狀態壓縮 State Compression

有時候狀態十分複雜，以致於儲存困難。此時我們可以經由一可逆函式將之轉換成較便於儲存之樣貌。在最常見的狀況下我們都會將之轉換成單一一個整數。

➢ 常見狀態壓縮

1. 各資料之可能狀態集合互不影響

由於各資料間可能的狀態集合互不影響，所以我們能以常數 Ci 代表第 i 項資料可能狀態之集合的大小。此時若我們將資料 i 之可能狀態編號爲 0 ~ Ci - 1，且以 Si 代表當前資料 i 的狀態，則我們可以函式 F(S, C) = S1 + C1 * (S2 + C2 * (S3 + C3 * (……))) 求出一整數來代表此狀態。

《打地鼠 (95 建中校內培訓模擬試題)》 TIOJ 1014

有 n 個從左而右排成一列的地鼠洞，相鄰兩個洞距離皆爲 1。
在每個洞各有一出現周期固定的地鼠，且地鼠被打一次之後就不會再出現。
你在第一個洞的左邊距離 1 處，你每秒可走 1 的距離。
現在給你每個地鼠的周期，求若要打完所有地鼠至少要花多久？

2. 各資料共享一可能狀態集合

如果各資料共享一個可能狀態的集合，我們雖然可當作所有資料之集合互不影響，以第一種方法來儲存，但這樣會有很多不可能的狀態。所以我們可以換個方法，利用字典序將這些狀態依序編號，如此一來就不用爲不必要的東西保留記憶體。

《8-puzzle (TOI2004 初選 problem 4)》 TIOJ 1198

輸入兩個 8-puzzle 的盤面，請求出從第一個盤面變成第二個盤面至少要移幾次。

➢ 例題

《Square (Waterloo local 2002.09.21)》 PKU 2362

給你 n 個木棍，問你能否以端點對端點的連接方式接出一個正方形。

《鍊金術 (第一屆快樂暑假營第三次練習比賽)》 TIOJ 1390

有 n 個物質與兩個 n x n 的表格。
第一個表格的第 i, j 項代表物質 i 與物質 j 融合會獲得多少價值。
第二個表格的第 i, j 項代表物質 i 與物質 j 融合後會留下哪個物質。
求可獲得的最高價值總和是多少。

《Cleaning Robot (Japan 2005 Domestic)》 PKU 2688

給你一個 r x c 的地圖。
o 代表機器人初始位置，* 代表髒東西，x 代表不可走區域，. 代表乾淨區域。
求機器人若要清掉所有的髒東西，至少要走幾格？

《巧拼放置問題 (97 建中校內培訓第三次模擬賽)》 TIOJ 1452

求出以 1 x 2(可旋轉)之巧拼填滿 n x m 地板之方法數。

單調隊列優化

在講這種優化技巧前，我們先介紹一種資料結構：雙向佇列(Double Ended Queue)。通常我們將之簡稱爲 Deque。這種資料結構的雙邊都可以進行 push 和 pop。其實 Stack 與 Queue 都可視爲這種資料結構的特化。像是 SGI 版本的 STL 中 Queue 就是用 Deque 修改出來的。

單調隊列優化的技巧主要用在取極值這個動作。有時候我們必須針對一個序列的好幾個區段取極值，而若這些區段是單調遞增的(即若 $L_i \le L_j$, 則 $R_i \le R_j$)則我們可以不用每次都 O(區段長度)查詢。假設序列長度 n、詢問數 q，我們可以在 O(n + q)的時間內回答所有查詢。

此優化的是利用我們查詢的左界 L 與右界 R 只需要單調遞增之特性。藉此我們可開始推知：

1. **一元素從右界進入後必定只會從左界出去。**
2. **由於 1，越左邊之元素一定會越先出去。**
3. **由於 2，若區間中一較右元素之值比一較左元素之值極端，則較左元素再也不會是極值。**

因爲以上的結論，所以當我們加入一元素時，所有比他差的元素就都可以剔除。於是我們只需用 Deque 維護的一有單調性的序列。在加入新元素時就開始從 Deque 右方 pop 元素，直到最右方的元素比新元素更極端或 Deque 已經空了，再將新元素 push 入。要刪除元素時只需檢查所要刪的是否是當前 Deque 最左端的元素，是即刪。查詢極值時，直接取 Deque 最左端元素之值即可。

