

時間複雜度？

- **加入更新的第一步：找到第一個需被保留的行區間。**
  $\Rightarrow$ AYCP-stack – **均攤** $O(N)$。
- **加入更新的第二步：二分搜該區間與新的行區間的分界。**
  $\Rightarrow$ 共 $O(\lg N) = O(N \lg N)$。
- **同樣地是** $O(N \lg N)$。

1D-1D Problems | 凸單調

時間複雜度？

- **加入更新的第一步：找到第一個需被保留的行區間。**
  $\Rightarrow$ AYCP-stack – **均攤** $O(N)$。
- **加入更新的第二步：二分搜該區間與新的行區間的分界。**
  $\Rightarrow N \cdot O(\lg N) = O(N \lg N)$。
- **同樣地是** $O(N \lg N)$。

1D-1D Problems | 凸單調

時間複雜度？

- **加入更新的第一步：找到第一個需被保留的行區間。**
  $\Rightarrow$ AYCP-stack – **均攤** $O(N)$。
- **加入更新的第二步：二分搜該區間與新的行區間的分界。**
  $\Rightarrow N \cdot O(\lg N) = O(N \lg N)$。
- **同樣地是** $O(N \lg N)$。

1D-1D Problems | 凸單調

時間複雜度？

- **加入更新的第一步：找到第一個需被保留的行區間。**
  $\Rightarrow$ AYCP-stack – **均攤** $O(N)$。
- **加入更新的第二步：二分搜該區間與新的行區間的分界。**
  $\Rightarrow N \cdot O(\lg N) = O(N \lg N)$。
- **同樣地是** $O(N \lg N)$。

Monge Condition

然而，我們如何知道一個矩陣 (或轉移) 具有單調性呢？

Definition (Monge Condition)

Monge Condition

然而，我們如何知道一個矩陣 (或轉移) 具有單調性呢？

Definition (Monge Condition)

對於 $m \times n$ **矩陣** B**，若** $\forall 1 \leq i_1 < i_2 \leq m, 1 \leq j_1 < j_2 \leq n$ **有：**

$$B[i_1][j_1] + B[i_2][j_2] \quad \leq (\geq) \quad B[i_1][j_2] + B[i_2][j_1]$$

則我們說他符合 Convex (Concave) Monge Condition**。**

Monge Condition

然而，我們如何知道一個矩陣 (或轉移) 具有單調性呢？

Definition (Monge Condition – 等價形式)

對於 $m \times n$ **矩陣** B**，若** $\forall 1 \leq i < m, 1 \leq j < n$ **有：**

$$B[i][j] + B[i+1][j+1] \quad \leq (\geq) \quad B[i][j+1] + B[i+1][j]$$

則我們說他符合 Convex (Concave) Monge Condition**。**

Monge Condition

- Convex (Concave) Monge Condtion $\Rightarrow$ 凸 (凹) 單調性。
- 反方向推論不成立 (Monge Condition 較強)。
- Monge Condition 常有助於我們驗證單調性的存在。

Monge Condition

- Convex (Concave) Monge Condtion $\Rightarrow$ **凸 (凹) 單調性**。
- 反方向推論不成立 (Monge Condition 較強)。
- Monge Condition 常有助於我們驗證單調性的存在。

Monge Condition

- Convex (Concave) Monge Condtion $\Rightarrow$ **凸 (凹) 單調性**。
- **反方向推論不成立** (Monge Condition **較強**)。
- Monge Condition 常有助於我們驗證單調性的存在。

Monge Condition

- Convex (Concave) Monge Condtion $\Rightarrow$ **凸 (凹) 單調性**。
- **反方向推論不成立** (Monge Condition **較強**)。
- Monge Condition **常有助於我們驗證單調性的存在**。

例題：Batch Scheduling

N 樣編號依序為 $1, 2, 3, \ldots, N$ 的工作必須依序批次完成。

工作必須依序執行。每次可從尚未被執行的工作中挑若干個編號最小的工作 (亦即不能跳過尚未執行的工作) 來「批次執行」。

已知每個工作有他們的執行時間 t_i以及單位時間代價 p_i。若編號 $i, i+1, \ldots, i+k$ 的工作被同時批次執行，則該次執行要花時間 $s + \sum_{j=i}^{k} t_j$，其中 s 為一給定常數，代表批次執行準備時間。

工作 i 的完成時間視為他所屬的批次完全執行完畢的時間。若工作 i 在時間 t 完成，則總共必須付出 $p_i \cdot t$ 的代價。

給定 N, s, t_i, p_i，問在最佳批次排程下，執行完全部工作至少要付出多少代價？

$(N \leq 10^6, 0 \leq s, t_i, p_i \leq 10^9)$

例題：Batch Scheduling

分析：

- $p_i \cdot t$ 看起來難以直接處理，因為一個子問題中的 t 和之前的狀態是相關的，在狀態中紀錄當前用了多少時間亦不可行。
- 換個方向：每過一單位時間，就對尚未完成的工作 i 算 p_i 的代價。如此一來，子問題可以獨立出來！
- $DP[i] =$ 到 i 以前的工作都完成，至少付出多少代價。
- $DP[i] = \min_j \{DP[j] + f(j+1, i)\}$，$f(j+1, i)$ 為批次完成 $j+1, j+2, \ldots, i$ 這些工作所需付出的代價 (包括後面的工作「隨時間流失的代價」)。

例題：Batch Scheduling

分析：

- $p_i \cdot t$ 看起來難以直接處理，因為一個子問題中的 t 和之前的狀態是相關的，在狀態中紀錄當前用了多少時間亦不可行。
- 換個方向：每過一單位時間，就對尚未完成的工作 i 算 p_i 的代價。如此一來，子問題可以獨立出來！
- $DP[i] =$ 到 i 以前的工作都完成，至少付出多少代價。
- $DP[i] = \min_j \{DP[j] + f(j+1, i)\}$，$f(j+1, i)$ 為批次完成 $j+1, j+2, \ldots, i$ 這些工作所需付出的代價 (包括後面的工作「隨時間流失的代價」)。

例題：Batch Scheduling

分析：

- $p_i \cdot t$ 看起來難以直接處理，因為一個子問題中的 t 和之前的狀態是相關的，在狀態中紀錄當前用了多少時間亦不可行。
- 換個方向：每過一單位時間，就對尚未完成的工作 i 算 p_i 的代價。如此一來，子問題可以獨立出來！
- $DP[i] =$ 到i以前的工作都完成，至少付出多少代價。
- $DP[i] = \min_j \{DP[j] + f(j+1, i)\}$，$f(j+1, i)$ 為批次完成 $j+1, j+2, \ldots, i$ 這些工作所需付出的代價 (包括後面的工作「隨時間流失的代價」)。

例題：Batch Scheduling

分析：

- $p_i \cdot t$ 看起來難以直接處理，因為一個子問題中的 t 和之前的狀態是相關的，在狀態中紀錄當前用了多少時間亦不可行。
- 換個方向：每過一單位時間，就對尚未完成的工作 i 算 p_i 的代價。如此一來，子問題可以獨立出來！
- $DP[i] =$ 到i以前的工作都完成，至少付出多少代價。
- $DP[i] = \min_j \{DP[j] + f(j+1, i)\}$，
  $f(j+1, i)$ 為批次完成 $j+1, j+2, \ldots, i$ 這些工作所需付出的代價 (包括後面的工作「隨時間流失的代價」)。

離線矩陣列最小元素線性算法

- 我們看到一個線上單調矩陣 B 的列最小元素可以在 $O(N \lg N)$ 求出。
- 若 B 是離線的，情形是否會有所不同？
- 離線可能有什麼好處？
- 以下我們考慮 B 為一凸單調矩陣。

離線矩陣列最小元素線性算法

- 我們看到一個線上單調矩陣 B 的列最小元素可以在 $O(N \lg N)$ 求出。
- 若 B 是離線的，情形是否會有所不同？
- 離線可能有什麼好處？
- 以下我們考慮 B 為一凸單調矩陣。

離線矩陣列最小元素線性算法

- 我們看到一個線上單調矩陣 B 的列最小元素可以在 $O(N \lg N)$ 求出。
- 若 B 是離線的，情形是否會有所不同？
- 離線可能有什麼好處？
- 以下我們考慮 B 為一凸單調矩陣。

離線矩陣列最小元素線性算法

- 我們看到一個線上單調矩陣 B 的列最小元素可以在 $O(N \lg N)$ 求出。
- 若 B 是離線的，情形是否會有所不同？
- 離線可能有什麼好處？
- 以下我們考慮 B 為一凸單調矩陣。

離線矩陣列最小元素線性算法

- 我們看到一個線上單調矩陣 B 的列最小元素可以在 $O(N \lg N)$ 求出。
- 若 B 是離線的，情形是否會有所不同？
- 離線可能有什麼好處？
- 以下我們考慮 B 為一凸單調矩陣。

