基礎動態規劃

yp155136

January 25, 2022

講師簡介

- yp155136
- LYB / BBCube
  - ICPC 2021 台北站第二名
  - ICPC 2019 雅加達站第四名
- IOI 2018 國際資訊奧林匹亞銀牌
- APIO 2018 亞太資訊奧林匹亞金牌

講師簡介

- yp155136
- LYB / BBCube
  - ICPC 2021 台北站第二名
  - ICPC 2019 雅加達站第四名
- IOI 2018 國際資訊奧林匹亞銀牌
- APIO 2018 亞太資訊奧林匹亞金牌
- 因為想學習基礎 DP 所以當基礎 DP 講師

大綱

- 單調隊列優化
- 斜率優化
- 凹凸單調優化
- Knuth Optimization
- 轉移點單調優化
- SMAWK 演算法
- Aliens 優化

前言

- 相信大家已經具備基本的 DP 能力了
- 基礎 DP 這節課，是要教大家許多 DP 優化的技巧
- 大家可以想像 DP 優化是一個工具，用來幫助你更快的計算 DP 式子

単調隊列優化

單調隊列優化 – 例題

題目 (ZJ a146 Sliding Window)

給定一個長度為 N 的序列 A 以及一個整數 K，請找出所有長度為 K 的窗口極值。也就是說，對於所有的 $K \leq i \leq N$，求出 $\min_{i-K+1 \leq j \leq i} A_j$ 以及 $\max_{i-K+1 \leq j \leq i} A_j$。

- $N, K \leq 10^6$

比如說，如果 $A = (4, 3, 5, 1, 6), K = 3$，那你要分別輸出 $(5, 5, 6)$ 跟 $(3, 1, 1)$

單調隊列優化

- naive 作法：$O(NK)$，太慢了
- 因此，我們需要一些觀察
- 假設我們在做 min 的情況

單調隊列優化

- 假設我們已經計算完尾端是 i 的答案
- 那，對所有 $j < i$，如果 $A_j > A_i$，那 A_j 就永遠不可能是答案了
- $A = [9, 6, 2]$，在看到 2 之後，6 就不可能成為答案了
- 因此，我們可以維護一個結構，來儲存**目前有可能是答案的元素**
- 想想看，這個結構裡面的元素，應該是遞增的，還是遞減的？

單調隊列優化

- 結構裡面的元素是遞增的
- 後面的元素，雖然目前暫時不是最佳解，**但是等到前面的元素過期（window 大小超過 K）的時候**，就有可能變成最佳解了
- 算法結構如下：
  - 每次要將 i 放入之前，可以先從後面將所有大於 A_i 的元素全部 pop 掉（因為被 pop 掉的元素不可能成為最佳解了）
  - 每次計算答案的時候還需檢查在資料結構最前面（基於單調性，一定是目前裡面最小的元素）的元素是否還合法，不合法的話要 pop 掉
  - 我們便可以確定此時資料結構最前的元素就是位置 i 的答案
- 那個資料結構需要支援從前面刪除元素，從後面加入及刪除元素，雙向佇列（double-ended queue, deque）是一個不錯的選擇，且 STL 也有提供 `std::deque` 容器，不需要自己手寫。

單調隊列優化

假設 $A = [4, 6, 8, 7, 9, 1], K = 3$

4	6	8	7	9	1

單調隊列優化 – 範例程式碼

```
1  void find_min(int n, int k) {
2    deque<int> dq;
3    for (int i = 1; i <= n; ++i) {
4      while (!dq.empty() && a[dq.back()] > a[i])
       ↪  dq.pop_back(); // 把所有大於 a[i] 的元素通通刪掉
5      dq.push_back(i);
6      if (dq.front() == i - k) dq.pop_front(); // 如果最前面
       ↪  的元素已經不合法（過期），就 pop 掉
7      ans_mn[i] = a[dq.front()];
8    }
9  }
```

單調隊列優化

- 時間複雜度？$O(N^2)$？
- 均攤分析：考慮**總共的操作數量**（在這題來說就是看 pop 跟 push 的數量）
- push 至多 N 次，因此也至多 pop N 次
- 因此總操作數量為 $O(N)$，每個操作是均攤 $O(1)$ 的
- 因此，總操作的複雜度為 $O(N)$

單調隊列優化 – 應用

題目 (CF 372C Watching Fireworks is Fun)