➢ 常見單調隊列優化

較常見需要單調隊列優化的題型有三種：

1. **Ans[j] = min or max { val[L[j]] ~ val[R[j]] }**

《簡單易懂的現代都市 (第二屆快樂暑假營最終練習賽)》 TIOJ 1566
給你 n 棟建築物，試問有多少個連續 m 棟建築的區間，最高建築與最矮建築高度差 k 公尺。
(如果區間有一端在邊界則允許只有 2 ~ m-1 棟建築)

2. **Ans[j] = min or max { F(L[j], R[j]) ~ F(R[j], R[j]) }; F(p, R) = val[p] + k * (R - p)**
3. **Ans[j] = min or max { F(1, T[j]) ~ F(j - 1, T[j]) }; F(i, T) = T*V[i] + G(Ans[i]); T[k - 1] ≤ T[k]**

第二、三種在 IOI 與 APIO 各有例題不過因爲有超出範圍或太難，故在此只提供題目名稱。二：IOI 2008 Island。三：APIO 2010 Commando。有興趣的人可自行去查或問講師。

➢ 例題

《城市景觀問題 (第三屆快樂暑假營第一次練習賽)》 TIOJ 1618
一個序列每個位置有兩個值：Hi、Bi。定義一個區間的值是：這個區間內每個比右邊所有的 H 值大的點的 B 值和。問在所有長度爲 k 的區間中，最大值是？
(若區間有一端爲邊界，則允許區間長度爲 1 ~ k - 1)

《K-Anonymous Sequence (POJ Founder Monthly Contest – 2008.12.28)》 PKU 3709
給你一個長度爲 n 的非嚴格遞增序列與一個數字 k。現在你必須減小序列中的某些數字，使修正後的序列中，每種出現的數字都至少有 k 個。然而你把一個數字減小 x，會花掉 x 的 cost。請問你總共至少需要花多少 cost？

背包優化 BackPack Optimization

在限定物件個數的背包問題(多重背包)中，最一般的解法複雜度爲 O(ntc)。然而，有時題目的數據範圍過大使得 O(ntc)無法快速的跑出來。這時我們就必須對算法進行優化。

➢ 多重背包拆解 O(nt * lg(c))

我們都知道，若我們有 2^0, 2^1, 2^2 …… 2^n 幾個數字，那我們必定可以湊出 2^(n + 1) - 1 內的所有數字。相似的，若我們想要湊出一個 c 以內所有的數字，我們可以將 c 拆解爲 k 與 2^n - 1 兩個部分，並使 n 盡量大，接著 2^n - 1 便可以拆解爲 2^0, 2^1, 2^2 …… 2^(n - 1)。如此一來我們若需背包一種數量爲 c 的物體，就可以依照上法將之拆爲 lg(c) + 1 團，我們只需要將每團各自視爲一個新的物體拿去背包即可。

➢ 多重背包之單調隊列優化 O(nt)

有沒有覺得出現眼熟的名詞？沒錯，多重背包的第二種優化其實只是單調隊列優化的應用。在原本的作法中，當我們把一個重量爲 w、值爲 v、數量爲 c 的東西拿來 DP 時，其實我們是枚舉看看這個東西拿幾個時最好。寫成數學式就是：$DP[n][t] = \max_{0 \le j \le c} (DP[n - 1][t - w * i] + v * i)$。
有沒有覺得又似曾相識？沒錯這正是上述的第二種形式！換個寫法：

$DP[n][t] = \max(F(t - w * c, t) \sim F(t, t)); F(p, t) = DP[n - 1][p] + v * (t - p) / w$

可以清楚的看出，DP[n]陣列就是 Ans 陣列、DP[n - 1]陣列就是 val 陣列、v / w 就是 k。
至於實際做法其實很簡單。第二種形式之所以無法用第一種形式的做法，是因爲 F(p, t)比 val[p]還多了一個呈線性之函數 k * (t - p)，而變得不是常數。所以若我們把 F(p, t)改寫一下：

$F(p, t) = (DP[n - 1][p] - p * v / w) + t * v / w$

則因爲 t * v / w 是不受 p 所影響的，可以提出 F(p, t)。所以整個轉移式就能改寫爲：

$DP[n][t] = \max (F(t - w * c) \sim F(t)) + t * v / w; F(p) = DP[n - 1][p] - p * v / w$