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

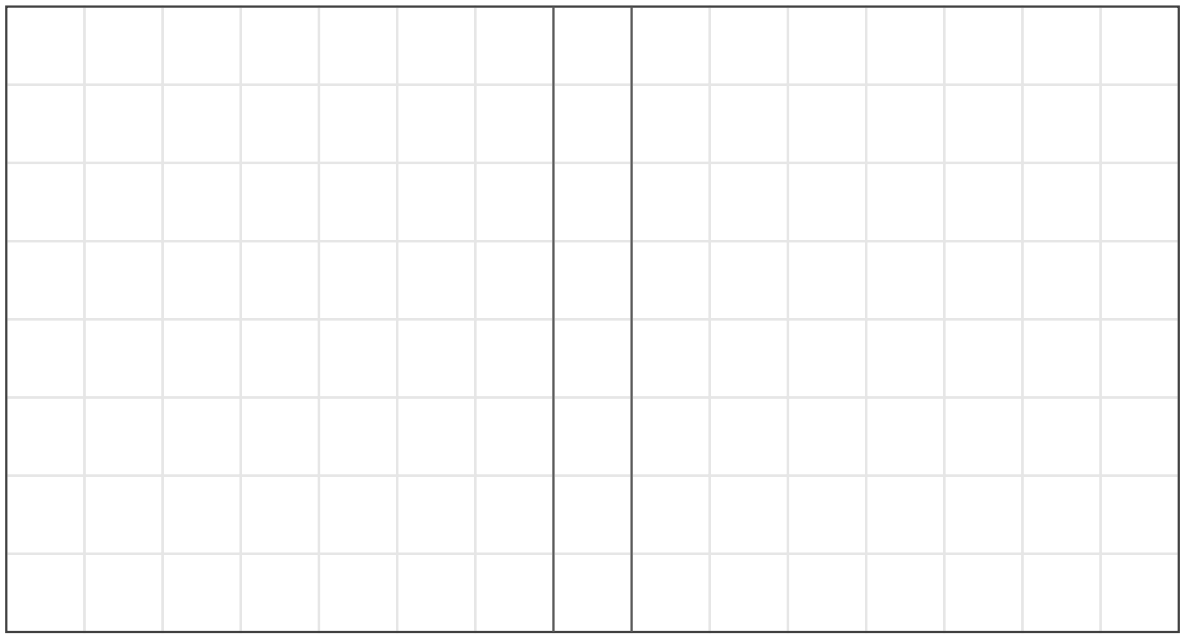


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

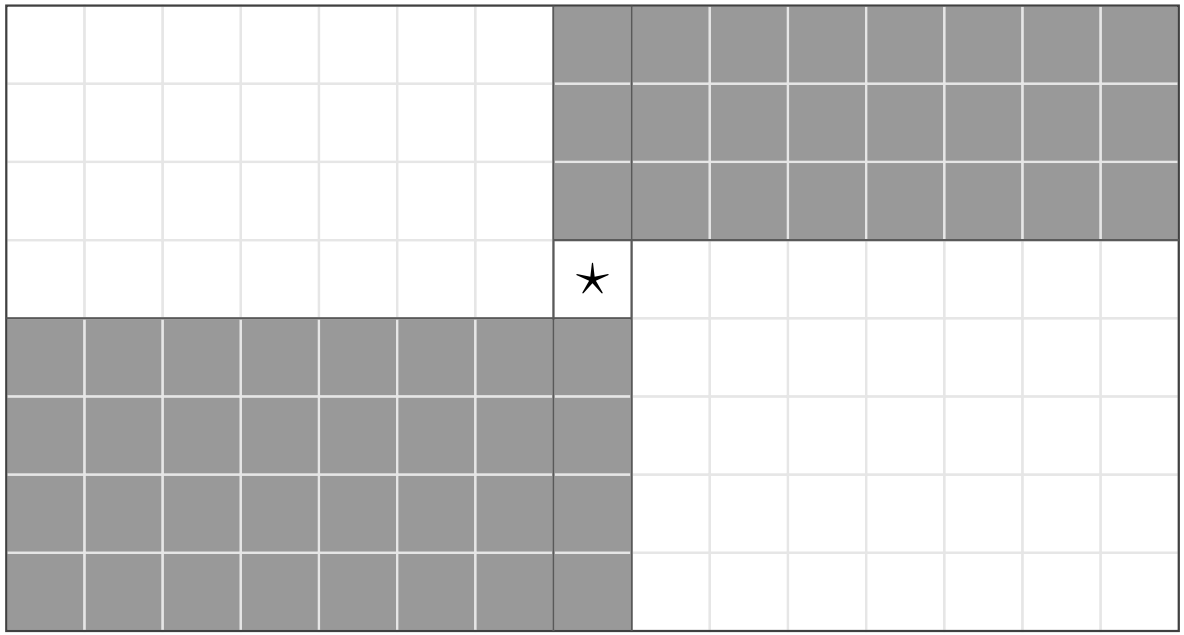


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

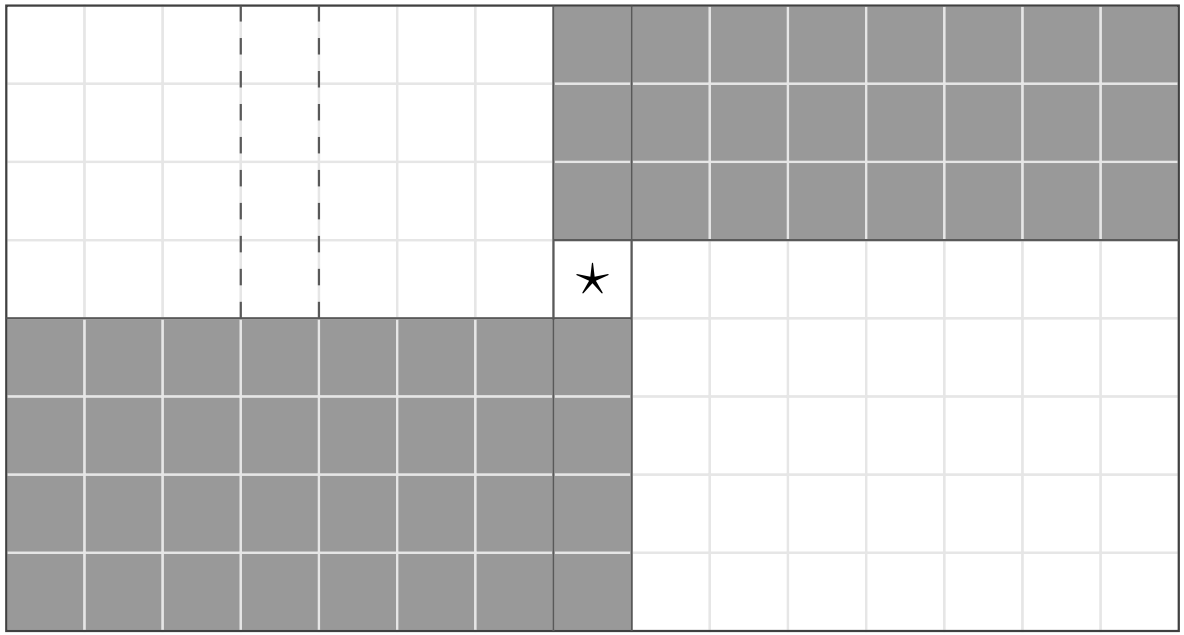


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

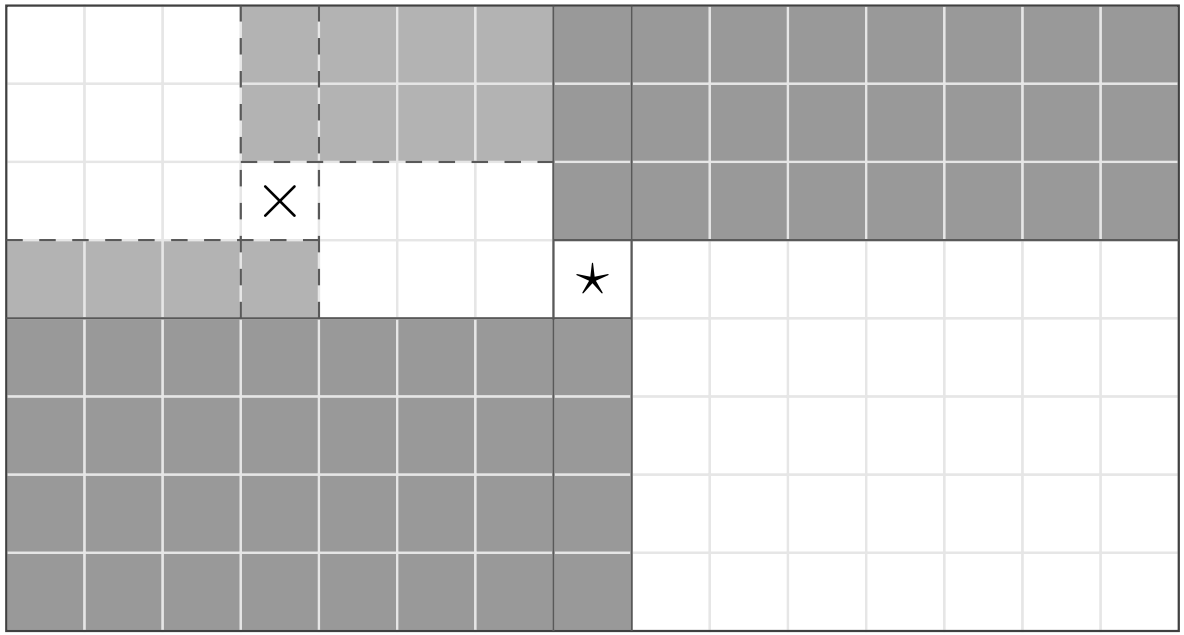


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

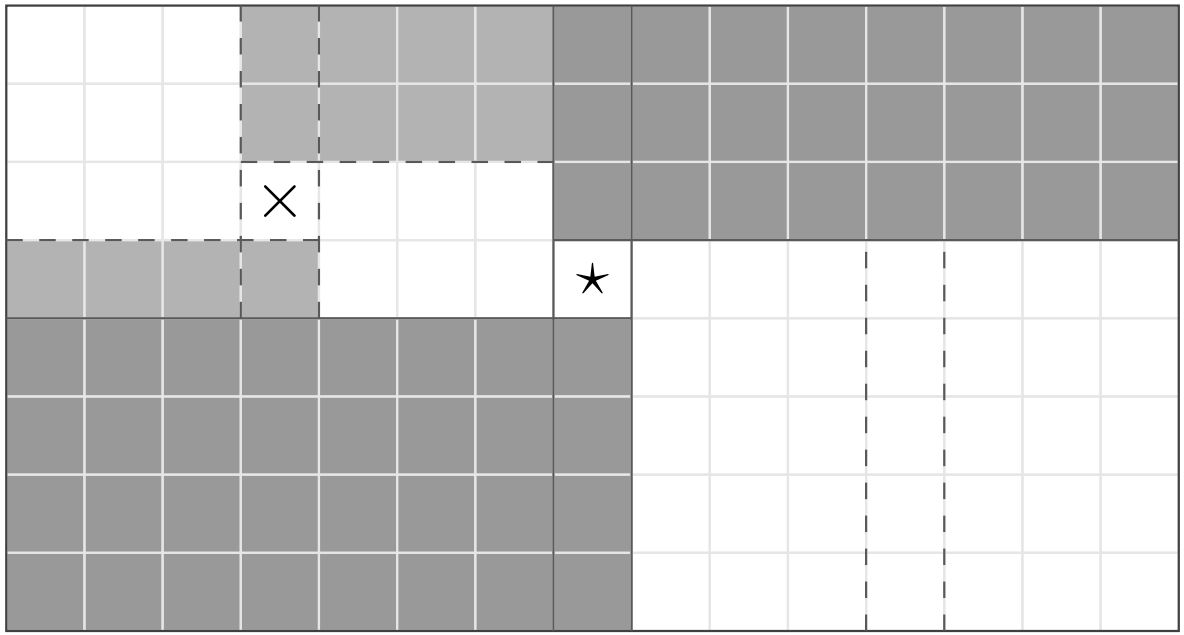


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

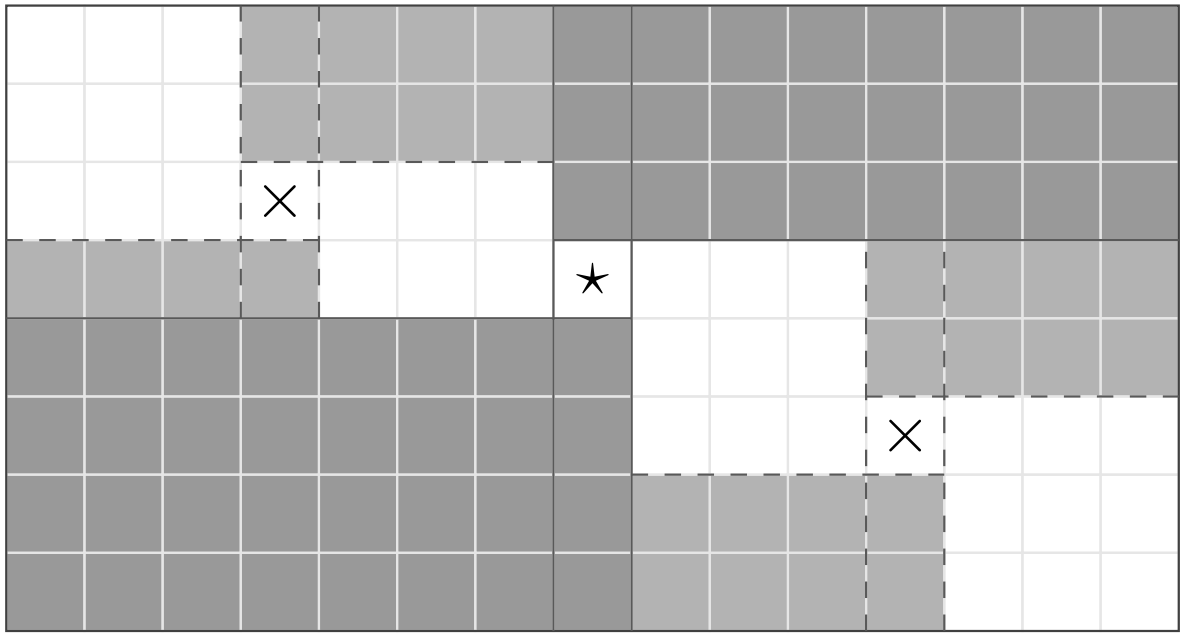


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

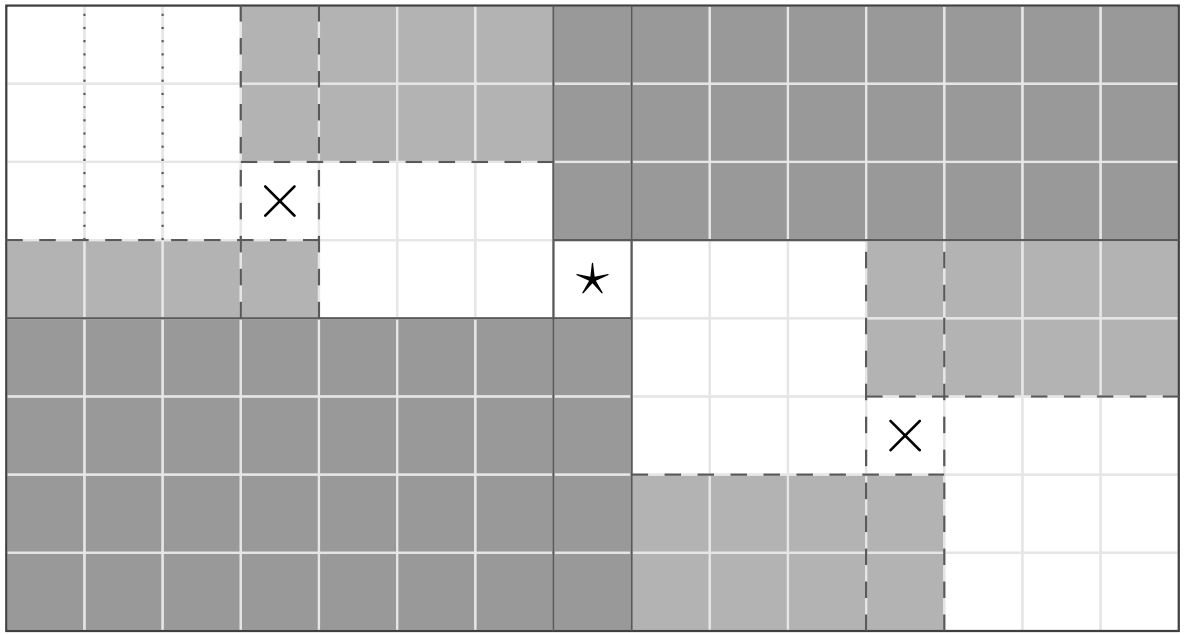


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

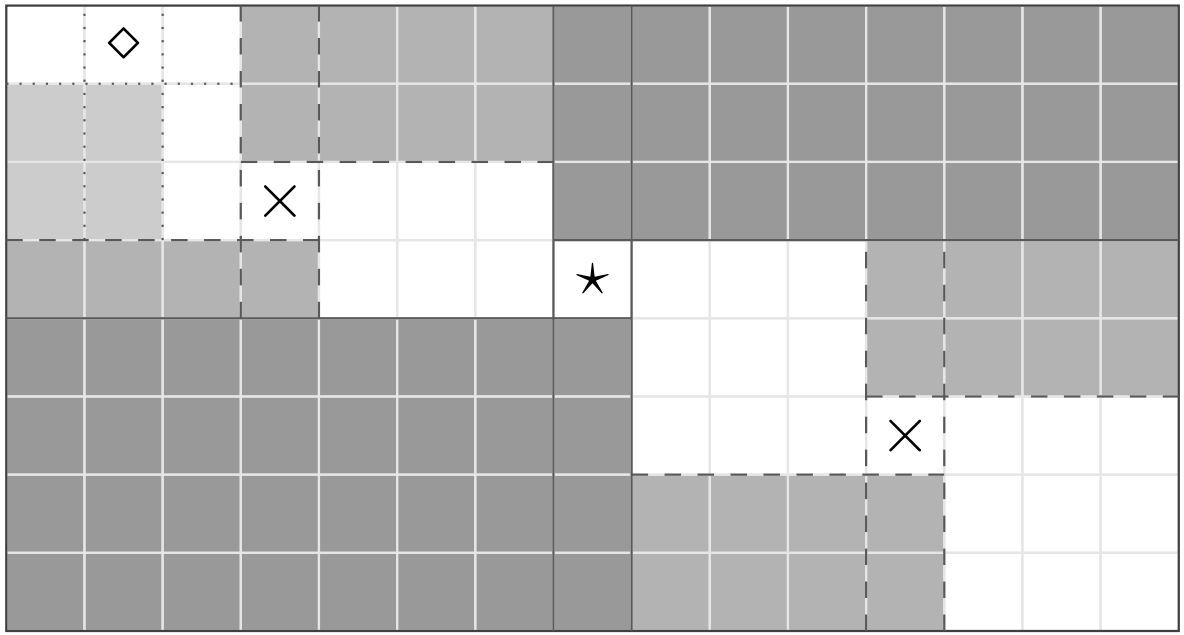


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

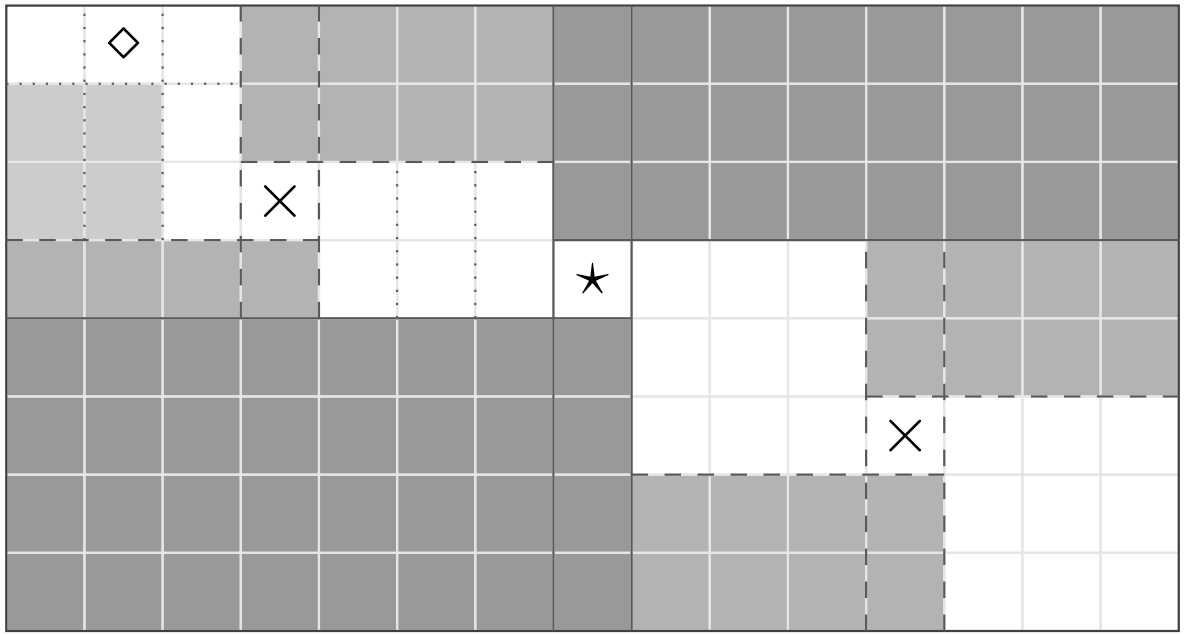


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

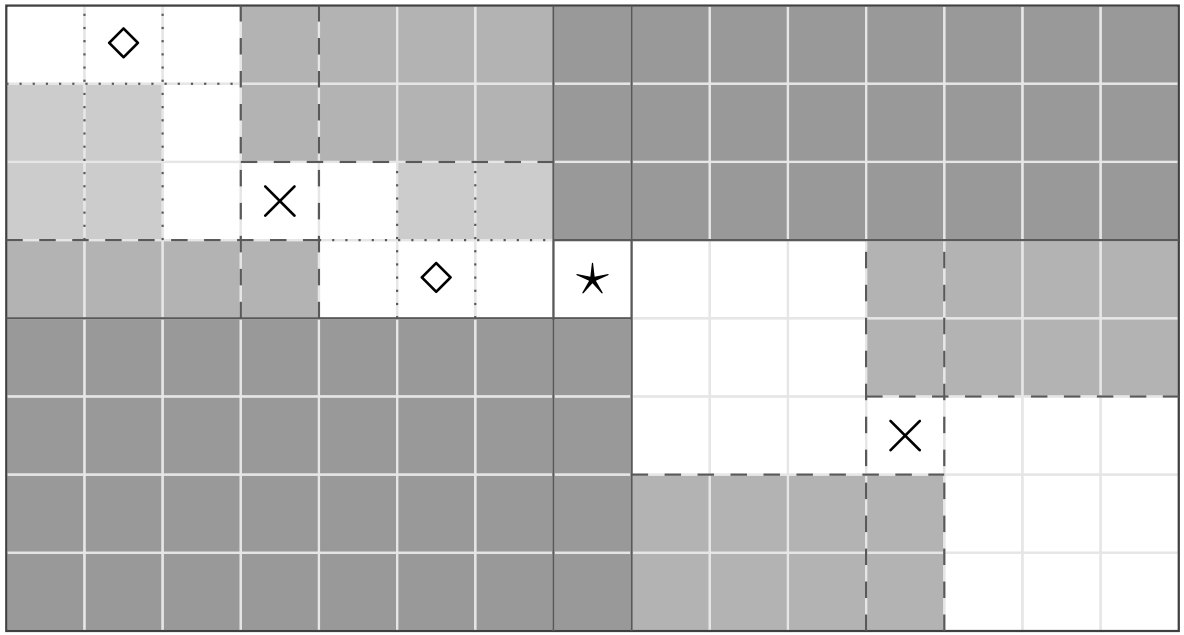


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

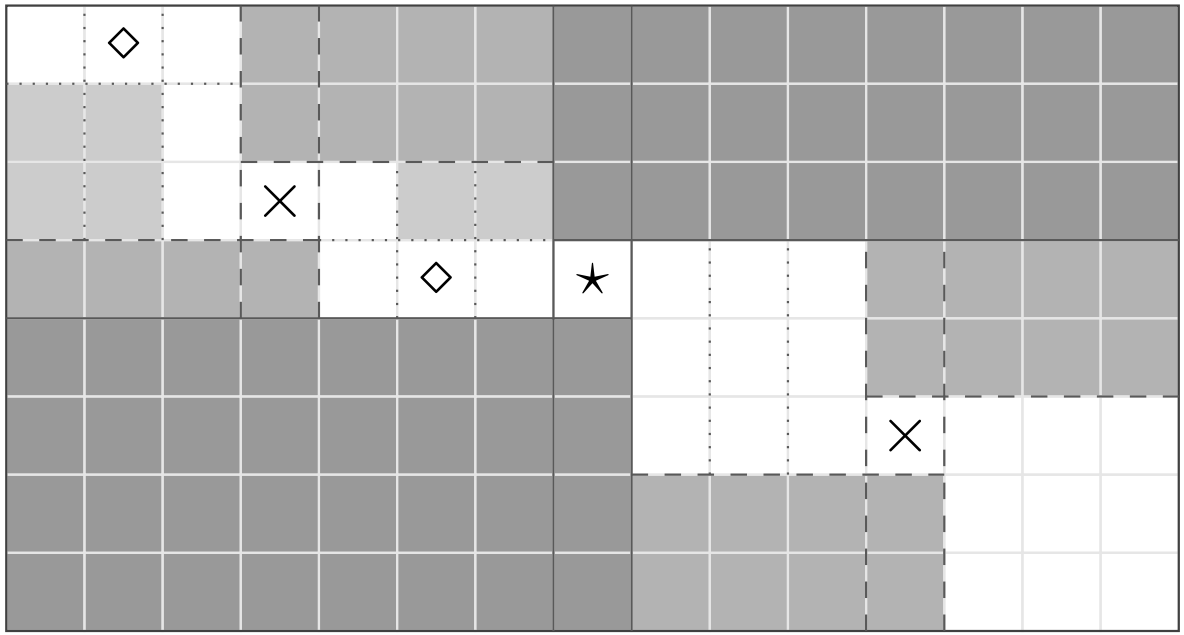


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

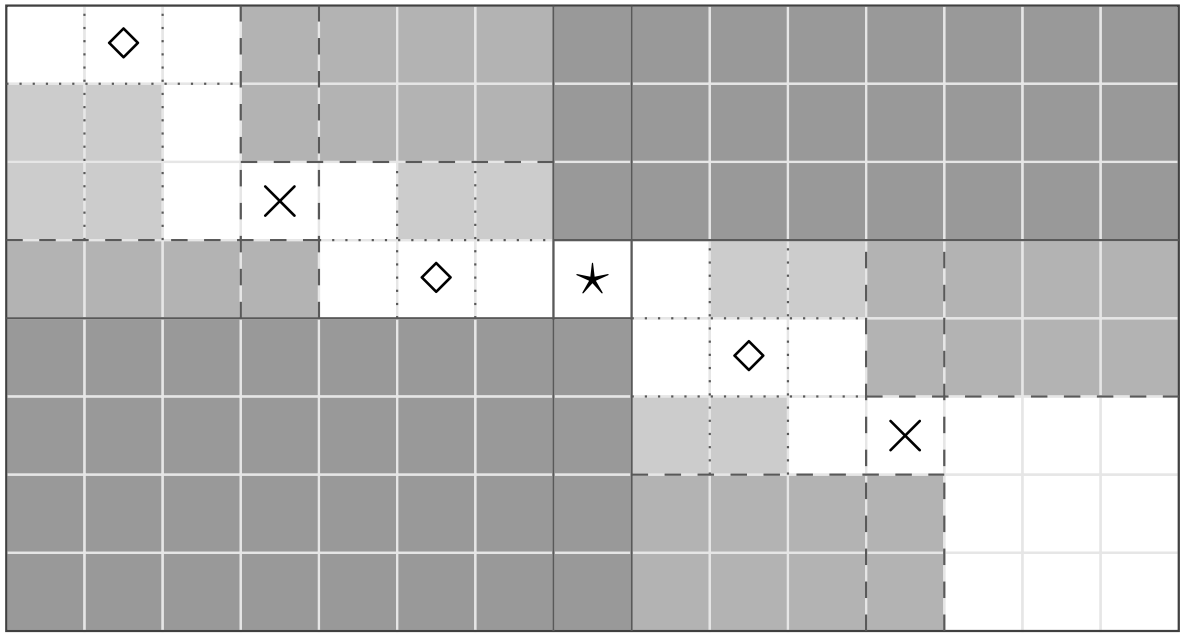


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

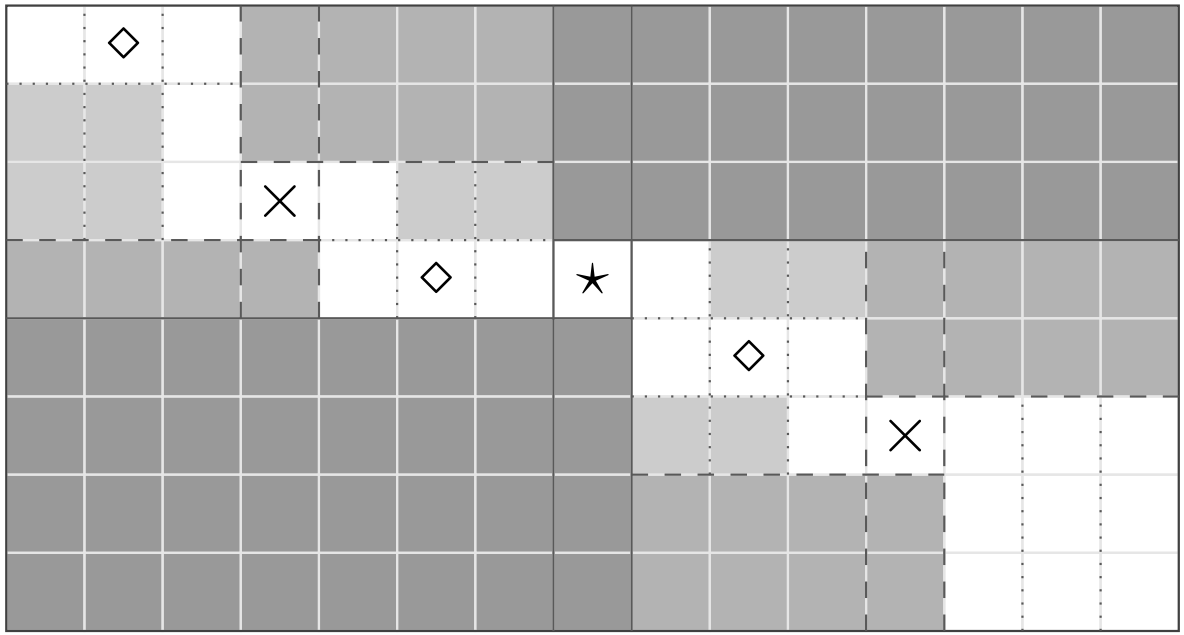


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

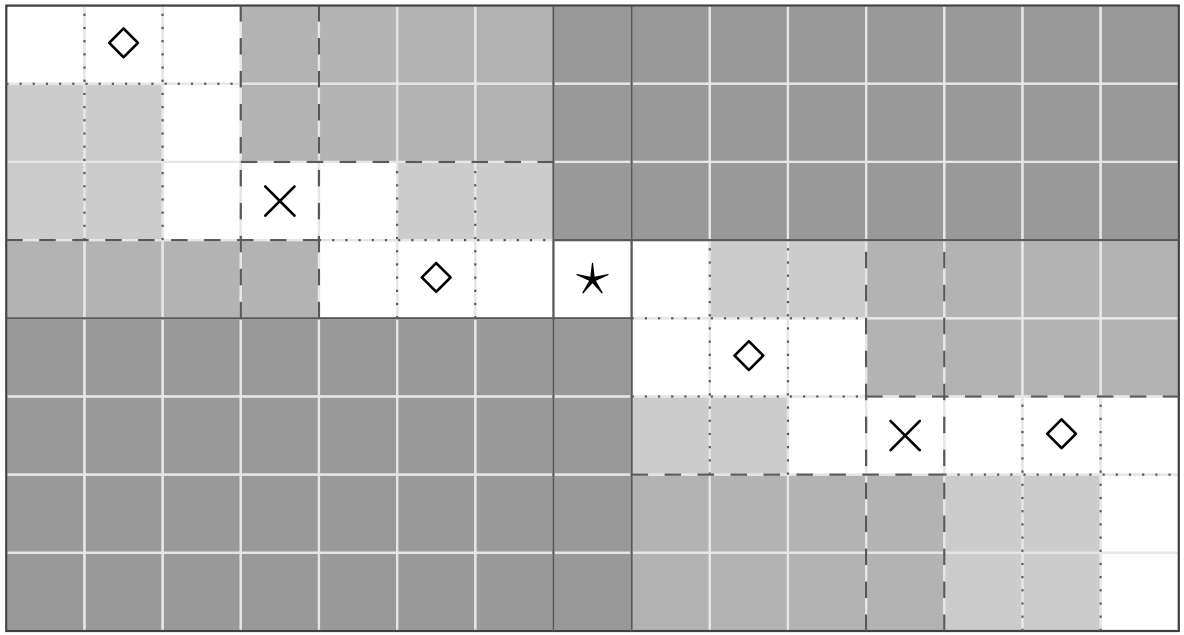