有 M 個煙火將在一維數線上綻放。第 i 個煙火在時間 t_i 、於位置 a_i 綻放，其中 $1 \leq a_i \leq N$。如果你在位置 $1 \leq x \leq N$ 觀看煙火 i 的話，你會獲得 $b_i - |a_i - x|$ 的開心度。此外，你每秒可以移動 d 單位的距離，且不能離開 1 到 N 的位置範圍。你可以任選初始位置，請問看完 M 的煙火所獲得的總開心度最大能多少。

- $1 \leq M \leq 300$
- $1 \leq N \leq 1.5 \times 10^5$

先想想看 DP 式子要怎麼列，再看看能不能用單調隊列優化達到更好的複雜度

單調隊列優化 – 應用

- 首先將所有的煙火按照綻放的時間，由小到大排序好
- 令 $dp(i,j)$ 為看完第 i 個煙火且位於位置 j 的最大開心度
- $dp(0,i) = 0$ 對所有 $1 \leq i \leq N$
- $dp(i,j) = \max_{|k-j| \leq d \times (t_i - t_{i-1})} dp(i-1,k) + b_i - |a_i - j|$
- 注意到 $b_i - |a_i - j|$ 跟 k 完全無關，所以重要的其實是求出所有在可到達範圍內的 $dp_{i-1,k}$ 的最大值
- 由於 $|k-j| \leq d \times (t_i - t_{i-1})$ 這個限制其實是 $\max(1, j - d \times (t_i - t_{i-1}) \leq k \leq \min(N, j + d \times (t_i - t_{i-1}))$，所以這個問題其實就是一個滑動窗口極值的計算

單調隊列優化 – 應用

題目 (有限個數的多重背包問題)

有 N 種物品，第 i 種重量為 w_i，價值為 v_i，且總共有 s_i 個。求選出若干個物品使得總重量不超過 W 的情況下，最大總價值為何。

最 Naive 的作法是 $O(NWK)$，如果用二進制拆解的話可以壓到 $O(NW \log K)$（K 是 s_i 的最大值），有沒有辦法作到更好呢？

單調隊列優化 – 應用

- 令 $dp(i,j)$ 代表前 i 種物品中，選出總重量為 j 的最大價值和
- $dp(i,j) = \max_{0 \le x \le s_i} \{dp(i-1, j - x \times w_i) + x \times v_i\}$
- $dp(i,j) =$
  $\max_{0 \le \frac{j-k}{w_i} \le s_i, j \equiv k \pmod{w}_i} \left\{dp(i-1,k) + \frac{j-k}{w_i} \times v_i\right\}$
- $dp(i,j) = \lfloor \frac{j}{w_i} \rfloor \times v_i +$
  $\max_{0 \le \frac{j-k}{w_i} \le s_i, j \equiv k \pmod{w}_i} \left\{dp(i-1,k) - \lfloor \frac{k}{w_i} \rfloor \times v_i\right\}$
- 如此一來，對於一個除以 w_i 的餘數 r，上面的轉移式就是一個滑動窗口的極值問題，因此可以套用單調隊列的手法加速達到 $O($除以 w_i 餘 r 的個數$)$，對所有餘數取總和的話恰好就是 $O(W)$，於是有限背包問題便得到了 $O(NW)$ 的解法

斜率優化

斜率優化 – 形式

- 斜率優化又稱凸包優化（convex-hull optimization），幾乎是所有 DP 優化的技巧中，最常見也最容易套用的一種
- 常見的 DP 式子形式：$dp(i) = c_i + \max_{j \le R_i}\{a_j x_i + b_j\}$
- 其中 R_i 是一個遞增的序列，而 c_i 是一個與取 max 沒有直接關係的項，可以視為不存在
- 通常來說 a_j 或 b_j 會包含 $dp(j)$ 的值，因此在這種情況下會額外的有 $R_i < i$ 的限制

斜率優化

- 與剛才介紹的單調隊列優化很相似，對於兩個 j, k，如果有 $a_k > a_j$ 且 $a_k x + b_k > a_j x + b_j$
- 那麼對於所有的 $x' > x$ 而言，j 都不可能是最好的轉移來源
- 基於這個觀察，我們可以對已經算好的 $j \leq R_i$，都將 $a_j x + b_j$ 當成是二維平面上的一條直線畫出來
- 如此一來，我們的 DP 轉移就變成要查詢 x_i 帶入這些線的最大值，以及要支援動態的將新的直線加入
- 先介紹斜率與查詢都是單調的版本，接下來才會進入斜率、查詢不單調的版本