因爲對同個 DP[n]，F(p)是定值，所以就變成了第一種形式！只是最後要加上 t * v / w 罷了。所以當我們加入一個新的東西時，就只相當於面對一個新的單調隊列優化問題。不過要注意 w 的每個同餘系互相獨立，也就是 DP[n][a]和 DP[n][b]用同個 Deque，若且唯若 $a \equiv b \pmod w$。所以我們若想照一般背包的順序，t 從 w 一直枚舉到 T，就必須一次開 w 條 Deque！由於這麼作實在太浪費記憶體了，所以我們通常會一個一個同餘系來作。寫成虛擬碼的話：

```
Macro F(p) (DP[k - 1][p] - p * item.v / item.w)
Preocdure BackPack( itemAry, limit, DP)
 1.  For k from 1 to itemAry.length
 2.      For i from 0 to item.w - 1 with item ← itemAry[k]
 3.          deque.clear()
 5.          for j from i to limit step item.w
 6.              If deque.notEmpty() and deque.first() < j - item.c * item.w
 7.                  deque.pop_front()
 8.              While deque.notEmpty() and F(j) ≥ F(deque.last())
 9.                  deque.pop_back()
10.              deque.push_back(j)
11.              DP[k][j] ← F(deque.first()) + j * item.v / item.w
```

動態規劃優化 Dynamic Programming Optimization

決定動態規劃效率的因素有兩者：狀態空間與轉移複雜度。故對動態規劃進行優化時有兩種方向。其一，改變狀態，以使狀態數降低。其二，對轉移步驟進行優化。第一種是狀態設計時的問題，在此不多討論，我們只針對第二種。

➢ 搭配資料結構

在動態規劃的轉移中常常會需要對一個集合求極值或總和之類的東西，此時我們就可以搭配資料結構，使得程式能在取值與改變所求集合兩種操作上優化或達成平衡。較常用到的資料結構有 Deque、Heap、Segment Tree、Balanced BST 等。像上述的背包優化就是個例子。

《小朋友上樓梯 (CKEISC 30th 演算法考題)》 TIOJ 1612
在一個數線上有 n 個點，每個點都有自己的價值。
你要從原點跳到 X 的位置，且剛好在 k 個點上各落腳一次。
若你一次最多只能向右跳 D 的距離，總價值最多可以取到多少？

《電腦檢查 (97 建中校內培訓第六次模擬賽)》 TIOJ 1483
給你一個 n x m 的地圖，每格上有一個數字。
現在你從最左上走到最右下，同時你可以拿走一些經過的數字，但你拿的數字必須嚴格遞增。
在要求至少要拿一個數字的狀況下，求總共有多少種拿法？
(此題用到之資料結構：Binary Index Tree 將在以後的課程上到，故在此不多贅述。)

➢ 使用單調性

第二種優化的方法是使用轉移方程本身的特性，以減少需考慮的子狀態數目。在講這種優化前我們先定義兩種 DP 的問題：

1D/1D DP[j] = $\min_{0\le i<j}$\{ DP[i] + w(i, j) \}; DP[0] = k

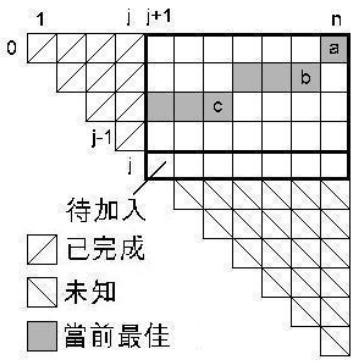
2D/1D DP[i][j] = $\min_{i<k\le j}$\{ DP[i][k - 1] + DP[k][j] \} + w(i, j); DP[i][i] = 0

定義 DP 的類型後，我們來定義單調性。

1. **凹四邊形不等式 (concave Monge condition)**
   若 A 要滿足凹四邊形不等式，則對任意 a < b, c < d 要有 A[a][c] + A[b][d] ≥ A[a][d] + A[b][c]
2. **凸四邊形不等式 (convex Monge condition)**
   若 A 要滿足凸四邊形不等式，則對任意 a < b, c < d 要有 A[a][c] + A[b][d] ≤ A[a][d] + A[b][c]
3. **凹完全單調性 (concave totally monotone)**
   若 A 要滿足凹完全單調性，則對任意 a < b, c < d 要有 A[a][c] ≤ A[b][c] => A[a][d] ≤ A[b][d]
4. **凸完全單調性 (convex totally monotone)**
   若 A 要滿足凸完全單調性，則對任意 a < b, c < d 要有 A[a][c] ≥ A[b][c] => A[a][d] ≥ A[b][d]