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

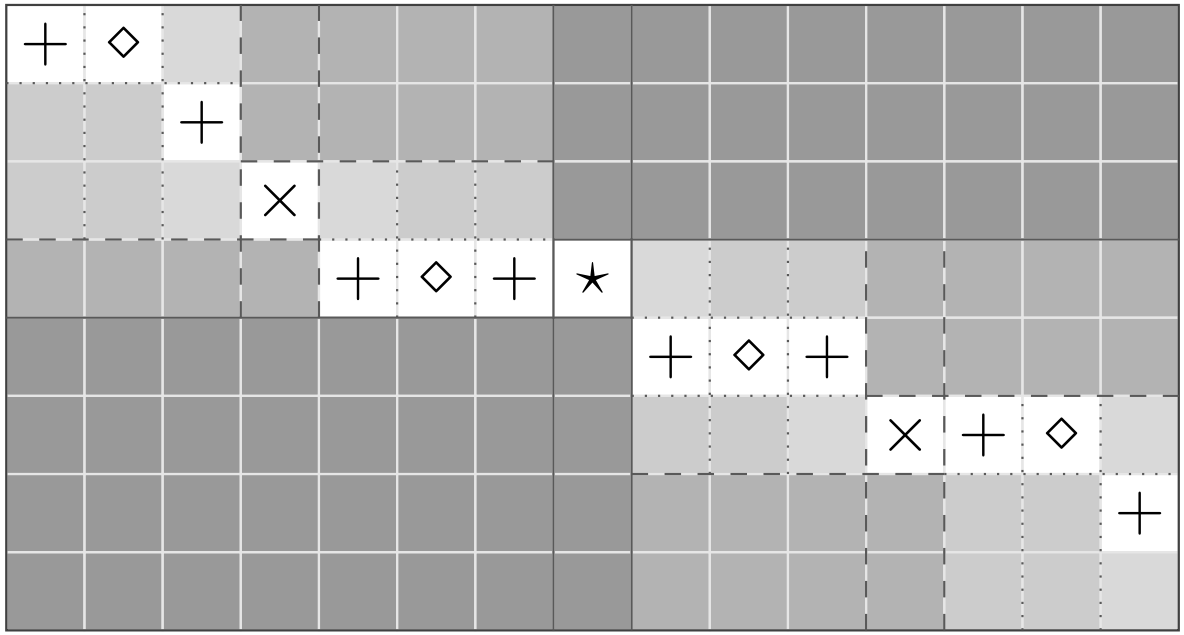


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

- 離線：可分而治之也。

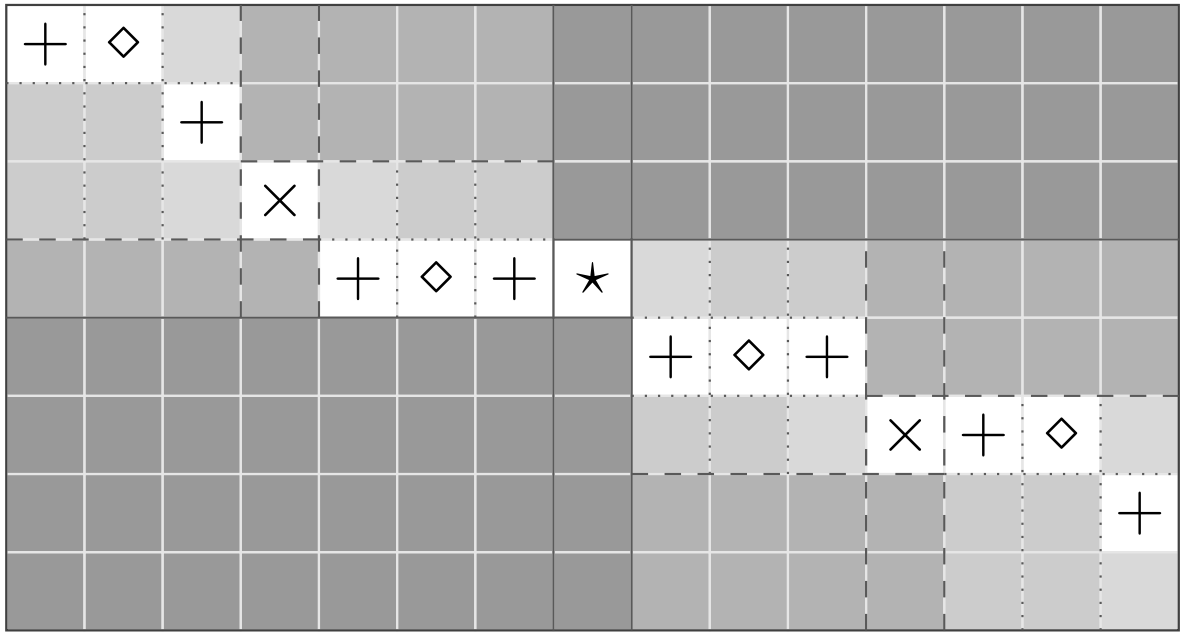


Figure: 分治算法？

離線矩陣列最小元素線性算法｜第一次嘗試

- 離線：可分而治之也。

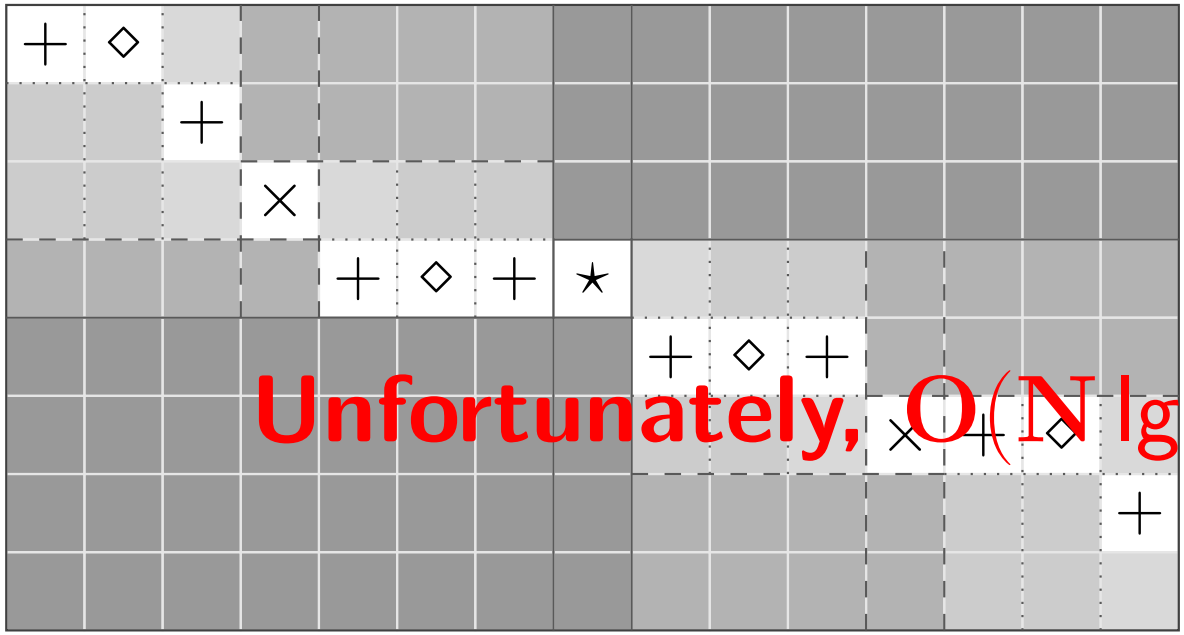


Figure: 分治算法？

離線矩陣列最小元素線性算法 | 第一次嘗試

問題在哪?!

- $T(N) = 2T(N/2) + O(N)$。
- 有沒有辦法一次只需要做一半的子問題呢？

離線矩陣列最小元素線性算法 | 第一次嘗試

問題在哪?!

- $T(N) = 2T(N/2) + O(N)$。
- 有沒有辦法一次只需要做一半的子問題呢？

離線矩陣列最小元素線性算法 | 第一次嘗試

問題在哪?!

- $T(N) = 2T(N/2) + O(N)$。
- 有沒有辦法一次只需要做一半的子問題呢？

離線矩陣列最小元素線性算法 | SMAWK 算法

SMAWK Algorithm

離線矩陣列最小元素線性算法 | SMAWK 算法

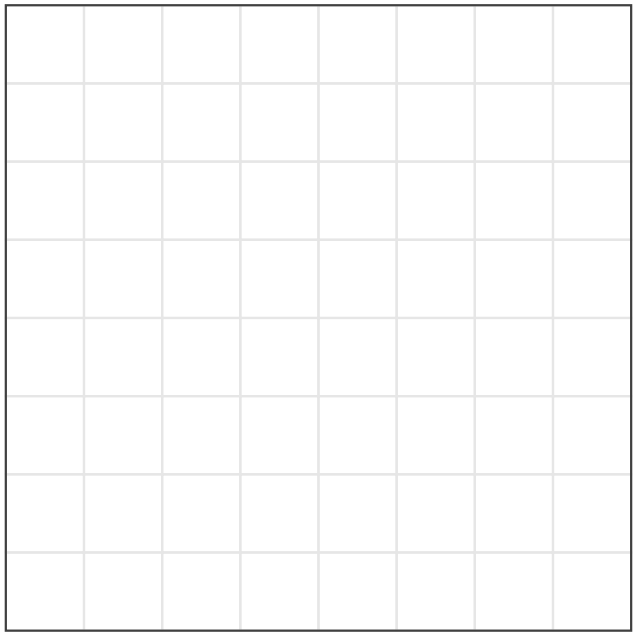


Figure: SMAWK 算法核心想法

離線矩陣列最小元素線性算法 | SMAWK 算法

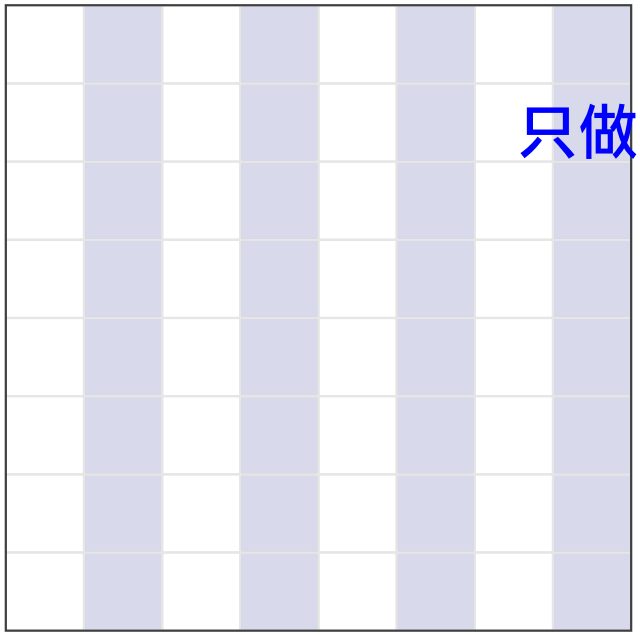


Figure: SMAWK 算法核心想法

離線矩陣列最小元素線性算法 | SMAWK 算法

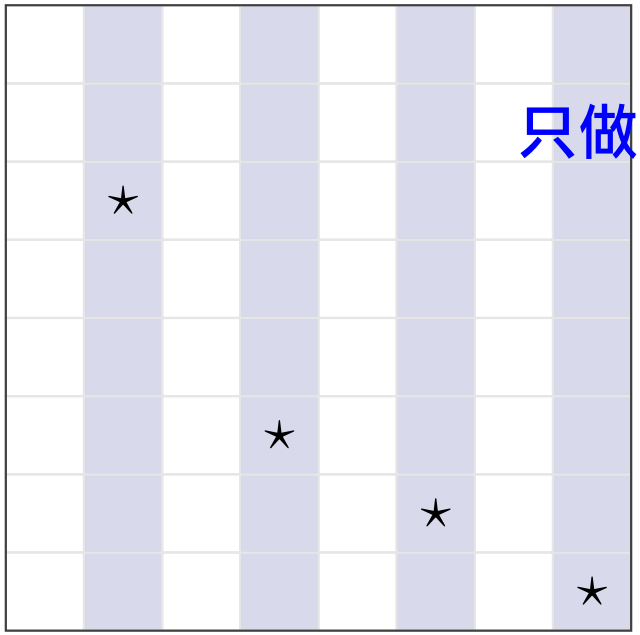


Figure: SMAWK 算法核心想法

離線矩陣列最小元素線性算法 | SMAWK 算法

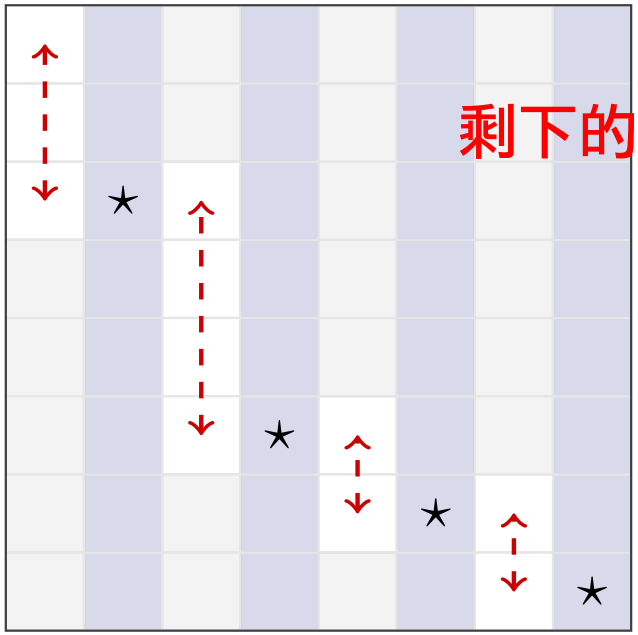


Figure: SMAWK 算法核心想法

離線矩陣列最小元素線性算法 | SMAWK 算法

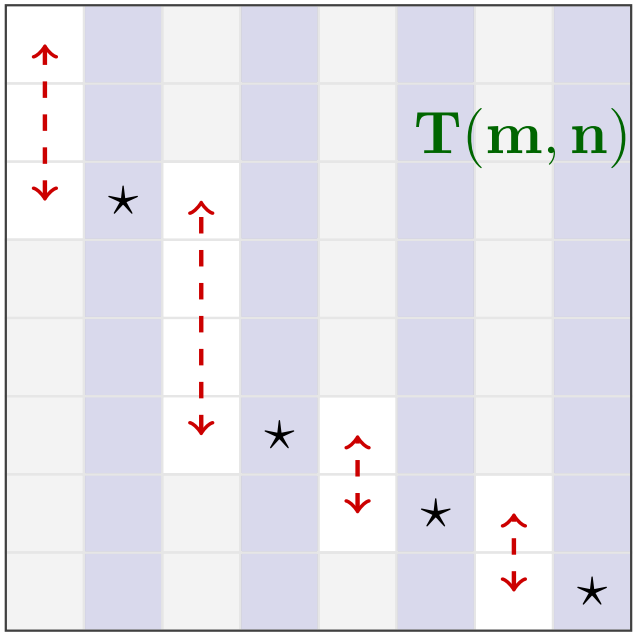


Figure: SMAWK 算法核心想法

離線矩陣列最小元素線性算法 | SMAWK 算法

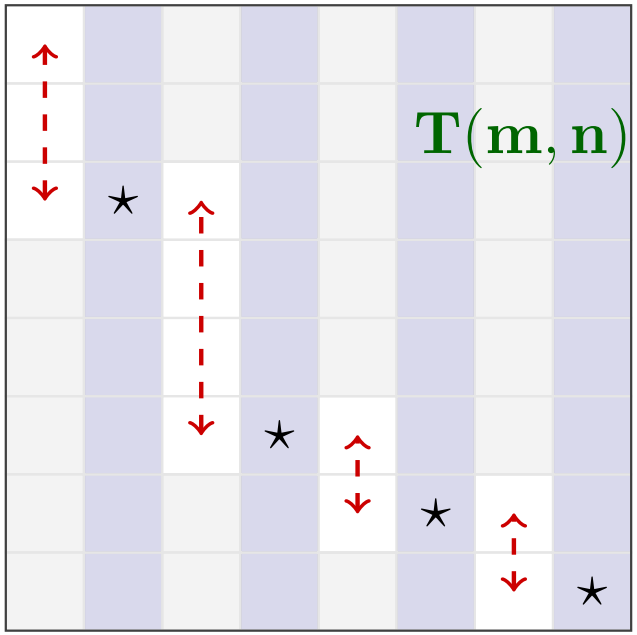


Figure: SMAWK 算法核心想法

離線矩陣列最小元素線性算法 | SMAWK 算法

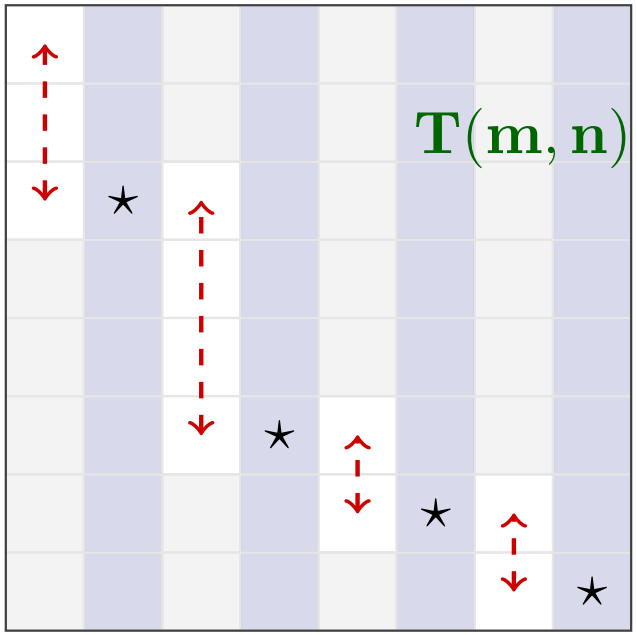


Figure: SMAWK 算法核心想法

離線矩陣列最小元素線性算法 | SMAWK 算法

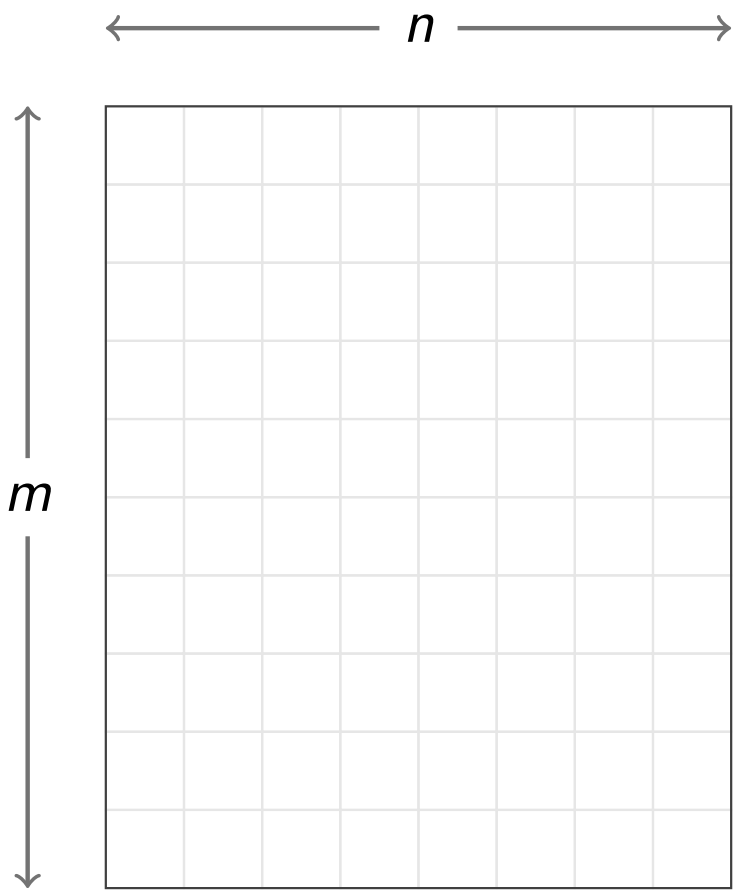
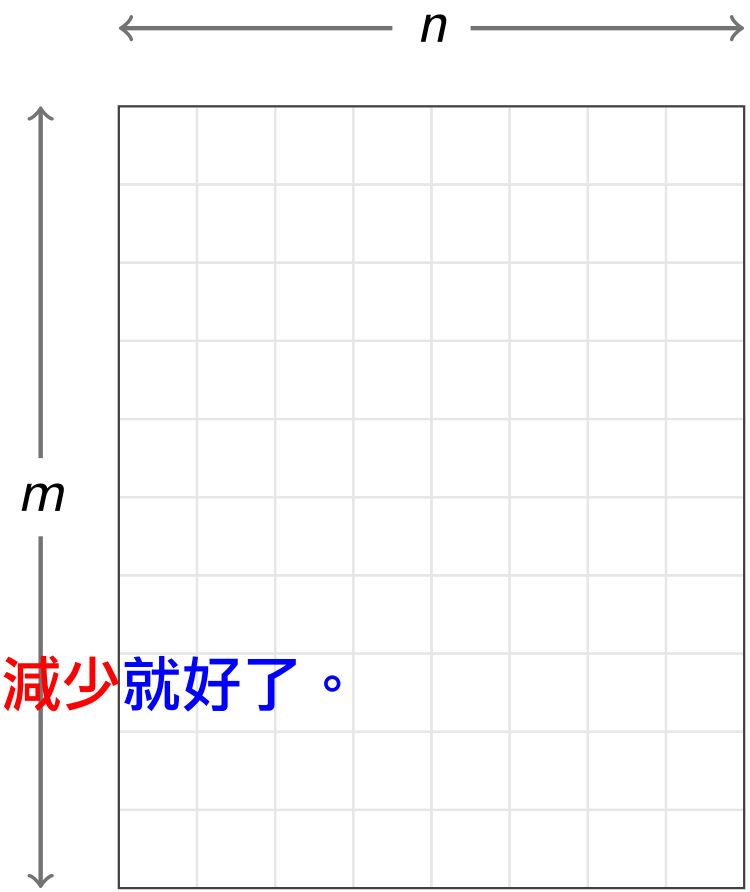


Figure: SMAWK–如何幹掉 “$O(m)$” ?