斜率優化 – 斜率、查詢單調

題目 (ZJ a146 Sliding Window)

現在你要過 N 個關卡，關卡編號為 $1, 2, \ldots, N$，每一個關卡都有一隻怪獸，第 i 個關卡的怪獸的能力值為 s_i。在還沒進入任何關卡之前，你有一個初始的技能值 x。
在第 $1, 2, \ldots, N-1$ 關卡中，你可以選擇打倒怪獸，或者是直接逃跑。如果你選擇打倒第 i 隻怪獸，你要花 $f \times s_i$ 的時間打倒怪獸，其中 f 是你當前的技能值。在打倒怪獸後，你的新的能力值會變成 f_i。
你的最終目標是打倒第 N 隻關卡的怪獸，請問在這個過程中，你最少需要花費多少時間。

- $N \leq 2 \times 10^5$
- $1 \leq s_1 \leq s_2 \leq \cdots \leq s_N \leq 10^6$
- $x \geq f_1 \geq f_2 \cdots \geq f_N \geq 1$

斜率優化 – 斜率、查詢單調

- 假設 $f_0 = 0, dp(0) = 0$，應該不難列出以下的 DP 式子：
- $dp(i) = \min_{0 \le j < i}(dp(j) + f_j \times s_i)$
- 為了之後講解方便，我們先把所有的 f_i 設成 $-f_i$，我們就可以把 DP 式子轉換成取最大值的形式
- $dp(i) = \max_{0 \le j < i}(dp(j) + f_j \times s_i)$
- 並且最後的答案為 $-dp(N)$

斜率優化 – 斜率、查詢單調

- $dp(i) = \max_{0 \le j < i}(dp(j) + f_j \times s_i)$
- 如果把上面那個 DP 式寫成斜率優化的形式
- 斜率 $a = f_j$，截距 $b = dp(j)$
- 轉移 $dp(i)$ 的時候，就是要查詢在 $x = s_i$ 的情況下，那些直線的最大值
- 特別的是，我們可以發現：查詢 s_i 跟斜率 f_j 都是**單調遞增**的

斜率優化 – 斜率、查詢單調

- 因此，在這個條件下，這個轉移式會跟前面介紹的單調隊列優化有類似著性質
- 如果 $k < j$ 是 i 的最佳轉移來源，那麼對於所有 $i < i'$，從 j 轉移一定會比從 k 轉移來還的好
- $a_j x_i + b_j > a_k x_i + b_k \implies a_j x_{i'} + b_j = a_j x_i + a_j(x_{i'} - x_i) + b_j > a_k x_i + a_k(x_{i'} - x_i) + b_k = a_k x_{i'} + b_k$
- 有了這個性質以後，我們可以進一步的將時間複雜度壓低

斜率優化 – 斜率、查詢單調

- 跟單調隊列一樣，我們維護一個直線的 deque，其中直線由頭至尾斜率遞增
- 每次轉移的時候我們可以考慮最前面兩個直線 L_1, L_2，比較是否有 $L_1(x_i) < L_2(x_i)$
- 根據上面的性質，如果成立的話則代表 L_1 再也不可能成為之後的最佳轉移來源，所以可以將 L_1 移除
- 重複這個動作直到現在 deque 最前面的元素 L 還沒有被淘汰，則這時 L 便會是 i 最佳的轉移來源

斜率優化 – 斜率、查詢單調

- 解決完查詢以後，接著必須也將加入直線的複雜度降低
- 由於斜率單調遞增，我們可以確定現在要加入的直線 L' 一定比現在 deque 中的直線斜率都還要大
- 而不難發現，L' 能淘汰的人會是 deque 尾端的一些**連續**的直線
- 假設現在 deque 中最後面的元素為 L_{-1}，倒數第二個為 L_{-2}
- 則可以發現，當 L' **與** L_{-1} **的交點比** L_{-1} **以及** L_{-2} **的交點還要左邊**時
- L_{-1} 的有效區間會完全的被 L' 覆蓋，再也不可能成為轉移的候選人