其中由 1 可推得 3，由 2 可推得 4。而要判斷一個函數 w(i, j)是否符合凹四邊形不等式，等價於要檢驗 w(i, j) + w(i + 1, j + 1) ≥ w(i, j + 1) + w(i + 1, j) 是否成立。同理，若我們要判斷函式 w(i, j)是否符合凸四邊形不等式，也只需檢驗 w(i, j) + (i + 1, j + 1) ≤ w(i, j + 1) + w(i + 1, j)。

凹性 1D/1D 優化



若定義 F[i][j] = DP[i] + w(i, j)，則若 w(i, j)符合凹四邊形不等式，可推知 F 必定也符合。此時 F[i][j]即具有凹完全單調性。而根據定義，DP[j]會是 $\min_{0\le i<j}$F[i][j]，所以我們用一資料結構來維護每條尚未計算之行的當前最小值在哪。且若 DP[j]已完成，由凹四邊形不等式，F[j + 1] ~ F[n]的當前最佳解位置必遞減。

右圖是剛得出 DP[j]而未將 DP[j]加入資料結構前的狀態，一但我們成功將第 j 列加入後，我們即可直接將第 j+1 行的當前最佳解視爲 DP[j + 1]之值。(因爲此時 F[0][j + 1] ~ F[j][j + 1]都被考慮到了)所以 DP 的瓶頸是在我們如何維護此結構。

一般而言對 1D/1D 凹的問題我們使用 Stack 來維護當前最佳解，Stack 中的每個元素都包含三個值(L, R, p)，代表在 L ~ R 行得當前最佳解是 p。當我們要加入新的一行 j 時，有兩種 case：

1. **F[j][j + 1] ≥ F[Stack.top().p][j + 1]**

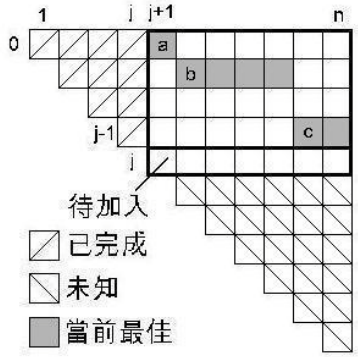
   這代表在第 j+1 行時最佳解的位置就已經比 j 還要小了，故此行根本無需加入。

2. **F[j][j + 1] < F[Stack.top().p][j + 1]**

   此時代表列 j 是有必要加入的，於是我們從 Stack 的頂端開始考慮。若以(nL, nR, np)代表當前 Stack 頂端元素的(nL, nR, np)則若 F[j][nR] < F[np][nR]就將 Stack 頂端的元素丟棄，如此反覆直到 Stack 已經空了或是 F[j][nR] ≥ F[np][nR]。當發生第一種狀況時直接 push 入(j + 1, n, j)即可，然而若是第二種狀況則需在(nL, nR)之間二分搜，找到第一次發生列 np 比列 j 好的地方。設此點爲 m，則需將 Stack 頂端之元素修正爲(m, nR, np)後再 push 入(j + 1, m - 1, j)。最後只需在取值後判斷 Stack 頂端之元素是否過期(nR < j + 1)，若是即將之 pop 出便可。

根據以上方法我們可得一算法在 O(nlgn)內求出 DP[n]。

凸性 1D/1D 優化



凸性的 1D/1D 優化與凹性十分相似。在凸四邊形不等式的影響下，當前最佳解的位置是呈現遞增，所以在凸性 1D/1D 問題中我們改用 Deque 來維護當前最佳位置。要插入新的一列時，我們就從右邊開始刪除元素，最後一樣二分搜出確切位置。而要取值時則是從左側取出，過期的東西也是從左側拿出。

這個作法的複雜度與凹性 1D/1D 一模一樣是 O(nlgn)。不過不管是凹性還是凸性，如果能利用 w(i, j)函數的特性而在 O(1)的時間內計算出 m 點所在，複雜度便可降至 O(n)。

凹性 2D/1D 優化

對於凹性的 2D/1D 問題可考慮枚舉一維後每次視爲 1D/1D 的問題。定義：

$$\mathbf{DP_i[j] = \min_{0\le k<j}\{ DP_i[k] + u_i(k, j) \} \ (1 \le j \le n - i)}$$

$$\mathbf{u_i(k, j) = DP_{i+k+1}[j - k - 1] + w(i, i + j)}$$

若可證明 $u_i(k, j)$符合凹四邊形不等式，則可以凹性 1D/1D 之法解 DP_i。總複雜度爲(n^2lgn)。不過要注意的是因爲 i 值較小的 u 會用到 i 值較大的 DP，故枚舉 i 值時須從大枚舉至小。