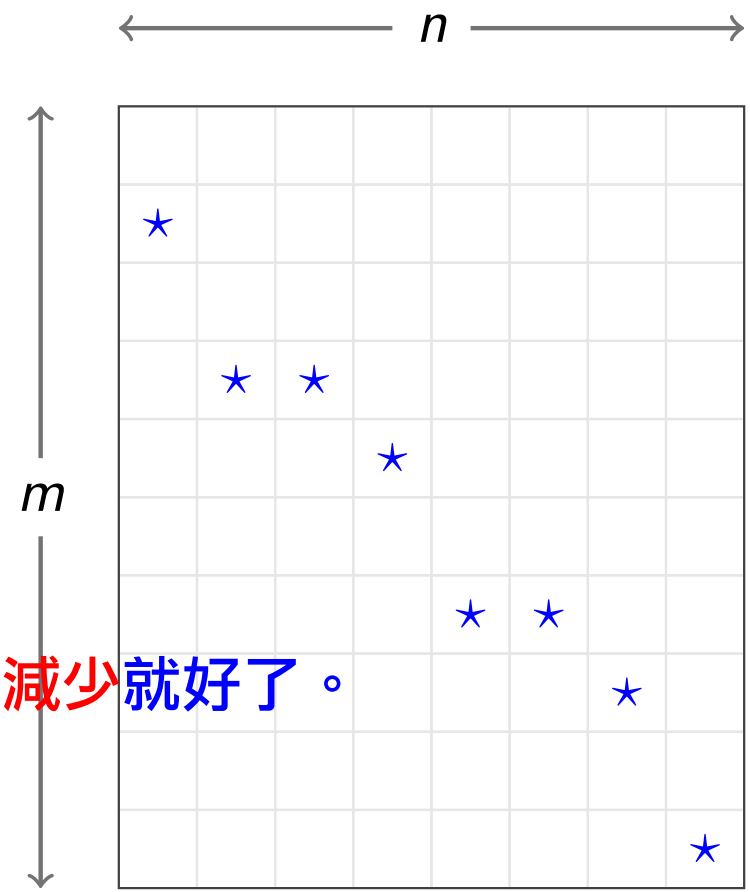
離線矩陣列最小元素線性算法 | SMAWK 算法



如果可以把行數減少就好了。

Figure: SMAWK–如何幹掉 “$O(m)$” ?

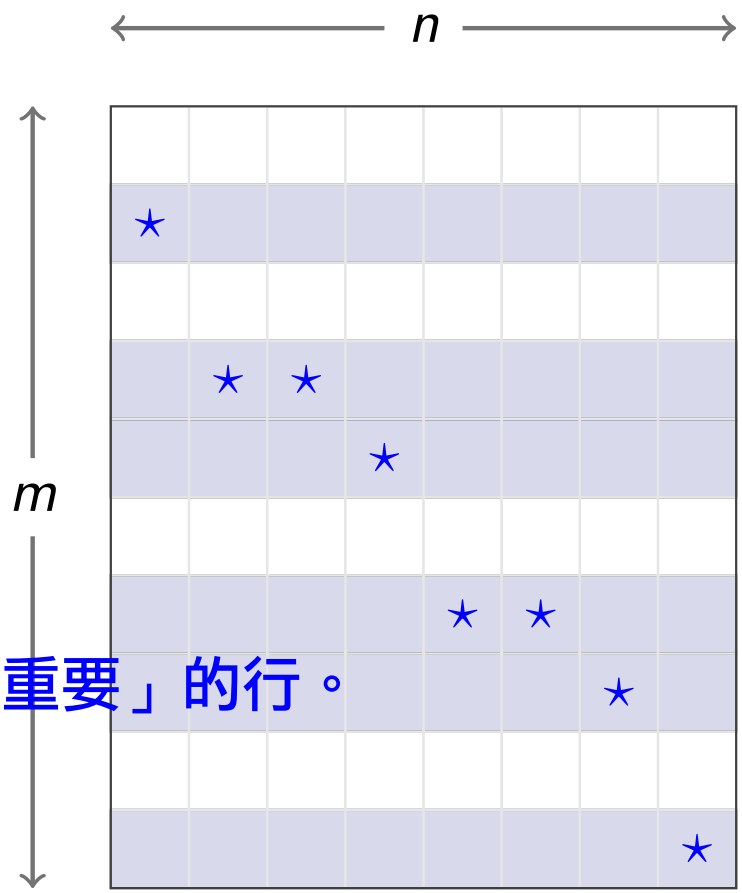
離線矩陣列最小元素線性算法 | SMAWK 算法



如果可以把行數減少就好了。

Figure: SMAWK–如何幹掉 “$O(m)$” ?

離線矩陣列最小元素線性算法 | SMAWK 算法



最多只有 n 個「重要」的行。

Figure: SMAWK–如何幹掉 "$O(m)$"？

離線矩陣列最小元素線性算法 | SMAWK 算法

M A G I C !

離線矩陣列最小元素線性算法 | SMAWK 算法

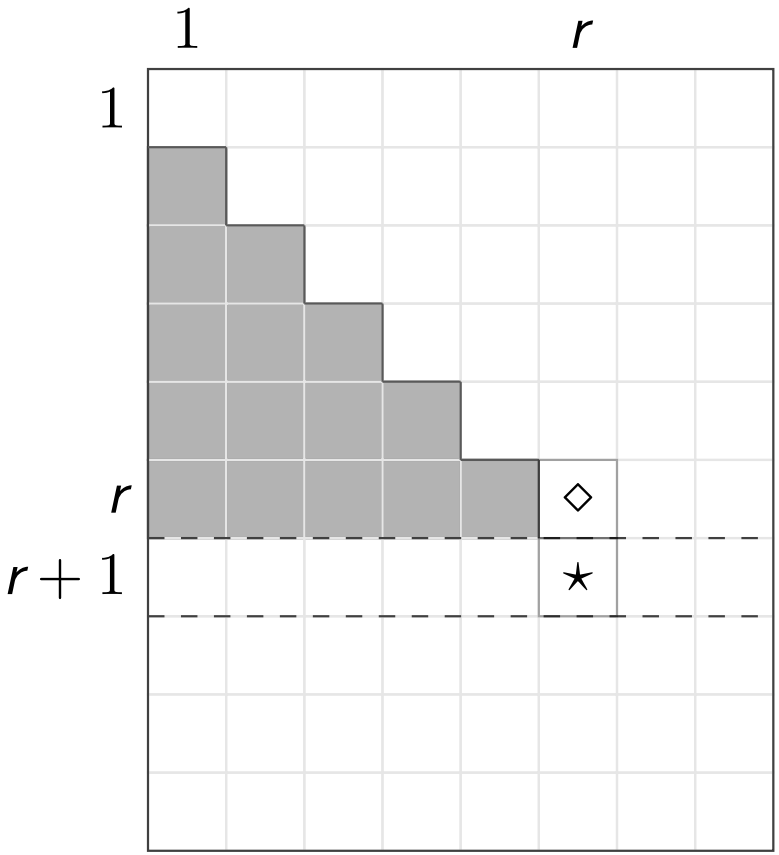


Figure: SMAWK–如何幹掉 “$O(m)$” ?

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 3: SMAWK-Reduce

```
1  function Reduce
2      Q ← B
3      r ← 1
4      while Q has more than n rows do
5          if Q[r+1][r] (⋆) < Q[r][r] (⋄) then // Row r is dead
6              delete row r from Q.
7              if k > 1 then k ← k − 1
8          else if r < n then // Row r+1 added
9              r ← r+1
10         else // Row r+1 is dead
11             delete row r+1 from Q
12         end
13     end
14 end
```

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 3: SMAWK-Reduce

```
1  function Reduce
2      Q ← B
3      r ← 1
4      while Q has more than n rows do
5          if Q[r+1][r] (⋆) < Q[r][r] (⋄) then // Row r is dead
6              delete row r from Q.
7              if k > 1 then k ← k − 1
8          else if r < n then // Row r+1 added
9              r ← r+1
10         else // Row r+1 is dead
11             delete row r+1 from Q
12         end
13     end
14 end
```

做到 Q 剩下不到 n 行為止

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 3: SMAWK-Reduce

```
1  function Reduce
2      Q ← B
3      r ← 1
4      while Q has more than n rows do
5          if Q[r+1][r] (⋆) < Q[r][r] (⋄) then // Row r is dead
6              delete row r from Q.
7              if k > 1 then k ← k − 1
8          else if r < n then // Row r+1 added
9              r ← r+1
10         else // Row r+1 is dead
11             delete row r+1 from Q
12         end
13     end
14 end
```

第 r 行確定「掰了」。

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 3: SMAWK-Reduce

```
1  function Reduce
2      Q ← B
3      r ← 1
4      while Q has more than n rows do
5          if Q[r+1][r] (⋆) < Q[r][r] (⋄) then // Row r is dead
6              delete row r from Q.
7              if k > 1 then k ← k − 1
8          else if r < n then // Row r+1 added
9              r ← r+1
10         else // Row r+1 is dead
11             delete row r+1 from Q
12         end
13     end
14 end
```

新的那行仍可能包含列最小值，可不破壞 loop-invariant 地加入。

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 3: SMAWK-Reduce

```
1  function Reduce
2      Q ← B
3      r ← 1
4      while Q has more than n rows do
5          if Q[r+1][r] (⋆) < Q[r][r] (⋄) then // Row r is dead
6              delete row r from Q.
7              if k > 1 then k ← k − 1
8          else if r < n then // Row r+1 added
9              r ← r+1
10         else // Row r+1 is dead
11             delete row r+1 from Q
12         end
13     end
14 end
```

在已經靠邊（維護 n 行）的情況下，新的一行完全「掰了」。

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 3: SMAWK-Reduce

```
1  function Reduce
2      Q ← B
3      r ← 1
4      while Q has more than n rows do
5          if Q[r+1][r] (⋆) < Q[r][r] (⋄) then // Row r is dead
6              delete row r from Q.
7              if k > 1 then k ← k − 1
8          else if r < n then // Row r+1 added
9              r ← r+1
10         else // Row r+1 is dead
11             delete row r+1 from Q
12         end
13     end
14 end
```

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 4: Column-Minima

```
1  function Column-Minima(A)
2      if n=1 then return column minimum (by linear sweep)
3      A ← Reduce(A)
4      A' ← A[:; 1, 3, 5, ...]
5      ans = {(1, r_1), (3, r_3), (5, r_5), ...} ← Column-Minima(A')
6      for j ← 2, 4, 6, ... do
7          // Minimum of column j could only be between row
           r_{j-1} to r_{j+1}
8          r_j ← argmin_r {A[r][j] | r_{j+1} ≤ r ≤ r_{j-1}}
9          ans ← ans ∪ (j, r_j)
10     end
11     return ans;
12 end
```

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 4: Column-Minima

```
1  function Column-Minima(A)
2      if n=1 then return column minimum (by linear sweep)
3      A ← Reduce(A)
4      A' ← A[:; 1, 3, 5, ...]
5      ans = {(1, r_1), (3, r_3), (5, r_5), ...} ← Column-Minima(A')
6      for j ← 2, 4, 6, ... do
7          // Minimum of column j could only be between row
             r_{j-1} to r_{j+1}
8          r_j ← argmin_r {A[r][j] | r_{j+1} ≤ r ≤ r_{j-1}}
9          ans ← ans ∪ (j, r_j)
10     end
11     return ans;
12 end
```

遞迴初始邊界。

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 4: Column-Minima

```
1  function Column-Minima(A)
2      if n=1 then return column minimum (by linear sweep)
3      A ← Reduce(A)
4      A' ← A[:; 1, 3, 5, ...]
5      ans = {(1, r_1), (3, r_3), (5, r_5), ...} ← Column-Minima(A')
6      for j ← 2, 4, 6, ... do
7          // Minimum of column j could only be between row
             r_{j-1} to r_{j+1}
8          r_j ← argmin_r {A[r][j] | r_{j+1} ≤ r ≤ r_{j-1}}
9          ans ← ans ∪ (j, r_j)
10     end
11     return ans;
12 end
```

使用 Reduce 大法確保 A 的行數不超過列數。

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 4: Column-Minima

```
1  function Column-Minima(A)
2      if n=1 then return column minimum (by linear sweep)
3      A ← Reduce(A)
4      A' ← A[:; 1, 3, 5, ...]
5      ans = {(1, r_1), (3, r_3), (5, r_5), ...} ← Column-Minima(A')
6      for j ← 2, 4, 6, ... do
7          // Minimum of column j could only be between row
             r_{j-1} to r_{j+1}
8          r_j ← argmin_r {A[r][j] | r_{j+1} ≤ r ≤ r_{j-1}}
9          ans ← ans ∪ (j, r_j)
10     end
11     return ans;
12 end
```

對 A' 的偶數行子矩陣遞迴求列最小元素。

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 4: Column-Minima

```
1  function Column-Minima(A)
2      if n=1 then return column minimum (by linear sweep)
3      A ← Reduce(A)
4      A' ← A[ : ; 1, 3, 5, . . . ]
5      ans = {(1, r_1), (3, r_3), (5, r_5), . . . } ← Column-Minima(A')
6      for j ← 2, 4, 6, . . . do
7          // Minimum of column j could only be between row
           r_{j-1} to r_{j+1}
8          r_j ← argmin_r {A[r][j] | r_{j+1} ≤ r ≤ r_{j-1}}
9          ans ← ans ∪ (j, r_j)
10     end
11     return ans;
12 end
```

對 A' 的偶數行子矩陣遞迴求列最小元素。

離線矩陣列最小元素線性算法 | SMAWK 算法

Algorithm 4: Column-Minima

```
1  function Column-Minima(A)
2      if n=1 then return column minimum (by linear sweep)
3      A ← Reduce(A)
4      A' ← A[:; 1, 3, 5, ...]
5      ans = {(1, r_1), (3, r_3), (5, r_5), ...} ← Column-Minima(A')
6      for j ← 2, 4, 6, ... do
7          // Minimum of column j could only be between row
           r_{j-1} to r_{j+1}
8          r_j ← argmin_r {A[r][j] | r_{j+1} ≤ r ≤ r_{j-1}}
9          ans ← ans ∪ (j, r_j)
10     end
11     return ans;
12 end
```

離線矩陣列最小元素線性算法 | SMAWK 算法

回顧一下我們做了什麼：

- 問題：給一離線單調的 $m \times n$ 矩陣，求其每列最小值。
- 我們希望可以單邊遞迴。方法是只挑偶數行來做，奇數行的部分利用偶數行部份回傳的列最小元素可以均攤線性求得。
- 分下去 (或一開始) 的子問題有可能發生高 $>$ 寬，此時就需要召喚神奇的 REDUCE 在線性時間挑出真的有用的 n 行。
- $T(m, n) = O(m + n) + T(\frac{n}{2}, n)$, 令 $N = m + n$ ：
  $\Rightarrow T(N) = O(N) + O(T(\frac{N}{2}, N))$
  $\Rightarrow T(N) = O(N)$

離線矩陣列最小元素線性算法 | SMAWK 算法

回顧一下我們做了什麼：

- 問題：給一離線單調的 $m \times n$ 矩陣，求其每列最小值。
- 我們希望可以單邊遞迴。方法是只挑偶數行來做，奇數行的部分利用偶數行部份回傳的列最小元素可以均攤線性求得。
- 分下去 (或一開始) 的子問題有可能發生高 $>$ 寬，此時就需要召喚神奇的 REDUCE 在線性時間挑出真的有用的 n 行。
- $T(m, n) = O(m + n) + T(\frac{n}{2}, n)$, 令 $N = m + n$ ：
  $\Rightarrow T(N) = O(N) + O(T(\frac{N}{2}, N))$
  $\Rightarrow T(N) = O(N)$

離線矩陣列最小元素線性算法 | SMAWK 算法

回顧一下我們做了什麼：

- 問題：給一離線單調的 $m \times n$ 矩陣，求其每列最小值。
- 我們希望可以單邊遞迴。方法是只挑偶數行來做，奇數行的部分利用偶數行部份回傳的列最小元素可以均攤線性求得。
- 分下去 (或一開始) 的子問題有可能發生高 $>$ 寬，此時就需要召喚神奇的 REDUCE 在線性時間挑出真的有用的 n 行。
- $T(m, n) = O(m + n) + T(\frac{n}{2}, n)$, 令 $N = m + n$ ：
  $\Rightarrow T(N) = O(N) + O(T(\frac{N}{2}, N))$
  $\Rightarrow T(N) = O(N)$

離線矩陣列最小元素線性算法 | SMAWK 算法

回顧一下我們做了什麼：

- 問題：給一離線單調的 $m \times n$ 矩陣，求其每列最小值。
- 我們希望可以單邊遞迴。方法是只挑偶數行來做，奇數行的部分利用偶數行部份回傳的列最小元素可以均攤線性求得。
- 分下去 (或一開始) 的子問題有可能發生高 $>$ 寬，此時就需要召喚神奇的 REDUCE 在線性時間挑出真的有用的 n 行。
- $T(m,n) = O(m+n) + T(\frac{n}{2}, n)$, 令 $N = m+n$ ：
  $\Rightarrow T(N) = O(N) + O(T(\frac{N}{2}, N))$
  $\Rightarrow T(N) = O(N)$

離線矩陣列最小元素線性算法 | SMAWK 算法

回顧一下我們做了什麼：

- 問題：給一離線單調的 $m \times n$ 矩陣，求其每列最小值。
- 我們希望可以單邊遞迴。方法是只挑偶數行來做，奇數行的部分利用偶數行部份回傳的列最小元素可以均攤線性求得。
- 分下去 (或一開始) 的子問題有可能發生高 $>$ 寬，此時就需要召喚神奇的 REDUCE 在線性時間挑出真的有用的 n 行。
- $T(m,n) = O(m+n) + T(\frac{n}{2}, n)$, 令 $N = m+n$ ：
  $\Rightarrow T(N) = O(N) + O(T(\frac{N}{2}, N))$
  $\Rightarrow T(N) = O(N)$

離線矩陣列最小元素線性算法 | SMAWK 算法

$O(N)$ 斯米打！

天真啊

賊哈哈哈哈我附身在凸包身上並且用她的肉體當作我靈魂的載具當她良善的一面被我吞噬殆盡當渾沌的呢喃超出理智所能負荷我毀面之王1D-1D將再度君臨世界因為我是傲嬌的大魔王為了讓你們勇者有事做我當然不能說聲呃阿就死了嘛說實在話你沒看過無敵破壞王嗎當惡人也是很辛苦的如果我有選擇你你你以為我不想成為正義的一方要和惡勢力來對抗有智慧有膽量越戰越堅強唉呀不過說了那麼多反正該做的事還是要做那就是我將以更新更酷更燒毀的姿態回歸折磨你的愛情邊界你叫破喉嚨也不會有人來救你的破喉嚨破喉嚨哈囉我來囉！不要啊臣亮言先帝創業未半而中道崩殂今天下三分益州疲敝此誠危急存亡之秋也然侍衛之臣不懈于內忠志之士忘身于外者蓋追先帝之殊遇欲報之于陛下也對不起驚恐之下就背出了出師表我想說的是念在你也是一代宗師演算法好像很厲害的樣子我倒數十秒鐘你趕快逃跑好了不要再來找我麻煩啦所謂投降輸一半感情不會散你現在投降我就算你有歐根號恩的解好了你找得到別家更便宜的我算你半價二分之一歐根號恩好了認真不騙什麼喂喂喂你別轉身就走阿我這麼好說歹說你當耳邊風我說你這人還有血有淚有點心肝麼算我求你好了別管什麼恩漏個恩了人要知足常樂恩平方不好嗎我們聊點別的吧你聽過安麗嗎安麗沒有的話科氏力有沒有聽過賊哈哈哈哈我附身在凸包身上並且用她的肉體當作我靈魂的載具當她良善的一面被我吞噬殆盡當渾沌的呢喃超出理智所能負荷我毀面之王將再度君臨世界因為我是傲嬌的大魔王為了讓你們勇者有事做我當然

還沒完呢！

凸包優化 DP

Convex Hull Trick

例題：Machine Works

你是一位生產線管理員，夢想是賺大錢。

現在你打聽到了一些消息：有 N 台生產機器，分別在第 t_i 天會被拍賣。機器只能在被拍賣的那天被買走，過了那天之後機器就會下架，之後再也買不到。

每台機器有它的購買價格 p_i 與轉售價格 r_i $(r_i \leq p_i)$：在拍賣那天你可以用 p_i 元買下它，之後任何時候你可以用 r_i 元的價格轉賣它。除此以外每台機器有其賺錢效率 g_i：當你擁有該機器時，每天可以賺進 g_i 元。

一開始你有 B 元的本金，且你一次只能擁有一台機器。問在 T 天後透過買進機器工作 (以及賣出機器)，你最多能賺多少錢？

$(1 \leq N \leq 10^5, 1 \leq r_i, p_i, g_i, B, T \leq 10^9)$

例題：Machine Works

分析：

- 不失一般性地假設機器是按照時間順序給的，亦即 $t_1 \leq t_2 \leq \cdots \leq t_N$。
- 直觀的 $O(N^2)$ DP：
  $DP[i] :=$ 買機器 i 並賣掉手上機器後最多賺多少錢。
- 轉移：
  $DP[i] = \max_j \{\$ - p_i + r_i \mid \$ \geq p_i\}$
  $\$ =$ 到時間 i 時有多少錢 $= DP[j] + g_j(t_i - t_j)$

例題：Machine Works

分析：

- 不失一般性地假設機器是按照時間順序給的，亦即 $t_1 \leq t_2 \leq \cdots \leq t_N$。
- 直觀的 $O(N^2)$ DP：
  $DP[i] :=$ 買機器 i 並賣掉手上機器後最多賺多少錢。
- 轉移：
  $DP[i] = \max_j \{\$ - p_i + r_i \mid \$ \geq p_i\}$
  $\$ =$ 到時間 i 時有多少錢 $= DP[j] + g_j(t_i - t_j)$