斜率優化 – 斜率、查詢單調

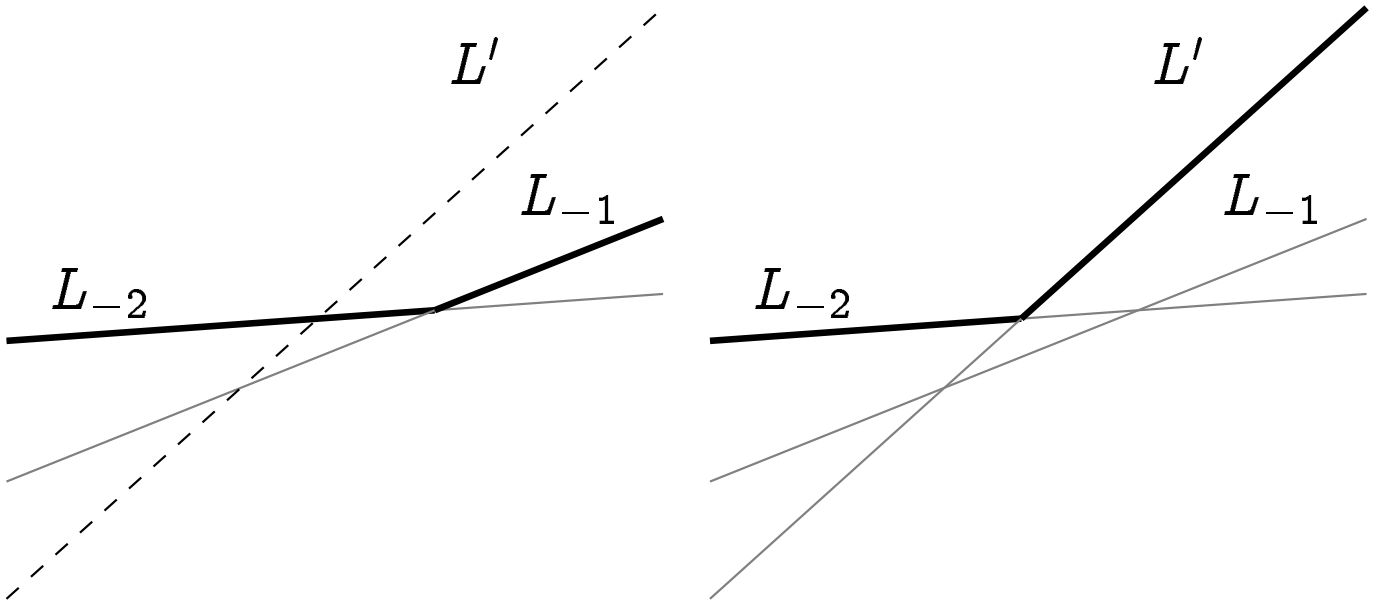


Figure: L' 與 L_{-2} 合力將 L_{-1} 淘汰

斜率優化 – 斜率、查詢單調

- 至此，單調版本的斜率優化已經成形
- 而根據與單調隊列優化相似的論述，每條直線只會被加入一次，並至多被移除一次
- 由於使用 deque，每次加入以及刪除都是 $O(1)$，所以總時間複雜度可以降為 $O(N)$
- 至於將最大值改為最小值的版本，則留給讀者自行推導

斜率優化 – 斜率、查詢單調

```cpp
typedef long long ll;

struct Line {
  ll a, b; // 一條 ax + b 的直線
  Line(ll _a, ll _b): a(_a), b(_b){}
  ll operator()(const ll x) {
    return a * x + b;
  }
};
```

斜率優化 – 斜率、查詢單調

```cpp
bool check(Line l1, Line l2, Line l3) {
  // l1 是講義中的 L_{-2}，l2 是講義中的 L_{-1}，l3 是想要新增的直線
  // double v12 = (l1.b - l2.b) / (l2.a - l1.a)
  // double v23 = (l2.b - l3.b) / (l3.a - l2.a)
  // return v12 >= v13
  // 但是上面的方法會有浮點數誤差，因此在這裡只考慮使用整數運算，方法如下：
  return (l3.a - l2.a) * (l1.b - l2.b) >= (l3.b - l2.b) * (l1.a - l2.a);
}
```