凸性 2D/1D 優化

如果 w(i, j)符合凸單調性，且有 w(i, i + 2) ≥ max(w(i, i + 1), w(i + 1, i + 2))，則 DP[i][j]也會符合凸四邊形不等式。

證明：
欲證明對所有 a<b≤c<d，有 D[a,c]+D[b,d] ≤D[a,d]+D[b,c]
對 d-a 使用數學歸納法，
當 d-a=2 時 b=c=a+1，由區間單調關係 w(a,a+2)≥max(w(a,a+1),w(a+1,a+2))，故
D[a,d]+D[b,c]=w(a,a+2)+min(w(a,a+1),w(a+1,a+2))+0≥
w(a,a+1)+w(a+1,a+2)=D[a,a+1]+D[a+1,a+2]=D[a,c]+D[b,d]
若 d-a≤k 時成立，
當 d-a=k+1 時，設 D[a,c]=w(a,c)+D[a,x]+D[x+1,c], D[b,d]=w(b,d)+D[b,y]+D[y+1,d],
D[a,d]=w(a,d)+D[a,z]+D[z+1,d], D[b,c]=w(b,c)+D[b,w]+D[w+1,c]
若 z<w，則 a≤z<w<c，b≤w<c<d，有
D[a,c]+ D[b,d]= w(a,c)+ w(b,d)+D[a,x]+D[x+1,c] +D[b,y]+D[y+1,d]
≤w(a,c)+ w(b,d)+D[a,z]+D[z+1,c] +D[b,w]+D[w+1,d] (因狀態轉移方程)
≤w(a,d)+ w(b,c)+D[a,z]+D[z+1,d] +D[b,w]+D[w+1,c] (因歸納假設且 a<z+1<w+1≤c<d)
= D[a,d]+D[b.c]
若 z≥w，則 a<b≤w<c，b≤w≤z <d，有
D[a,c]+ D[b,d]= w(a,c)+ w(b,d)+D[a,x]+D[x+1,c] +D[b,y]+D[y+1,d]
≤w(a,c)+ w(b,d)+D[a,w]+D[w+1,c] +D[b,z]+D[z+1,d] (因狀態轉移方程)
≤w(a,d)+ w(b,c)+D[b,w]+D[w+1,c] +D[a,z]+D[z+1,d] (因歸納假設且 a<b≤w<z<d)
= D[a,d]+D[b.c]
故 d-a=k+1 時成立，由數學歸納法有 D[i,j]滿足凸四邊形不等式

相較於凹性 2D/1D，凸性 2D/1D 能作的優化更多，他有一個重要的定理：
令 K[i, j]爲使 DP[i][j]達到最小值的 k 值，則有 K[i][j - 1] ≤ K[i][j] ≤ K[i + 1][j]。

證明：令 k= K[i,j-1]，對所有 l<k，有 l<k≤j-1<j，
由 D 之四邊形不等式，D[l,j-1]+D[k,j]≤D[l,j]+D[k,j-1]
左右同加 D[i,l]+D[i,k]，得(D[i,l]+D[l,j-1])+(D[i,k]+D[k,j])≤(D[i,l]+D[l,j])+(D[i,k]+D[k,j-1])
由 D[i,l]+D[l,j-1]≥D[i,k]+D[k,j-1]，有 D[i,k]+D[k,j]≤D[i,l]+D[l,j]
故 k 較 l 佳，即 K[i,j-1]≤K[i,j]
同理，K[i,j]≤K[i+1,j]

因爲以上定理，我們 DP 時，可以從 j - i 較小的狀態開始 DP 起。當我們在作 DP[i][j]時我們只需枚舉 K[i][j - 1] ~ K[i + 1][j]，故對於所有 j - i = c 的狀態，將他們都求出所需枚舉的狀態數只有：

K[2][1 + c] - K[1][c] + 1 + K[3][2 + c] - K[2][1 + c] + 1 + K[4][3 + c] - K[3][2 + c] + 1 ….

= K[n + 1 - c][n] - K[1][c] + n - c = O(n)。

而 j - i 總共只有 O(n)種，故求出整個 DP 表格便只需要 O(n^2)，比起直接枚舉快了一維。切記 2D/1D 的凸單調性優化算是這四種中在競賽最常見的，甚至在 TOI 模考中也出過，所以就算其他三種單調優化都不會，也得弄清楚這種。

※以上證明皆擷取自阿思講義