例題：Machine Works

分析：

- 不失一般性地假設機器是按照時間順序給的，亦即 $t_1 \leq t_2 \leq \cdots \leq t_N$。
- 直觀的 $O(N^2)$ DP：
  $DP[i] :=$ 買機器 i 並賣掉手上機器後最多賺多少錢。
- 轉移：
  $DP[i] = \max_j \{\$ - p_i + r_i \mid \$ \geq p_i\}$
  $\$ =$ 到時間 i 時有多少錢 $= DP[j] + g_j(t_i - t_j)$

例題：Machine Works

分析：

- 不失一般性地假設機器是按照時間順序給的，亦即 $t_1 \leq t_2 \leq \cdots \leq t_N$。
- 直觀的 $O(N^2)$ DP：
  $DP[i] :=$ 買機器 i 並賣掉手上機器後最多賺多少錢。
- 轉移：
  $DP[i] = \max_j \{\$ - p_i + r_i \mid \$ \geq p_i\}$
  $\$ =$ 到時間 i 時有多少錢 $= DP[j] + g_j(t_i - t_j)$

例題：Machine Works

分析：

- $DP[i] = \max_j \{\$ - p_i + r_i \mid \$ \geq p_i\}$
- 照時序做的話，DP 轉移並沒有單調性。
  $\rightarrow$ 單調性是來自賺錢效率的。
- 沒招了？

例題：Machine Works

分析：

- $DP[i] = \max_j \{\$ - p_i + r_i \mid \$ \geq p_i\}$
- **照時序做的話，DP 轉移並沒有單調性。**
  $\rightarrow$ 單調性是來自賺錢效率的。
- 沒招了？

例題：Machine Works

分析：

- $DP[i] = \max_j \{\$ - p_i + r_i \mid \$ \geq p_i\}$
- **照時序做的話，DP 轉移並沒有單調性。**
  $\rightarrow$ **單調性是來自賺錢效率的。**
- **沒招了？**

例題：Machine Works | 凸包優化 DP

凸包優化 DP

例題：Machine Works | 凸包優化 DP

- 可用直線 $y_i(t) = y_0 + g_i t$ 表示買了一台機器後的成長。
- 對於一個時間點 t_i 來說，我們需要知道的正是：$max_{j<i}\{\$_{j\to i}\} = max_{j<i}\{y_j(t_i)\}$。

例題：Machine Works | 凸包優化 DP

- 可用**直線** $y_i(t) = y_0 + g_i t$ 表示買了一台機器後的成長。
- 對於一個時間點 t_i 來說，我們需要知道的正是：$\max_{j<i}\{\$_{j\to i}\} = \max_{j<i}\{y_j(t_i)\}$。

例題：Machine Works | 凸包優化 DP

- 可用**直線** $y_i(t) = y_0 + g_i t$ 表示買了一台機器後的成長。
- 對於一個時間點 t_i 來說，我們需要知道的正是：$\max_{j<i}\{\$_{j\to i}\} = \max_{j<i}\{y_j(t_i)\}$。

例題：Machine Works | 凸包優化 DP

- 可用**直線** $y_i(t) = y_0 + g_i t$ 表示買了一台機器後的成長。
- 對於一個時間點 t_i 來說，我們需要知道的正是：$\max_{j<i}\{\$_{j\to i}\} = \max_{j<i}\{y_j(t_i)\}$。

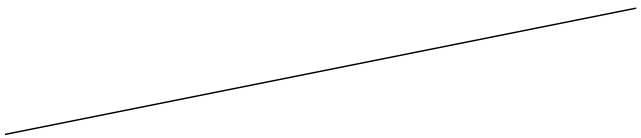


Figure: 包絡線

例題：Machine Works | 凸包優化 DP

- 可用**直線** $y_i(t) = y_0 + g_i t$ 表示買了一台機器後的成長。
- 對於一個時間點 t_i 來說，我們需要知道的正是：$\max_{j<i}\{\$_{j\to i}\} = max_{j<i}\{y_j(t_i)\}$。

Figure: 包絡線

例題：Machine Works | 凸包優化 DP

- 可用**直線** $y_i(t) = y_0 + g_i t$ 表示買了一台機器後的成長。
- 對於一個時間點 t_i 來說，我們需要知道的正是：$\max_{j<i}\{\$_{j\to i}\} = \max_{j<i}\{y_j(t_i)\}$。

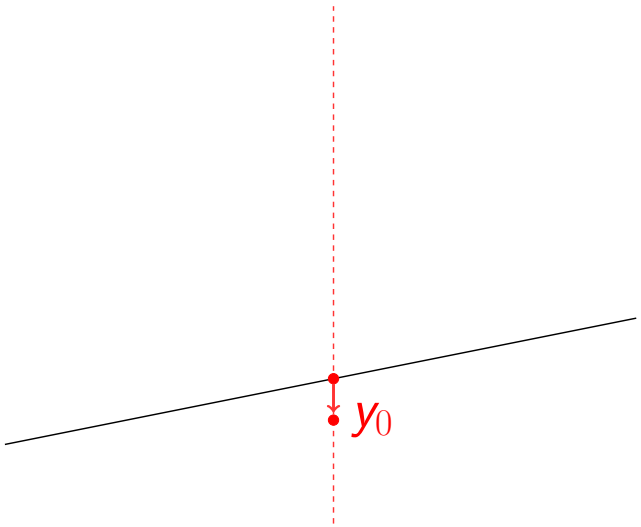


Figure: 包絡線

例題：Machine Works | 凸包優化 DP

- 可用**直線** $y_i(t) = y_0 + g_i t$ 表示買了一台機器後的成長。
- 對於一個時間點 t_i 來說，我們需要知道的正是：$\max_{j<i}\{\$_{j\to i}\} = max_{j<i}\{y_j(t_i)\}$。

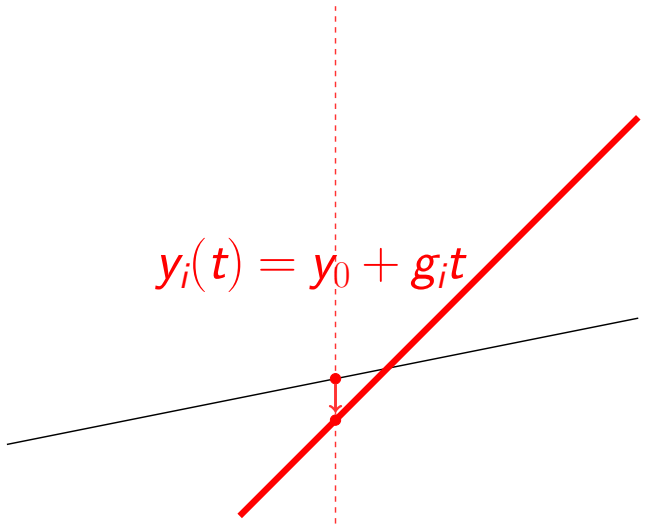


Figure: 包絡線

例題：Machine Works | 凸包優化 DP

- 可用**直線** $y_i(t) = y_0 + g_i t$ 表示買了一台機器後的成長。
- 對於一個時間點 t_i 來說，我們需要知道的正是：$\max_{j<i}\{\$_{j\to i}\} = \max_{j<i}\{y_j(t_i)\}$。

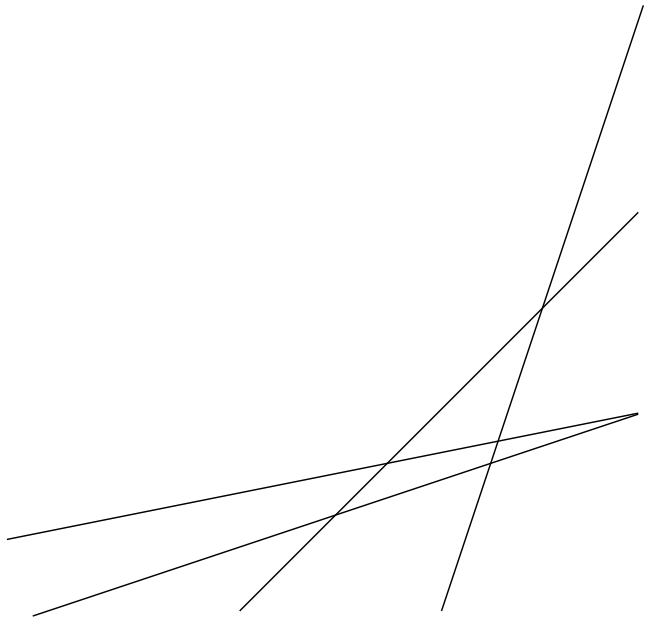


Figure: 包絡線

例題：Machine Works | 凸包優化 DP

- 可用**直線** $y_i(t) = y_0 + g_i t$ 表示買了一台機器後的成長。
- 對於一個時間點 t_i 來說，我們需要知道的正是：$\max_{j<i}\{\$_{j\to i}\} = max_{j<i}\{y_j(t_i)\}$。

Figure: 包絡線

例題：Machine Works | 凸包優化 DP

- 維護 piecewise-linear 包絡線 $\Gamma(t) = y$！
  $\rightarrow$ 使用 set 由左到右 (斜率由小到大) 維護線段。
- 查找特定時間 t 最佳：
  $\rightarrow$ 二分搜找到包含 t 之線段區間，並算 $y = \Gamma(t)$。
- 插入新線段並維護：
  $\rightarrow$ 二分搜找到斜率在 g_i 前後的線段，視情況維護。

例題：Machine Works | 凸包優化 DP

- 維護 piecewise-linear **包絡線** $\Gamma(t) = y$！
  $\rightarrow$ 使用 set 由左到右 (斜率由小到大) 維護線段。
- 查找特定時間 t 最佳：
  $\rightarrow$ 二分搜找到包含 t 之線段區間，並算 $y = \Gamma(t)$。
- 插入新線段並維護：
  $\rightarrow$ 二分搜找到斜率在 g_i 前後的線段，視情況維護。

例題：Machine Works | 凸包優化 DP

- 維護 piecewise-linear **包絡線** $\Gamma(t) = y$！
  $\rightarrow$ 使用 set 由左到右 (斜率由小到大) 維護線段。
- **查找**特定時間 t 最佳：
  $\rightarrow$ 二分搜找到包含 t 之線段區間，並算 $y = \Gamma(t)$。
- 插入新線段並維護：
  $\rightarrow$ 二分搜找到斜率在 g_i 前後的線段，視情況維護。

例題：Machine Works | 凸包優化 DP

- 維護 piecewise-linear **包絡線** $\Gamma(t) = y$！
  $\rightarrow$ 使用 set 由左到右 (斜率由小到大) 維護線段。
- **查找**特定時間 t 最佳：
  $\rightarrow$ 二分搜找到包含 t 之線段區間，並算 $y = \Gamma(t)$。
- **插入**新線段並維護：
  $\rightarrow$ 二分搜找到斜率在 g_i 前後的線段，視情況維護。

```
class Segment {
   public:
      long double m,c,x1,x2; // y=mx+c in [x1,x2]
      bool flag;
      long double evaly(long double x) const {
         return m*x+c;
      }
      const bool operator<(long double x) const {
         return x2-eps<x;
      }
      const bool operator<(const Segment &b) const {
         if(flag||b.flag) return *this<b.x1;
         return m+eps<b.m;
      }
};
```

```
1  void insert(Segment s) {
2     set<Segment>::iterator it=hull.find(s);
3     // check for same slope
4     if(it!=hull.end()) {
5        if(it->c+eps>=s.c) return;
6        hull.erase(it);
7     }
8     // check if below whole hull
9     it=hull.lower_bound(s);
10    if(it!=hull.end() &&
11       s.evaly(it->x1)<=it->evaly(it->x1)+eps) return;
```

```
12      // update right hull
13      while(it!=hull.end()) {
14          long double x=xintersection(s,*it);
15          if(x>=it->x2-eps) hull.erase(it++);
16          else {
17              s.x2=x;
18              it=replace(hull,it,Segment(it->m,it->c,x,it->
                    x2));
19              break;
20          }
21      }
```

```
22      // update left hull
23      while(it!=hull.begin()) {
24          long double x=xintersection(s,*(--it));
25          if(x<=it->x1+eps) hull.erase(it++);
26          else {
27              s.x1=x;
28              it=replace(hull,it,Segment(it->m,it->c,it->x1
                    ,x));
29              break;
30          }
31      }
32      // insert s
33      hull.insert(s);
34  }
```

例題：Machine Works | 凸包優化 DP

回顧一下我們做了什麼：

- 給一個 1D-1D 的 DP 問題，每個狀態對後續狀態的轉移可以用一條直線描述。
- 當前狀態的最佳轉移選擇是所有轉移直線座標最高者，即包絡線上此時此刻的座標。
- 可以用 set 維護包絡線。
- $O(\lg N)$ 查找、$O(\lg N)$ 插入。
  $\Rightarrow$ 整體時間複雜度為 $O(N \lg N)$！

例題：Machine Works | 凸包優化 DP

回顧一下我們做了什麼：

- 給一個 1D-1D 的 DP 問題，每個狀態對後續狀態的轉移**可以用一條直線描述**。
- 當前狀態的最佳轉移選擇是所有轉移直線座標最高者，即**包絡線上此時此刻的座標**。
- 可以用 set 維護包絡線。
- $O(\lg N)$ 查找、$O(\lg N)$ 插入。
  $\Rightarrow$ 整體時間複雜度為 $O(N \lg N)$！

例題：Machine Works | 凸包優化 DP

回顧一下我們做了什麼：

- 給一個 1D-1D 的 DP 問題，每個狀態對後續狀態的轉移**可以用一條直線描述**。
- 當前狀態的最佳轉移選擇是所有轉移直線座標最高者，即**包絡線上此時此刻的座標**。
- 可以用 set 維護包絡線。
- $O(\lg N)$ 查找、$O(\lg N)$ 插入。
  $\Rightarrow$ 整體時間複雜度為 $O(N \lg N)$！

例題：Machine Works | 凸包優化 DP

回顧一下我們做了什麼：

- 給一個 1D-1D 的 DP 問題，每個狀態對後續狀態的轉移可以用一條直線描述。
- 當前狀態的最佳轉移選擇是所有轉移直線座標最高者，即包絡線上此時此刻的座標。
- 可以用 set 維護包絡線。
- $O(\lg N)$ 查找、$O(\lg N)$ 插入。
  $\Rightarrow$ 整體時間複雜度為 $O(N \lg N)$！

例題：倉庫建設

一座山上從山頂到山腳共有 N 個銷貨點，由上至下分別位在遞增位置 $x_1, x_2, \ldots, x_N$。

有天下起了大雨，因此需要在某些銷貨點設置**臨時儲貨點**，並把貨物都運到儲貨點。為了節省時省力，商品只能**由山上往山下運**，亦即在銷貨點 i 的商品只能運到銷貨點 $j \mid j \geq i$。

已知位在銷貨點 i 的商品**每運一單位距離需要花費 c_i 元**，且**在 x_i 設置儲貨點的成本為 s_i**。問：若必須把所有貨物都運到儲貨點保存，至少要花多少錢？

$(1 \leq N \leq 10^5, 1 \leq x_i, c_i, s_i \leq 10^9)$

2D-1D 的凸單調

2D-1D 凸單調

2D-1D 的凸單調

Definition (2D-1D 問題)

- **目標函數**：$C[i][j],\ \forall\ 1 \leq i < j \leq n$。
- Input：
  - **轉移代價** $w(i,j),\ \forall\ 1 \leq i < j \leq n$
  - **初始化邊界** $C[i][i] = 0,\ \forall\ 1 \leq i \leq n$
- Output：

$$C[i][j] = w(i,j) + \min_{i \leq k < j} \{C[i][k] + C[k+1][j]\},\ \forall\ 1 \leq i < j \leq n$$

2D-1D 的凸單調

$$C[i][j] = w(i,j) + \min_{i \le k < j} \{C[i][k] + C[k+1][j]\},\ \forall\ 1 \le i < j \le n$$

- 有顯然的 $O(N^3)$ 作法：
  對於 $C[i][j]$，枚舉 $1 \le k < j$ 作為轉移分割點。
- 稱最佳選擇 k 為：
  $K_{i,j} = \operatorname{argmin}_{i \le k < j} \{C[i][k] + C[k+1][j]\}$。

2D-1D 的凸單調

$$C[i][j] = w(i,j) + \min_{i \le k < j} \{C[i][k] + C[k+1][j]\},\ \forall\ 1 \le i < j \le n$$

- **有顯然的 $O(N^3)$ 作法：**
  對於 $C[i][j]$，枚舉 $1 \le k < j$ 作為轉移分割點。
- **稱最佳選擇 k 為：**
  $K_{i,j} = \text{argmin}_{i \le k < j} \{C[i][k] + C[k+1][j]\}$。

2D-1D 的凸單調

$$C[i][j] = w(i,j) + \min_{i \le k < j} \{C[i][k] + C[k+1][j]\}, \ \forall\ 1 \le i < j \le n$$

- **有顯然的 $O(N^3)$ 作法：**
  對於 $C[i][j]$，枚舉 $1 \le k < j$ 作為轉移分割點。
- **稱最佳選擇 k 為：**
  $K_{i,j} = \text{argmin}_{i \le k < j} \{C[i][k] + C[k+1][j]\}$。

2D-1D 的凸單調

Theorem (Monge Condition ⇒ 最佳分割點滿足區間單調)

當 *2D/1D* 問題代價 C 滿足 *convex Monge condition*，亦即：

$$C[i][j] + C[i+1][j+1] \geq B[i][j+1] + B[i+1][j]$$

則**分割點具區間單調性**：

$$K_{i,j-1} \leq K_{i,j} \leq K_{i+1,j}$$

2D-1D 的凸單調

$$
\begin{array}{ccccc}
\cdots & & \cdots & & \cdots \\
K_{i-2,j-3} & \leq & K_{i-2,j-2} & \leq & K_{i-1,j-2} \\
K_{i-1,j-2} & \leq & K_{i-1,j-1} & \leq & K_{i,j-1} \\
K_{i,j-1} & \leq & K_{i,j} & \leq & K_{i+1,j} \\
K_{i+1,j} & \leq & K_{i+1,j+1} & \leq & K_{i+2,j+1} \\
K_{i+2,j+1} & \leq & K_{i+2,j+2} & \leq & K_{i+3,j+2} \\
\cdots & & \cdots & & \cdots
\end{array}
$$

2D-1D 的凸單調

$$
\begin{array}{rcccc}
\cdots & & & \cdots & \\
T_{i-2,j-3} & \leq & K_{i-1,j-2} & - & K_{i-2,j-3} \\
T_{i-1,j-2} & \leq & K_{i,j-1} & - & K_{i-1,j-2} \\
T_{i,j-1} & \leq & K_{i+1,j} & - & K_{i,j-1} \\
T_{i+1,j} & \leq & K_{i+2,j+1} & - & K_{i+1,j} \\
T_{i+2,j+1} & \leq & K_{i+3,j+2} & - & K_{i+2,j+1} \\
\cdots & & & \cdots &
\end{array}
$$

2D-1D 的凸單調

$$
\begin{array}{rcccc}
\cdots & & & \cdots & \\
T_{i-2,j-3} & \leq & K_{i-1,j-2} & - & K_{i-2,j-3} \\
T_{i-1,j-2} & \leq & K_{i,j-1} & - & K_{i-1,j-2} \\
T_{i,j-1} & \leq & K_{i+1,j} & - & K_{i,j-1} \\
T_{i+1,j} & \leq & K_{i+2,j+1} & - & K_{i+1,j} \\
T_{i+2,j+1} & \leq & K_{i+3,j+2} & - & K_{i+2,j+1} \\
\cdots & & & \cdots &
\end{array}
$$

2D-1D 的凸單調

$$
\begin{array}{rcccc}
\cdots & & & \cdots & \\
T_{i-2,j-3} & \leq & K_{i-1,j-2} & - & K_{i-2,j-3} \\
T_{i-1,j-2} & \leq & K_{i,j-1} & - & K_{i-1,j-2} \\
T_{i,j-1} & \leq & K_{i+1,j} & - & K_{i,j-1} \\
T_{i+1,j} & \leq & K_{i+2,j+1} & - & K_{i+1,j} \\
T_{i+2,j+1} & \leq & K_{i+3,j+2} & - & K_{i+2,j+1} \\
\cdots & & & \cdots &
\end{array}
$$

2D-1D 的凸單調

$$
\begin{array}{rcccc}
\cdots & & & \cdots & \\
T_{i-2,j-3} & \leq & K_{i-1,j-2} & - & K_{i-2,j-3} \\
T_{i-1,j-2} & \leq & K_{i,j-1} & - & K_{i-1,j-2} \\
T_{i,j-1} & \leq & K_{i+1,j} & - & K_{i,j-1} \\
T_{i+1,j} & \leq & K_{i+2,j+1} & - & K_{i+1,j} \\
T_{i+2,j+1} & \leq & K_{i+3,j+2} & - & K_{i+2,j+1} \\
\cdots & & & \cdots &
\end{array}
$$

2D-1D 的凸單調

$$
\begin{array}{rcccc}
\cdots & & & \cdots & \\
T_{i-2,j-3} & \leq & K_{i-1,j-2} & - & K_{i-2,j-3} \\
T_{i-1,j-2} & \leq & K_{i,j-1} & - & K_{i-1,j-2} \\
T_{i,j-1} & \leq & K_{i+1,j} & - & K_{i,j-1} \\
T_{i+1,j} & \leq & K_{i+2,j+1} & - & K_{i+1,j} \\
T_{i+2,j+1} & \leq & K_{i+3,j+2} & - & K_{i+2,j+1} \\
\cdots & & & \cdots &
\end{array}
$$

2D-1D 的凸單調

$$
\begin{array}{rcccc}
\cdots & & & \cdots & \\
T_{i-2,j-3} & \leq & K_{i-1,j-2} & - & K_{i-2,j-3} \\
T_{i-1,j-2} & \leq & K_{i,j-1} & - & K_{i-1,j-2} \\
T_{i,j-1} & \leq & K_{i+1,j} & - & K_{i,j-1} \\
T_{i+1,j} & \leq & K_{i+2,j+1} & - & K_{i+1,j} \\
T_{i+2,j+1} & \leq & K_{i+3,j+2} & - & K_{i+2,j+1} \\
\cdots & & & \cdots &
\end{array}
$$