斜率優化 – 斜率、查詢單調

```
1  void solve(int n) {
2    for (int i = 1; i <= n; ++i) {
3      while ((int)dq.size() >= 2 && dq[0](s[i]) <=
       ↪  dq[1](s[i])) {
4        // 把比較差的線丟掉，注意到這邊寫 <= 或 < 其實都 ok
5        dq.pop_front();
6      }
7      dp[i] = dq[0](s[i]);
8      Line l = Line(f[i], dp[i]);
9      while ((int)dq.size() >= 2 &&
       ↪  check(dq[(int)dq.size() - 2], dq[(int)dq.size()
       ↪  - 1], l)) {
10       // 把新的線加進去，看看 L_{-2} 跟 L 有沒有辦法把 L_{-1}
         ↪  殺掉
11       dq.pop_back();
12     }
13     dq.push_back(l);
14   }
```

斜率優化 – 會單調過期的斜率、查詢單調

- 有時候在斜率單調的優化時，會遇到一條線再加入之後會單調過期（也就是過期的時間與斜率都具有單調性）的版本
- 一個例子就是當 DP 轉移只能轉移前 K 項時，那 $dp(i)$ 所代表的那條線在 $i+K$ 的時候就會被迫移出凸包
- 直接來看例題吧～

斜率優化 – 會單調過期的斜率、查詢單調

題目 (NEOJ 186 烏龜疊疊樂改)

給你一個長度為 N 的序列 $a_1, a_2, \ldots, a_N$，以及一個常數 K。請你把序列分成 M 段，每段的長度不超過 K。假設第 i 段的開頭為 l_i，結尾為 r_i。請你最大化：
$\sum_{i=1}^{M}((\sum_{x=l_i}^{r_i} a_x) \times i - (r_i - l_i + 1)^2)$

- $1 \leq K \leq N \leq 5 \times 10^5$

註：上面的式子跟原題稍微不太一樣

斜率優化 – 會單調過期的斜率、查詢單調

- 把 a 序列進行翻轉（reverse），並且定義 pre 陣列為 a 序列的前綴和陣列：$pre_i = \sum_{x=1}^{i} a_x$
- 定義 $dp(i)$ 代表我們看到序列的第 i 項時，可能的最大值
- 則我們可以列出以下的 DP 式子
- $dp(i) = pre_i + \max_{\max(i-K,0) \le j < i}(dp(j) - (i-j)^2)$
- 把式子改寫成斜率優化的形式（斜率為 $2j$，截距為 $dp(j) - j^2$）
- $dp(i) = pre_i - i^2 + \max_{\max(i-K,0) \le j < i}(2j \times i + dp(j) - j^2)$
- 而我們可以注意到，因為每段的長度不能超過 K，因此**轉移的線會單調過期**

斜率優化 – 會單調過期的斜率、查詢單調

- 考慮新的直線加入凸包時的狀況
- 當新的直線 L 與凸包中倒數第二的直線 L_{-2} 合力「殺掉」凸包中最後的直線 L_{-1} 時
- 若直線會被淘汰掉，那麼其實不能直接將 L_{-1} 移除，因為 L_{-1} 能存活的時間比 L_{-2} 久
- 當 L_{-2} 過期之後，L_{-1} 可能重新「復活」變成最佳轉移來源