2D-1D 的凸單調

$$
\begin{array}{rcccc}
\cdots & & & \cdots & \\
T_{i-2,j-3} & \le & K_{i-1,j-2} & - & K_{i-2,j-3} \\
T_{i-1,j-2} & \le & K_{i,j-1} & - & K_{i-1,j-2} \\
T_{i,j-1} & \le & K_{i+1,j} & - & K_{i,j-1} \\
T_{i+1,j} & \le & K_{i+2,j+1} & - & K_{i+1,j} \\
T_{i+2,j+1} & \le & K_{i+3,j+2} & - & K_{i+2,j+1} \\
\cdots & & & \cdots &
\end{array}
$$

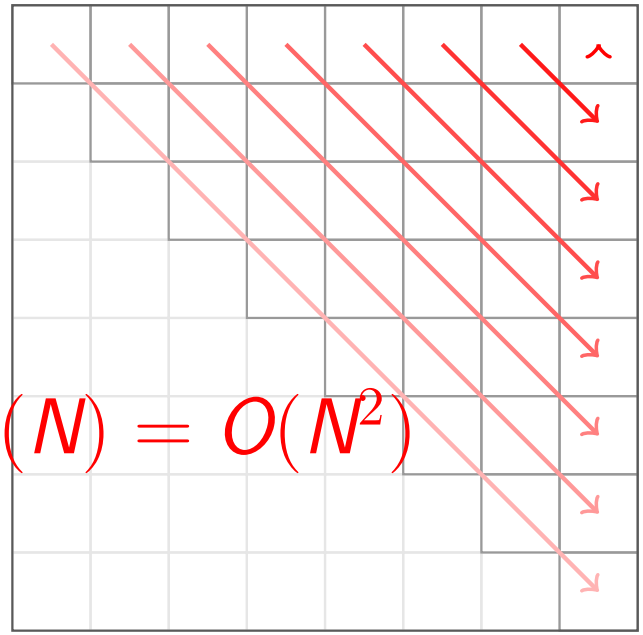
$$\sum_{j-i=C} T_{i,j} \le K_{*,*} - K_{*,*}$$

2D-1D 的凸單調

$$
\begin{array}{rcccc}
\cdots & & & \cdots & \\
T_{i-2,j-3} & \le & K_{i-1,j-2} & - & K_{i-2,j-3} \\
T_{i-1,j-2} & \le & K_{i,j-1} & - & K_{i-1,j-2} \\
T_{i,j-1} & \le & K_{i+1,j} & - & K_{i,j-1} \\
T_{i+1,j} & \le & K_{i+2,j+1} & - & K_{i+1,j} \\
T_{i+2,j+1} & \le & K_{i+3,j+2} & - & K_{i+2,j+1} \\
\cdots & & & \cdots &
\end{array}
$$

$$
\begin{aligned}
\sum_{j-i=C} T_{i,j} &\le K_{*,*} - K_{*,*} \\
&\le N
\end{aligned}
$$

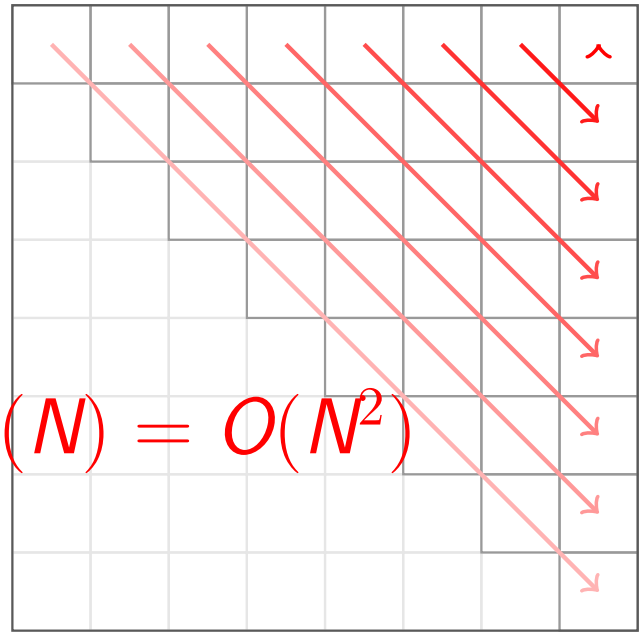
2D-1D 的凸單調



$$O(N) \times O(N) = O(N^2)$$

Figure: 2D-1D 凸單調 DP

2D-1D 的凸單調



$$O(N) \times O(N) = O(N^2)$$

Figure: 2D-1D 凸單調 DP

例題：Optimal BST

你要建立一顆二元搜索樹 (binary search tree) 儲存 $1, 2, 3, \ldots, N$ 等共 N 個數 (在葉節點上)。

若你已經知道數字 i 被查詢了 t_i 次，而每次查詢所花的時間正是該數字所在的葉節點的深度。由於一棵 N 個葉節點的二元搜索樹可以有非常多種不同長相，你想利用這些節點被查詢次數的資訊建出一顆二元搜索樹，使得查詢花的總時間盡可能的少。

試問在最佳的二元搜索樹結構下，查詢總時間最少是多少？

$(1 \leq N \leq 2000)$

例題：Optimal BST

- 二元樹 = 根 + 左子樹 + 右子樹。
- $DP[i][j] = i$ 到 j 若建成 BST 之最小查詢時間。
- $$DP[i][j] = \max_{1 \le k < j} \{DP[i][k] + DP[k+1][j] + w(i,j)\}$$
$$w(i,j) = \sum_{1 \le k \le j} q[k]$$
- 分割點具區間單調性！

例題：Optimal BST

- 二元樹 = 根 + 左子樹 + 右子樹。
- $DP[i][j] = i$ 到 j 若建成 BST 之最小查詢時間。
- $$DP[i][j] = \max_{1 \le k < j} \{DP[i][k] + DP[k+1][j] + w(i,j)\}$$
$$w(i,j) = \sum_{1 \le k \le j} q[k]$$
- 分割點具區間單調性！

例題：Optimal BST

- 二元樹 $=$ 根 $+$ 左子樹 $+$ 右子樹。
- $DP[i][j] = i$ 到 j 若建成 BST 之最小查詢時間。
- $$DP[i][j] = \max_{1 \le k < j} \{DP[i][k] + DP[k+1][j] + w(i,j)\}$$
$$w(i,j) = \sum_{1 \le k \le j} q[k]$$
- 分割點具區間單調性！

例題：Optimal BST

- 二元樹 $=$ 根 $+$ 左子樹 $+$ 右子樹。
- $DP[i][j] = i$ 到 j 若建成 BST 之最小查詢時間。
-

$$DP[i][j] = \max_{1 \le k < j} \{DP[i][k] + DP[k+1][j] + w(i,j)\}$$

$$w(i,j) = \sum_{1 \le k \le j} q[k]$$

- 分割點具區間單調性！

例題：Optimal BST

- **二元樹** $=$ **根** $+$ **左子樹** $+$ **右子樹**。
- $DP[i][j] = i$ **到** j **若建成** BST **之最小查詢時間**。
-

$$DP[i][j] = \max_{1 \le k < j} \{DP[i][k] + DP[k+1][j] + w(i,j)\}$$

$$w(i,j) = \sum_{1 \le k \le j} q[k]$$

- **分割點具區間單調性！**

Space Compression

Chapter III

SPACE COMPRESSION

例題：Longest Common Subsequence

對於一個字串 S，定義其**子序列** (subsequence) 為

$$(S_{i1}, S_{i_2}, \ldots, S_{i_l}), \text{ for some } 0 \leq i_1 < i_2 < \cdots < i_l < |S|,$$

亦即選擇性刪去其中一些字元後照原順序拼回的字串。

若一字串 S 同時是 A 與 B 的 subsequence，則說 S 是 A 與 B 的一個**共同子序列** (common subsequence)。

今給定兩字串 A, B，求其**最長共同子序列** (LCS)。

$(1 \leq |A|, |B| \leq 2000)$

例題：Longest Common Subsequence

- $DP[i][j] :=$ 前綴 $A[0:i]$ 與前綴 $B[0:j]$ 的 LCS 是多長。
- $DP[i][j] = \begin{cases} DP[i-1][j-1]+1 & \dots\ A[i]=B[j] \\ \max(DP[i][j-1], DP[i-1][j]) & \dots\ \text{otherwise} \end{cases}$
- 那怎麼復原真正的 LCS 字串 呢？
- $pred[i][j] = (i', j')$。

例題：Longest Common Subsequence

- $DP[i][j] :=$ **前綴** $A[0:i]$ **與前綴** $B[0:j]$ **的** LCS **是多長。**
- $DP[i][j] = \begin{cases} DP[i-1][j-1]+1 & \dots\ A[i]=B[j] \\ \max(DP[i][j-1], DP[i-1][j]) & \dots\ \text{otherwise} \end{cases}$
- **那怎麼復原真正的** LCS **字串 呢？**
- $pred[i][j] = (i', j')$。

例題：Longest Common Subsequence

- $DP[i][j] :=$ **前綴** $A[0:i]$ **與前綴** $B[0:j]$ **的** LCS **是多長。**
- $DP[i][j] = \begin{cases} DP[i-1][j-1]+1 & \dots\ A[i]=B[j] \\ \max(DP[i][j-1], DP[i-1][j]) & \dots\ \text{otherwise} \end{cases}$
- **那怎麼復原真正的** LCS **字串 呢？**
- $pred[i][j] = (i', j')$。

例題：Longest Common Subsequence

- $DP[i][j] :=$ **前綴** $A[0:i]$ **與前綴** $B[0:j]$ **的** LCS **是多長。**
- $DP[i][j] = \begin{cases} DP[i-1][j-1]+1 & \dots\ A[i]=B[j] \\ \max(DP[i][j-1], DP[i-1][j]) & \dots\ \text{otherwise} \end{cases}$
- **那怎麼復原真正的** LCS **字串 呢？**
- $pred[i][j] = (i', j')$。

例題：Longest Common Subsequence

- $DP[i][j] :=$ **前綴** $A[0:i]$ **與前綴** $B[0:j]$ **的** LCS **是多長。**
- $$DP[i][j] = \begin{cases} DP[i-1][j-1]+1 & \dots\ A[i]=B[j] \\ \max(DP[i][j-1], DP[i-1][j]) & \dots\ \text{otherwise} \end{cases}$$
- **那怎麼復原真正的** LCS **字串 呢？**
- $pred[i][j] = (i', j')$。

例題：Longest Common Subsequence

- 時間複雜度：$O(|A| \cdot |B|)$
- 空間複雜度：$O(|A| \cdot |B|)$
- 更新只跟前一行 $(i-1)$ 相關，不用知道再之前的訊息。
- 滾動數組：$DP[i][j] \to DP[i \bmod 2][j]$。 空間複雜度：$O(|B|)$。
- 路徑怎麼辦

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- 空間複雜度：$O(|A| \cdot |B|)$
- 更新只跟前一行 $(i-1)$ 相關，不用知道再之前的訊息。
- 滾動數組：$DP[i][j] \to DP[i \bmod 2][j]$。空間複雜度：$O(|B|)$。
- 路徑怎麼辦

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$
- 更新只跟前一行 $(i-1)$ 相關，不用知道再之前的訊息。
- 滾動數組：$DP[i][j] \to DP[i \bmod 2][j]$。 空間複雜度：$O(|B|)$。
- 路徑怎麼辦

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$ ...?
- **更新只跟前一行** $(i-1)$ **相關，不用知道再之前的訊息。**
- **滾動數組：** $DP[i][j] \to DP[i \bmod 2][j]$。 **空間複雜度：** $O(|B|)$。
- **路徑怎麼辦**

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$ **...?**
- **更新只跟前一行** $(i-1)$ **相關，不用知道再之前的訊息。**
- **滾動數組：** $DP[i][j] \to DP[i \bmod 2][j]$**。 空間複雜度：** $O(|B|)$**。**
- **路徑怎麼辦**

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$ **...?**
- **更新只跟前一行** $(i-1)$ **相關，不用知道再之前的訊息。**
- **滾動數組：** $DP[i][j] \to DP[i \bmod 2][j]$**。 空間複雜度：** $O(|B|)$**。**
- **路徑怎麼辦**

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$ **...?**
- **更新只跟前一行** $(i-1)$ **相關，不用知道再之前的訊息。**
- **滾動數組：** $DP[i][j] \to DP[i \bmod 2][j]$**。 空間複雜度：** $O(|B|)$**。**
- **路徑怎麼辦辦**

例題：Longest Common Subsequence

- 時間複雜度：$O(|A| \cdot |B|)$
- 空間複雜度：$O(|A| \cdot |B|)$ ...?
- 更新只跟前一行 $(i-1)$ 相關，不用知道再之前的訊息。
- 滾動數組：$DP[i][j] \to DP[i \bmod 2][j]$。 空間複雜度：$O(|B|)$。
- 路徑怎麼辦辦辦

例題：Longest Common Subsequence

- 時間複雜度：$O(|A| \cdot |B|)$
- 空間複雜度：$O(|A| \cdot |B|)$ ...?
- 更新只跟前一行 $(i-1)$ 相關，不用知道再之前的訊息。
- 滾動數組：$DP[i][j] \to DP[i \bmod 2][j]$。 空間複雜度：$O(|B|)$。
- 路徑怎麼辦辦辦辦

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$ **...?**
- **更新只跟前一行** $(i-1)$ **相關，不用知道再之前的訊息。**
- **滾動數組：** $DP[i][j] \to DP[i \bmod 2][j]$**。 空間複雜度：** $O(|B|)$**。**
- **路徑怎麼辦辦辦辦辦辦**

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$ **...?**
- **更新只跟前一行** $(i-1)$ **相關，不用知道再之前的訊息。**
- **滾動數組：** $DP[i][j] \to DP[i \bmod 2][j]$**。 空間複雜度：** $O(|B|)$**。**
- **路徑怎麼辦辦辦辦辦辦辦**

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$ **...?**
- **更新只跟前一行** $(i-1)$ **相關，不用知道再之前的訊息。**
- **滾動數組：** $DP[i][j] \to DP[i \bmod 2][j]$**。 空間複雜度：** $O(|B|)$**。**
- **路徑怎麼辦辦辦辦辦辦辦辦**

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$ **...?**
- **更新只跟前一行** $(i-1)$ **相關，不用知道再之前的訊息。**
- **滾動數組：** $DP[i][j] \to DP[i \bmod 2][j]$**。 空間複雜度：** $O(|B|)$**。**
- **路徑怎麼辦辦辦辦辦辦辦辦辦辦**

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$ **...?**
- **更新只跟前一行** $(i-1)$ **相關，不用知道再之前的訊息。**
- **滾動數組：** $DP[i][j] \to DP[i \bmod 2][j]$**。 空間複雜度：** $O(|B|)$**。**
- **路徑怎麼辦辦辦辦辦辦辦辦辦辦辦辦**

例題：Longest Common Subsequence

- **時間複雜度：** $O(|A| \cdot |B|)$
- **空間複雜度：** $O(|A| \cdot |B|)$ **...?**
- **更新只跟前一行** $(i-1)$ **相關，不用知道再之前的訊息。**
- **滾動數組：** $DP[i][j] \to DP[i \bmod 2][j]$**。 空間複雜度：** $O(|B|)$**。**
- **路徑怎麼辦辦辦辦辦辦辦辦辦辦辦辦辦辦**

例題：Longest Common Subsequence | 線性空間作法

- 要線性空間得到路徑，對於過程我們幾乎沒有「容錯的記憶量」。
- 關鍵就成為：一旦紀錄就是紀錄正確資訊 (對的路徑)。
- 有可能嗎?!

例題：Longest Common Subsequence | 線性空間作法

- 要線性空間得到路徑，對於過程我們幾乎沒有「容錯的記憶量」。
- 關鍵就成為：一旦紀錄就是紀錄正確資訊 (對的路徑)。
- 有可能嗎?!

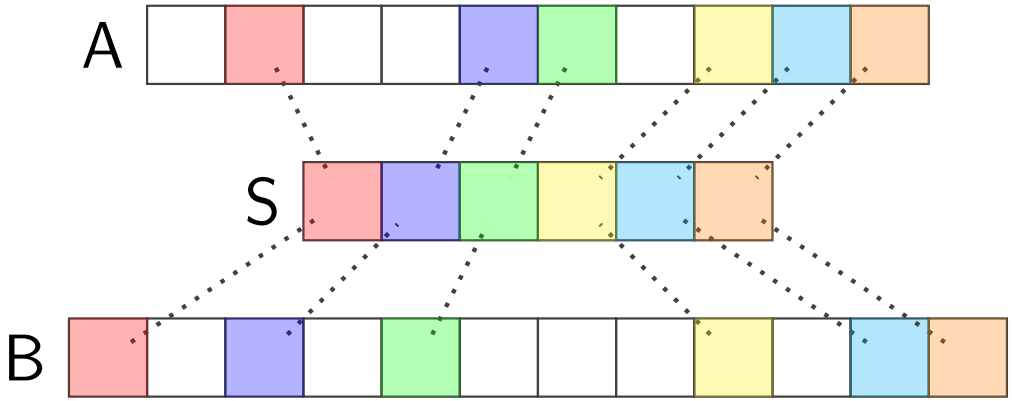
例題：Longest Common Subsequence | 線性空間作法

- 要線性空間得到路徑，對於過程我們幾乎沒有「容錯的記憶量」。
- 關鍵就成為：一旦紀錄就是紀錄正確資訊 (對的路徑)。
- 有可能嗎?!

例題：Longest Common Subsequence | 線性空間作法

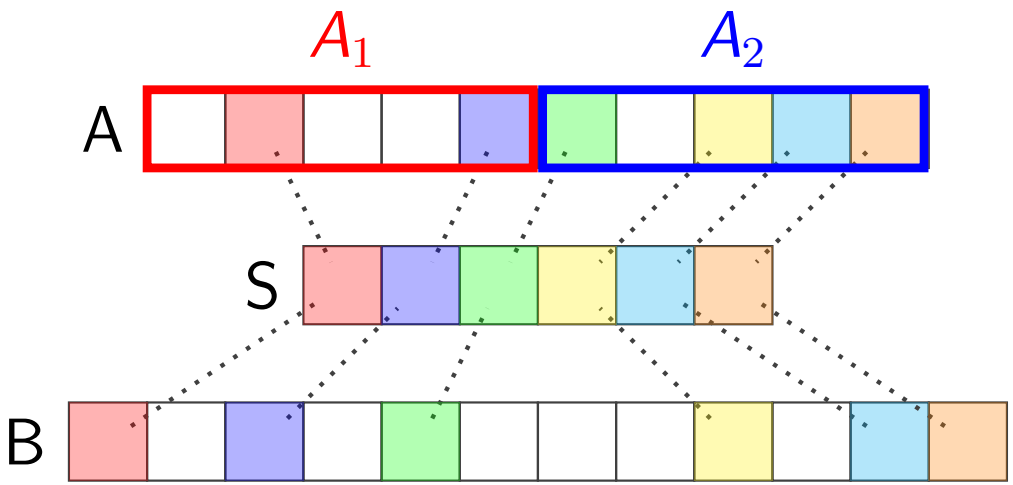
- 要線性空間得到路徑，對於過程我們幾乎沒有「容錯的記憶量」。
- 關鍵就成為：一旦紀錄就是紀錄正確資訊 (對的路徑)。
- 有可能嗎?!

例題：Longest Common Subsequence | 線性空間作法



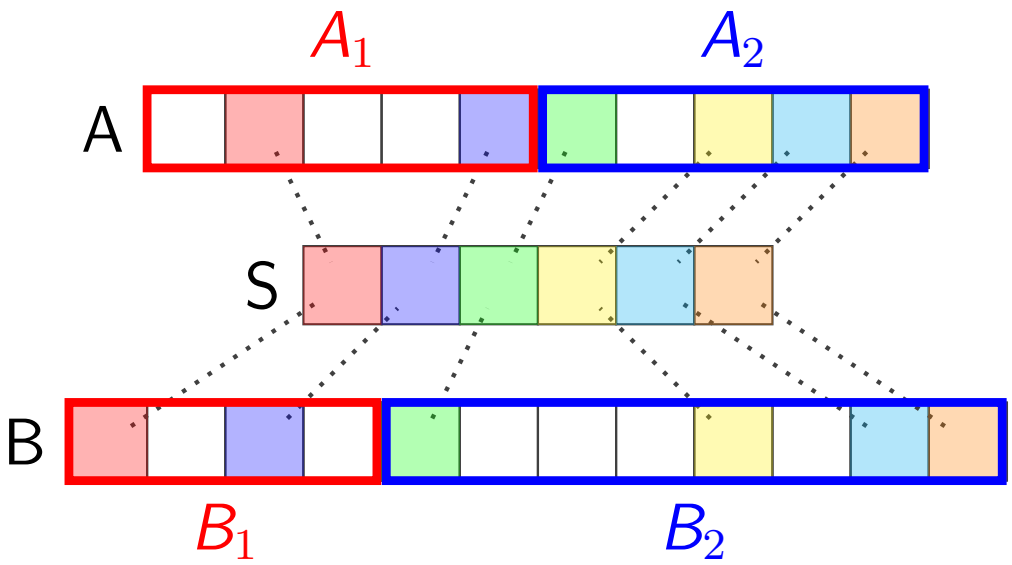
- $LCS(A, B) = \max_t \{LCS(A_1, B_1) + LCS(A_2, B_2)\}$, 其中 $0 < t \leq |B|,\ B_1 = B[0 \ldots t-1],\ B_2 = B[t \ldots |B|-1]$
- 找出最佳的 B 的分割方式，並遞迴下去做 $(A_1, B_1), (A_2, B_2)$!

例題：Longest Common Subsequence | 線性空間作法



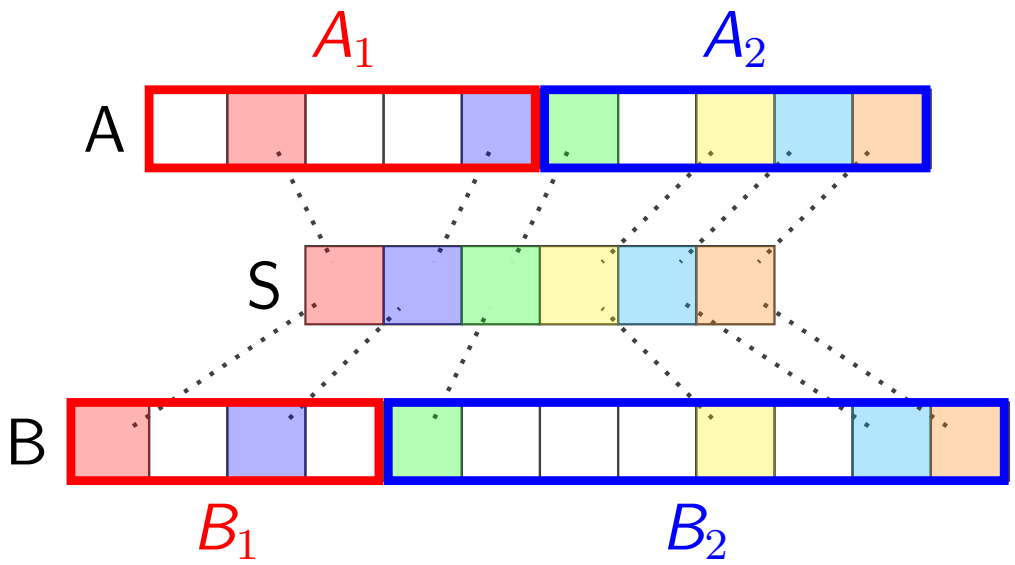
- $LCS(A, B) = \max_t \{LCS(A_1, B_1) + LCS(A_2, B_2)\}$, 其中 $0 < t \leq |B|,\ B_1 = B[0 \ldots t-1],\ B_2 = B[t \ldots |B|-1]$
- 找出最佳的 B 的分割方式，並遞迴下去做 $(A_1, B_1), (A_2, B_2)$!

例題：Longest Common Subsequence | 線性空間作法



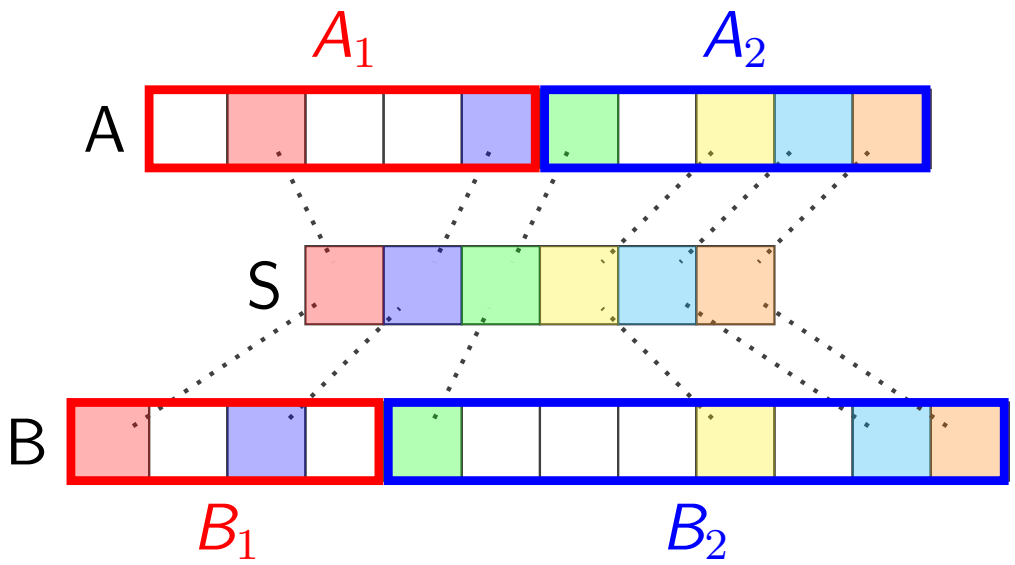
- $LCS(A, B) = \max_t \{LCS(A_1, B_1) + LCS(A_2, B_2)\}$, 其中 $0 < t \leq |B|,\ B_1 = B[0 \ldots t-1],\ B_2 = B[t \ldots |B|-1]$
- 找出最佳的 B 的分割方式，並遞迴下去做 $(A_1, B_1), (A_2, B_2)$!

例題：Longest Common Subsequence | 線性空間作法



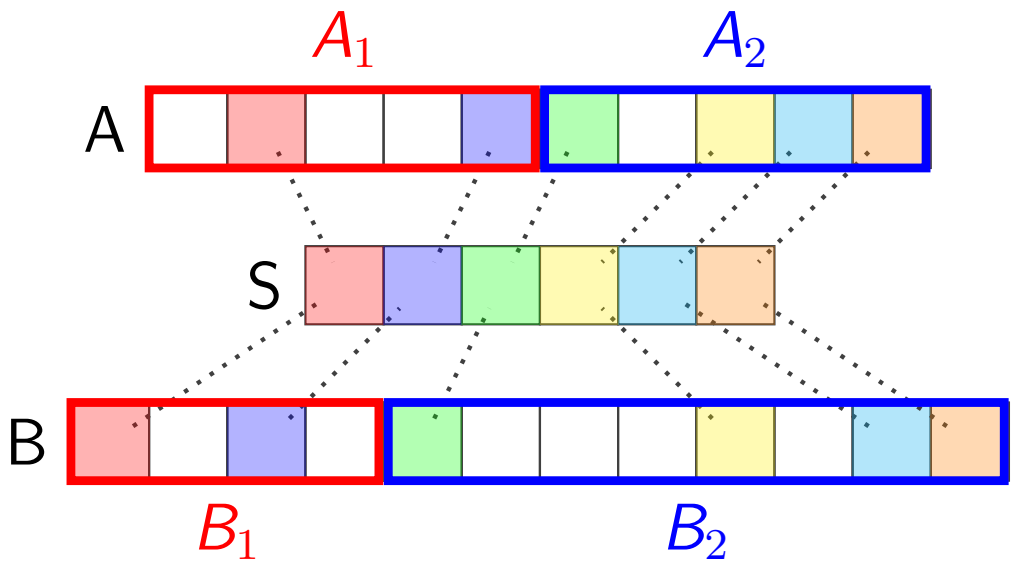
- $LCS(A, B) = \max_t \{LCS(A_1, B_1) + LCS(A_2, B_2)\}$, **其中** $0 < t \leq |B|,\ B_1 = B[0 \ldots t-1],\ B_2 = B[t \ldots |B|-1]$
- **找出最佳的 B 的分割方式，並遞迴下去做 $(A_1, B_1), (A_2, B_2)$!**

例題：Longest Common Subsequence | 線性空間作法



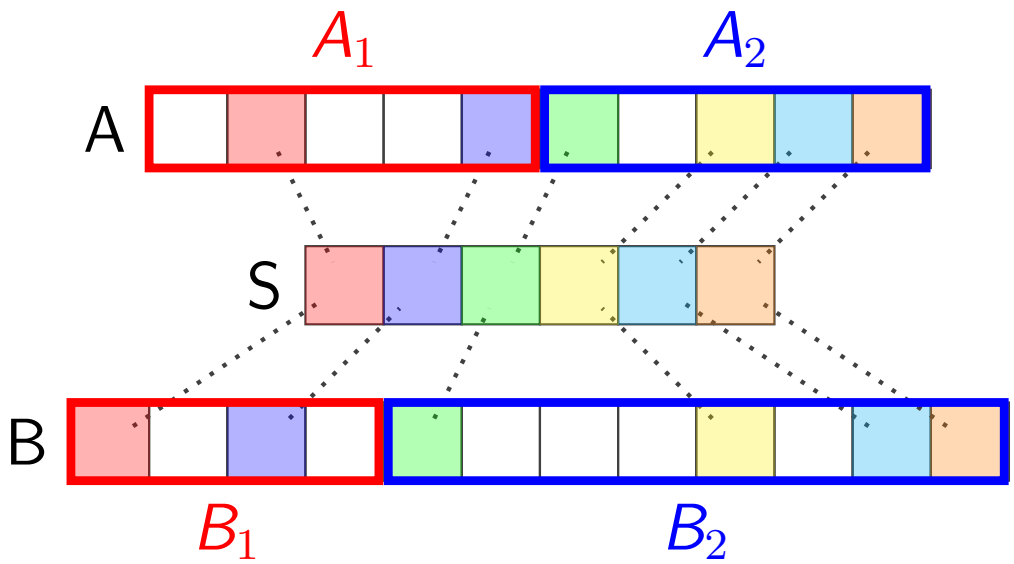
- $LCS(A, B) = \max_t \{LCS(A_1, B_1) + LCS(A_2, B_2)\}$, **其中** $0 < t \leq |B|,\ B_1 = B[0 \ldots t-1],\ B_2 = B[t \ldots |B|-1]$
- **找出最佳的 B 的分割方式，並遞迴下去做** $(A_1, B_1), (A_2, B_2)$ **！**

例題：Longest Common Subsequence | 線性空間作法



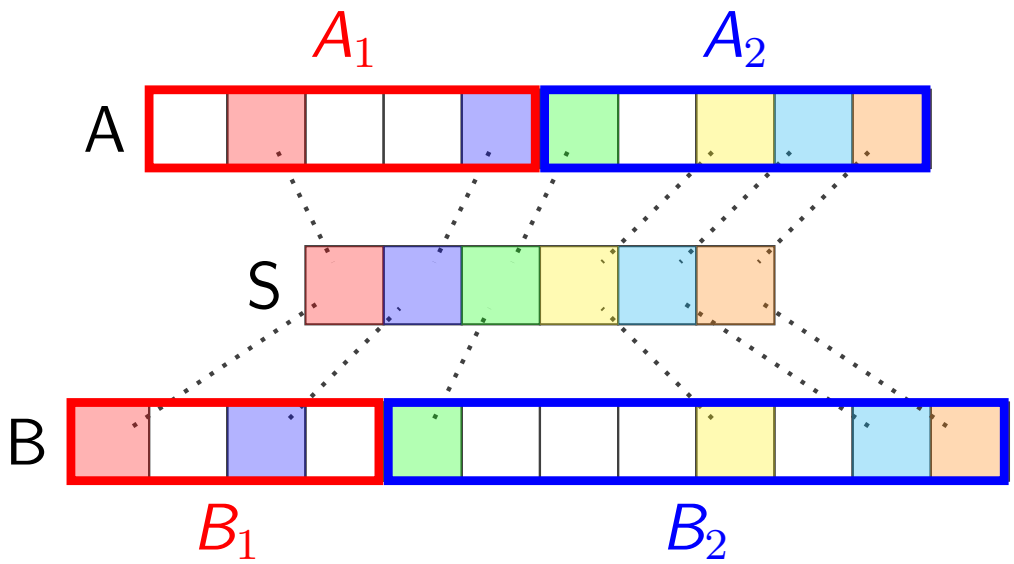
- 如何找出最佳分割 B_1, B_2 ？
- 做 $LCS(A_1, B)$, $LCS(A_2', B')$。
  其中 “ S' ” 表反轉的字串 S。
- 可線性時間決定 $\arg\max_t \{LCS(A_1, B_1) + LCS(A_2, B_2)\}$

例題：Longest Common Subsequence | 線性空間作法



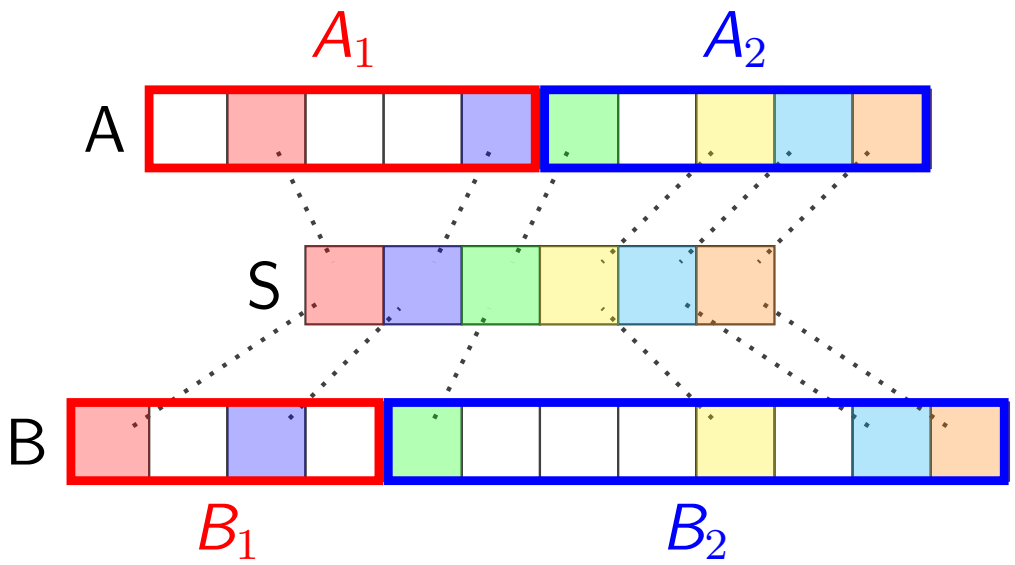
- 如何找出最佳分割 B_1, B_2 ？
- 做 $LCS(A_1, B)$, $LCS(A_2', B')$。
  其中 " S' " 表反轉的字串 S。
- 可線性時間決定 $\arg\max_t \{LCS(A_1, B_1) + LCS(A_2, B_2)\}$

例題：Longest Common Subsequence | 線性空間作法



- 如何找出最佳分割 B_1, B_2 ？
- 做 $LCS(A_1, B)$, $LCS(A_2', B')$。
  其中 “ S' ” 表反轉的字串 S。
- 可線性時間決定 $\arg\max_t \{LCS(A_1, B_1) + LCS(A_2, B_2)\}$

例題：Longest Common Subsequence | 線性空間作法



- 如何找出最佳分割 B_1, B_2 ？
- 做 $LCS(A_1, B)$, $LCS(A_2', B')$。
  其中 “ S' ” 表反轉的字串 S。
- 可線性時間決定 $\arg\max_t \{LCS(A_1, B_1) + LCS(A_2, B_2)\}$

例題：Longest Common Subsequence | 線性空間作法

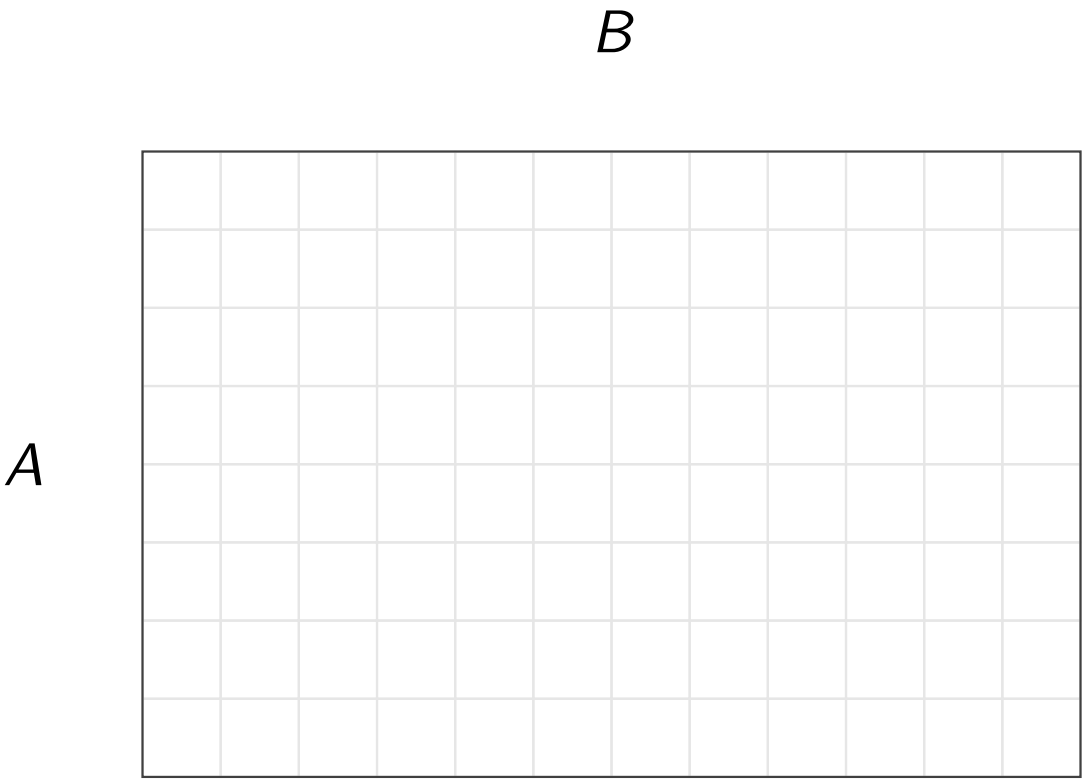


Figure: 分治法求 DP 路徑

例題：Longest Common Subsequence | 線性空間作法

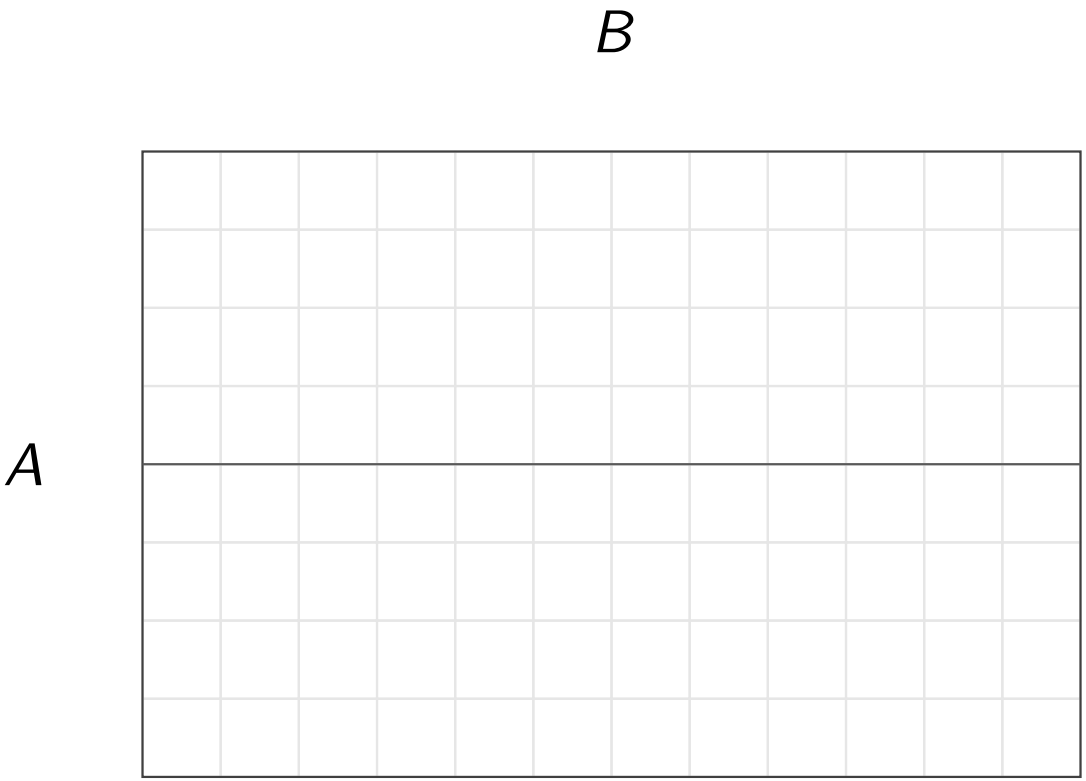


Figure: 分治法求 DP 路徑

例題：Longest Common Subsequence | 線性空間作法

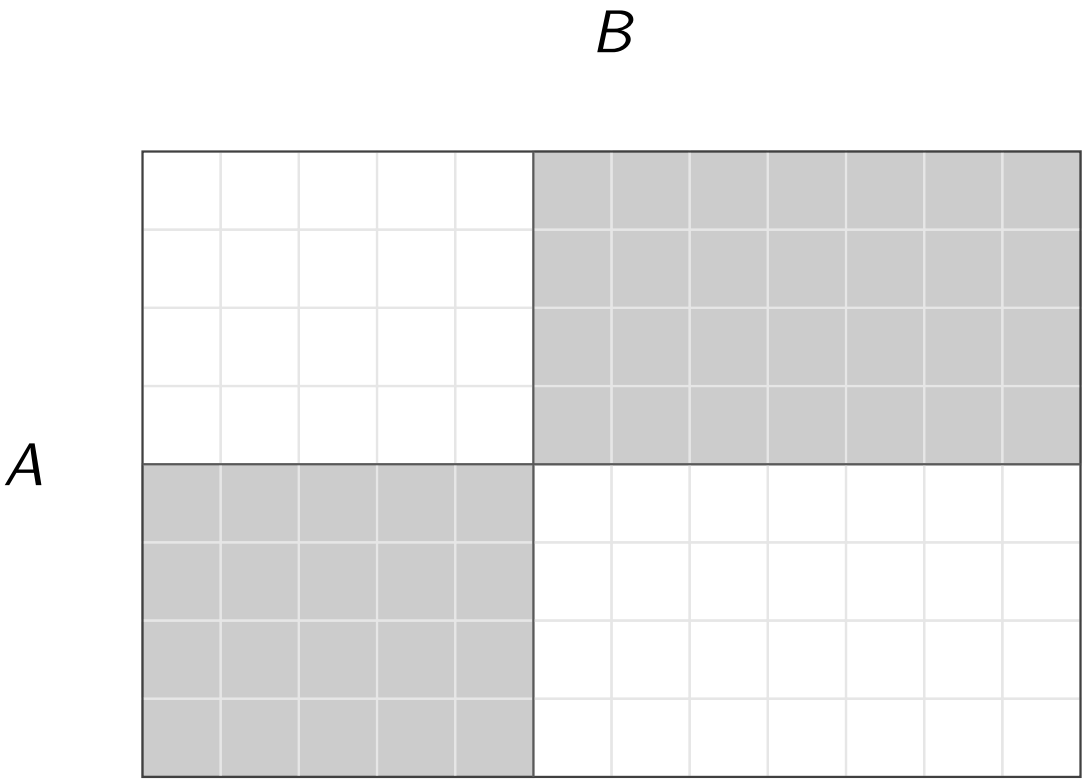


Figure: 分治法求 DP 路徑

例題：Longest Common Subsequence | 線性空間作法

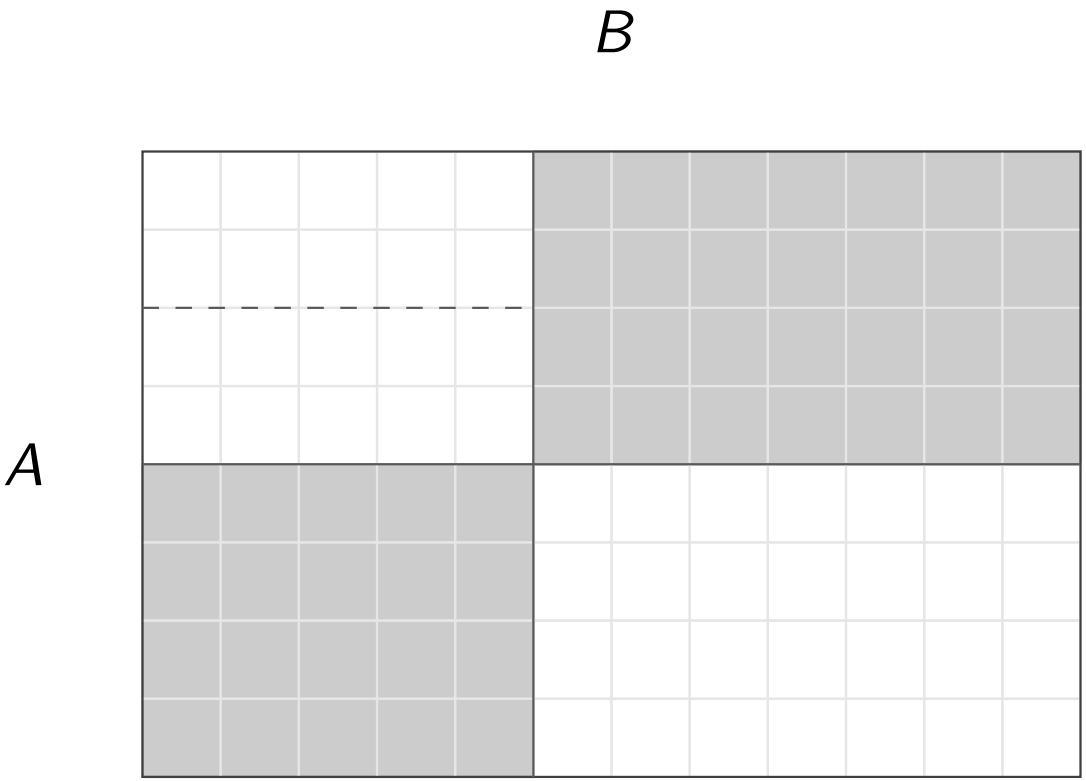


Figure: 分治法求 DP 路徑

例題：Longest Common Subsequence | 線性空間作法

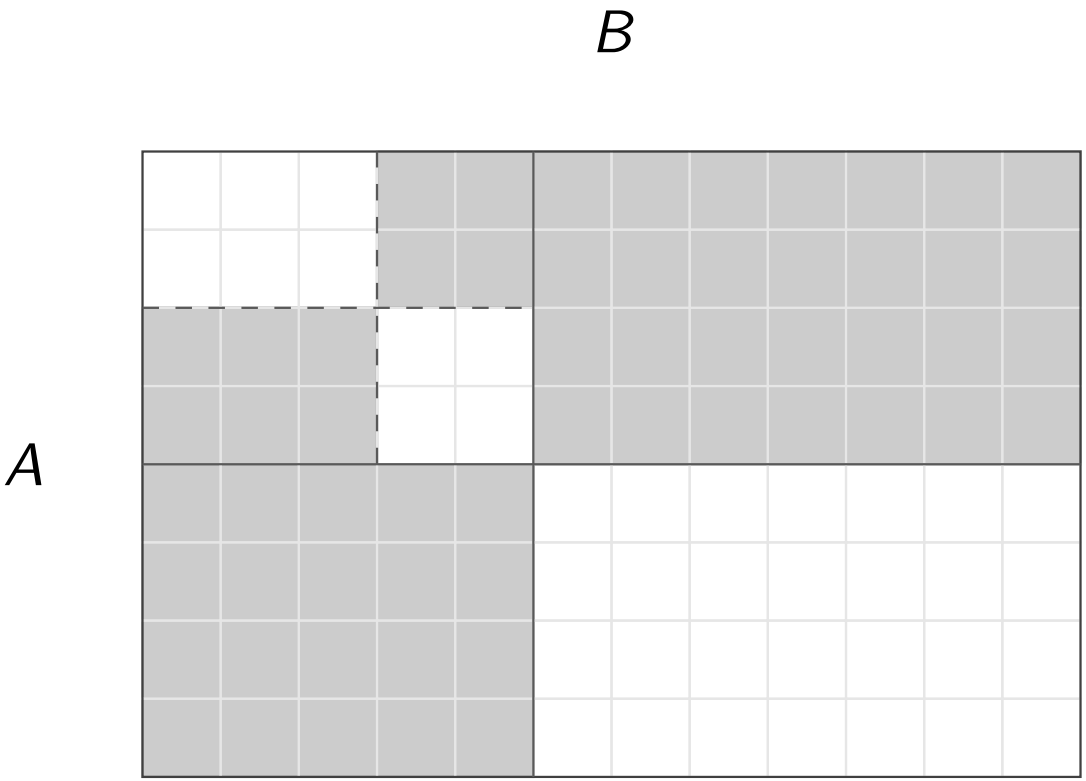


Figure: 分治法求 DP 路徑

例題：Longest Common Subsequence | 線性空間作法

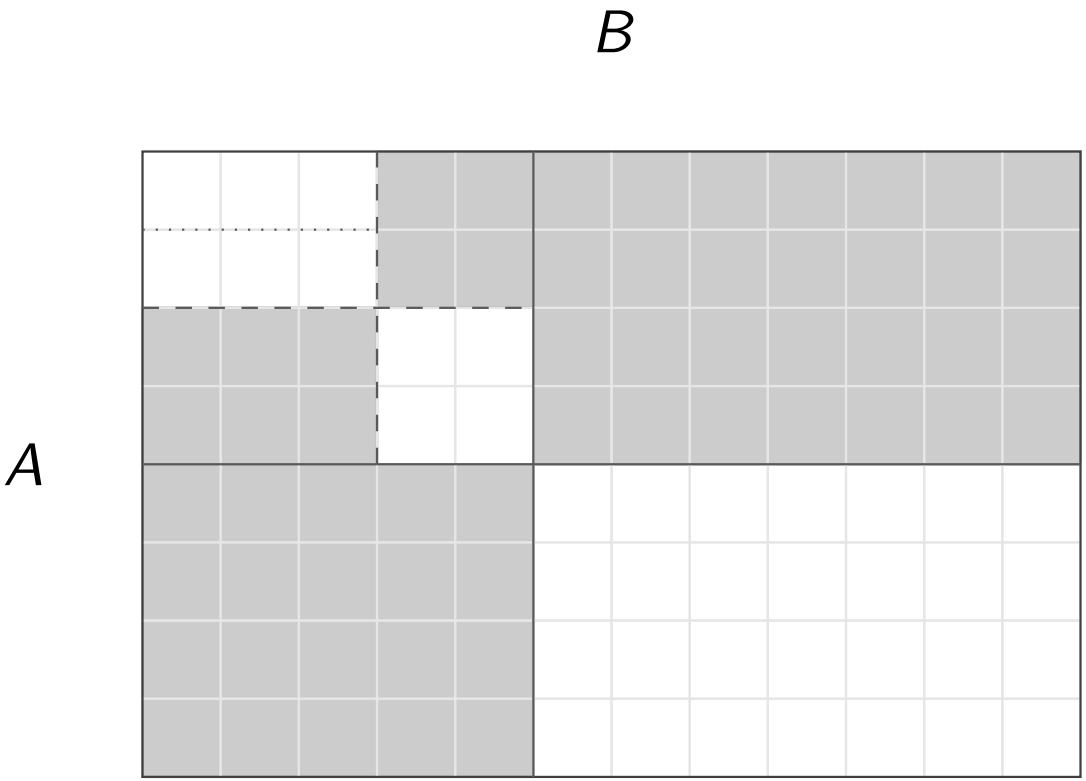


Figure: 分治法求 DP 路徑

例題：Longest Common Subsequence | 線性空間作法

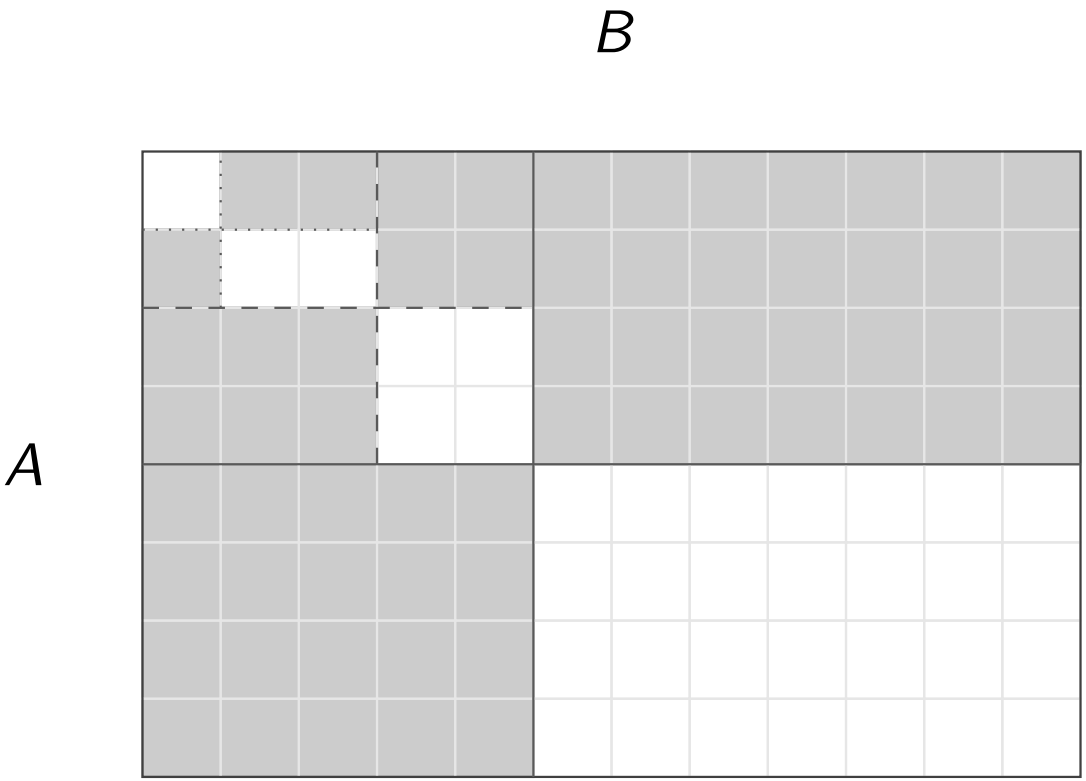


Figure: 分治法求 DP 路徑

例題：Longest Common Subsequence | 線性空間作法

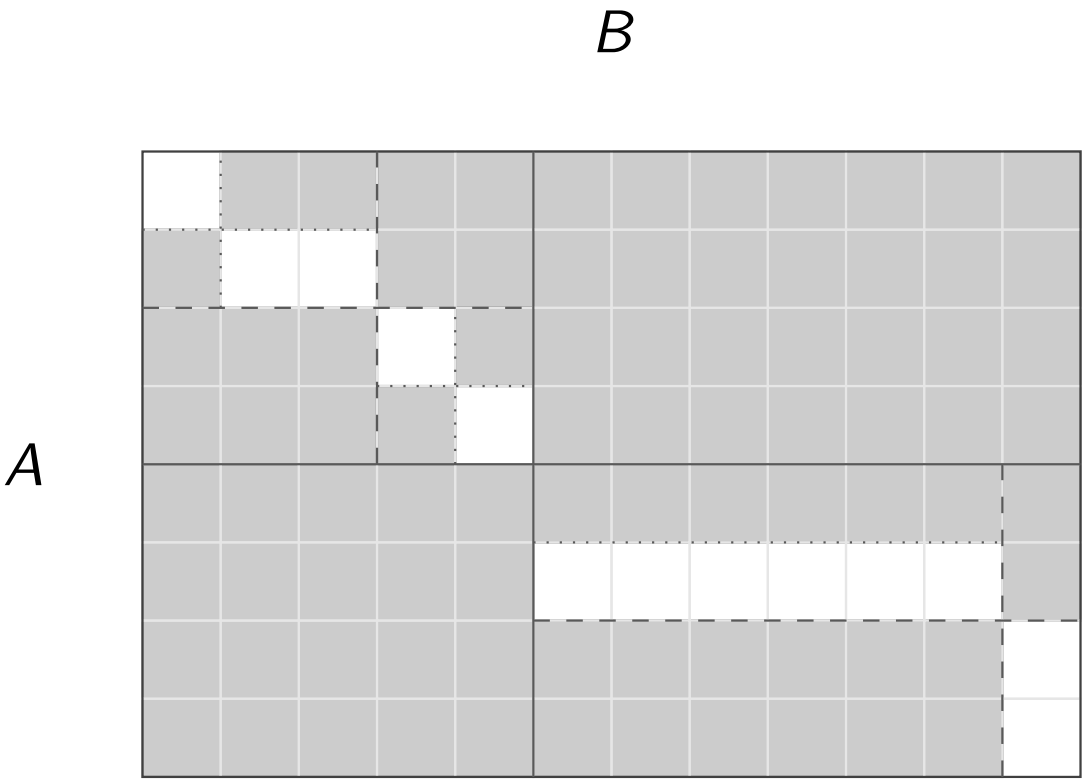


Figure: 分治法求 DP 路徑

例題：Longest Common Subsequence | 線性空間作法

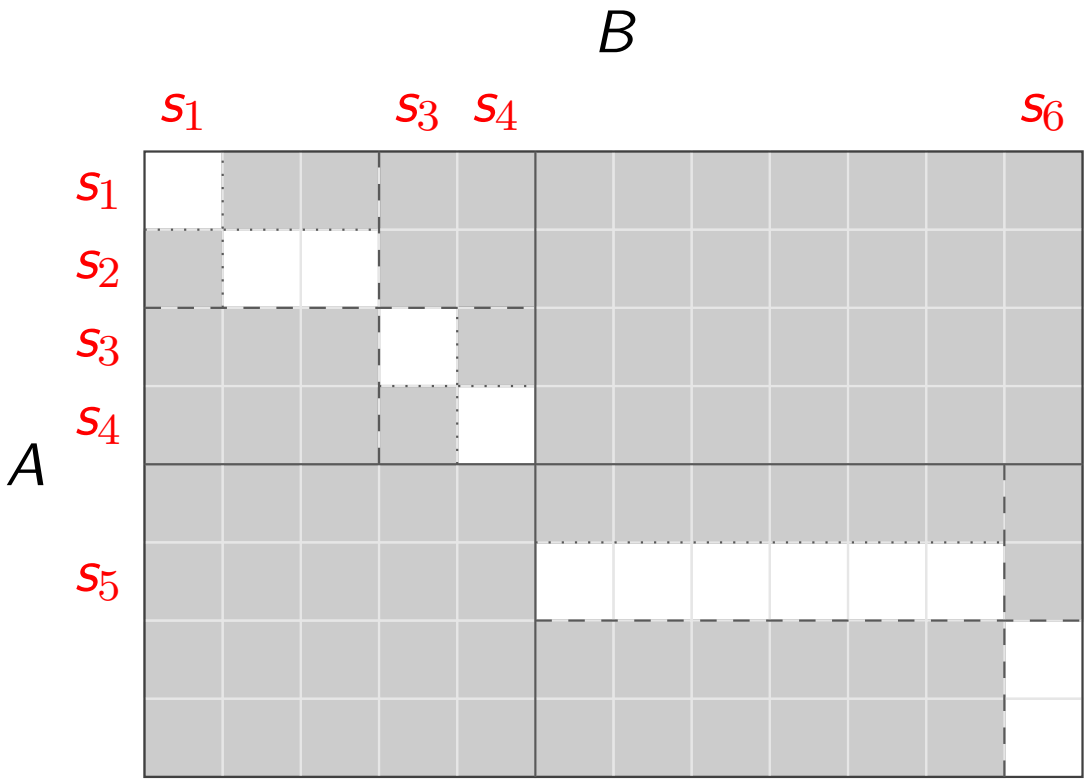


Figure: 分治法求 DP 路徑

例題：Longest Common Subsequence | 線性空間作法

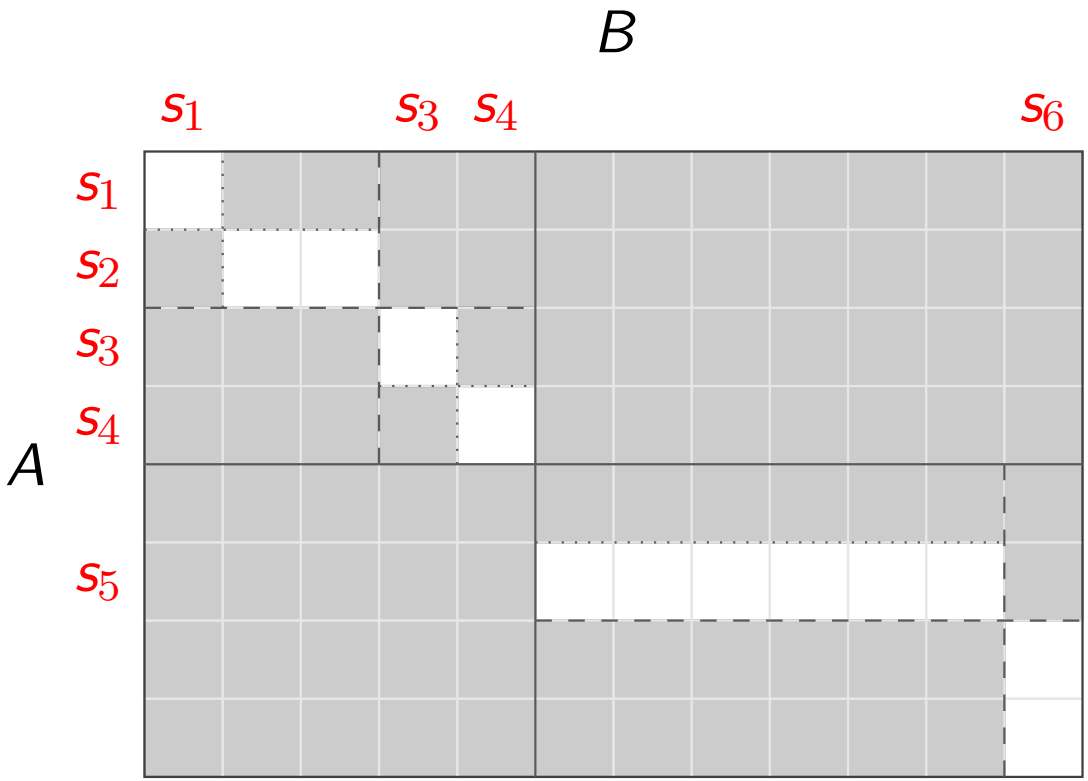


Figure: 分治法求 DP 路徑

$$|A| \cdot |B| + \frac{1}{2}|A| \cdot |B| + \frac{1}{4}|A| \cdot |B| + \cdots = O(|A| \cdot |B|)$$

結語

DP 好棒棒!

例題：圈地問題

阿朗．伍德是一位農夫，他擁有一片他非常自豪的圓形牧場。

他決定派他的獵犬來鎮守牧場邊境。方便起見，讓我們假設阿朗的牧場是一個以原點 $(0,0)$ 為圓心，$R=1$ 為半徑的正圓，而阿朗的農舍則座落在原點上。阿朗在圓上找了 N 個適合讓獵犬鎮守的點，我們姑且稱他們為「哨站」。

然而他只養了 K 條獵犬，因此他只能選擇其中的 K 個哨站防守。更明確地說，阿朗會選擇其中的 K 個哨站讓獵犬防守，並按照他們在圓周上的順序將相鄰的哨站用圍籬兩兩連接。對於這樣一個配置，我們稱其「有效防守面積」為被圍籬圍起的總面積。要注意的是，一個合法的配置必須確定阿朗休息的農舍（即原點）被包含在圍籬中或是恰好在圍籬上。

給定牧圈圓周上的 N 個點，幫阿朗決定在合法的最佳配置下，「有效防守面積」最大可以是多少？