斜率優化 – 會單調過期的斜率、查詢單調

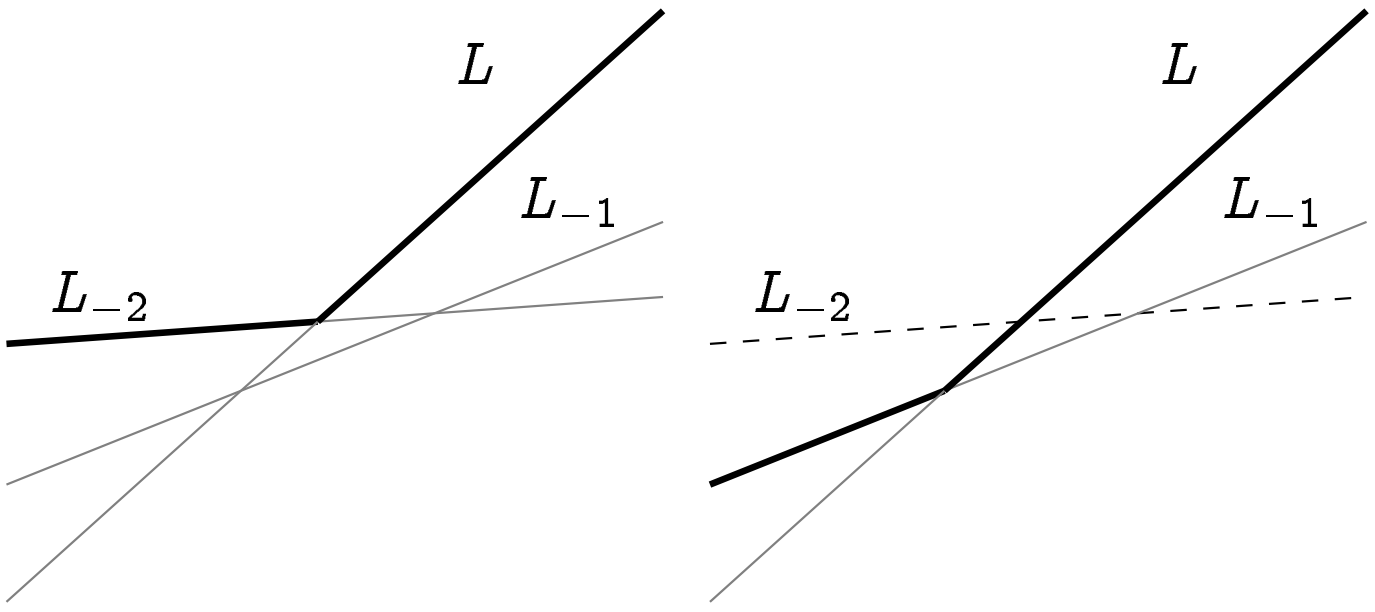


Figure: 當 L_{-2} 過期之後，L_{-1} 又重回凸包上。

斜率優化 – 會單調過期的斜率、查詢單調

- 那這個時候 L 與 L_{-2} 合力殺掉 L_{-1} 的條件就是「就算 L_{-2} 過期 L 還是一直比 L_{-1} 優」
- 也就是說在檢查交點的時候，除了判斷 (L, L_{-1}) 與 (L_{-2}, L_{-1}) 這兩組交點外，還需要考慮 L_{-2} 過期時的查詢位置 x
- 所以**只要當 (L, L_{-1}) 的交點比 x 還有 (L_{-2}, L_{-1}) 都還要左邊時，才能完全淘汰 L_{-1}**。

斜率優化 – 會單調過期的斜率、查詢單調

```cpp
ll DivCeil(ll x, ll y) { // 計算 x / y 取上高斯，不管 x, y 的正負情況都適用
  return x / y + (((x < 0) != (y > 0)) && (x % y));
}

bool check(Line l1, Line l2, Line l3) {
  // 注意到這題點只有定義在整數上，因此可以用下面的技巧來避免浮點數
  ll v1 = DivCeil((l1.b - l2.b), (l2.a - l1.a)); // L_{-2} 與 L_{-1} 的交點。在這邊代表 L_{-1} 第一個超越 L_{-2} 的整數點
  ll v2 = DivCeil((l2.b - l3.b), (l3.a - l2.a)); // L_{-1} 與 L 的交點。在這邊代表 L 第一個超越 L_{-1} 的整數點
  ll v3 = l1.i + k; // L_{-2} 最後一個合法的位置， v3 + 1 就過期了
  return v2 <= v3 && v2 <= v1;
}
```

斜率優化

- 從這邊開始，要介紹斜率優化最 general 的版本
- 同樣的，我們先假設 DP 式子是以下的形式：
- $dp(i) = c_i + \max_{j \le R_i}\{a_j x_i + b_j\}$
- 跟前面的章節類似，我們會把 $a_j x + b_j$ 當成二維平面上的一條直接去畫出來
- 讓 DP 轉移變成帶入查詢 x_i 得到最大值，並且我們要支援新的直線的加入

斜率優化

- 由於最大值一定在這些直線所形成的下凸包中（若為最小值的話改為上凸包）
- 我們可以將所有的線照斜率排好，並且對於每條直線維護它出現在凸包上的 x 軸範圍
- 實作上可以直線 L 維護 p_L，代表 L 最左從 p_L 開始出現在凸包上（L 右界就是下一條線的左界）
- 如此一來，每次查詢 x 的時候只要二分搜到斜率最大的 L'，使得 $p'_L \leq x$ 即可。

斜率優化

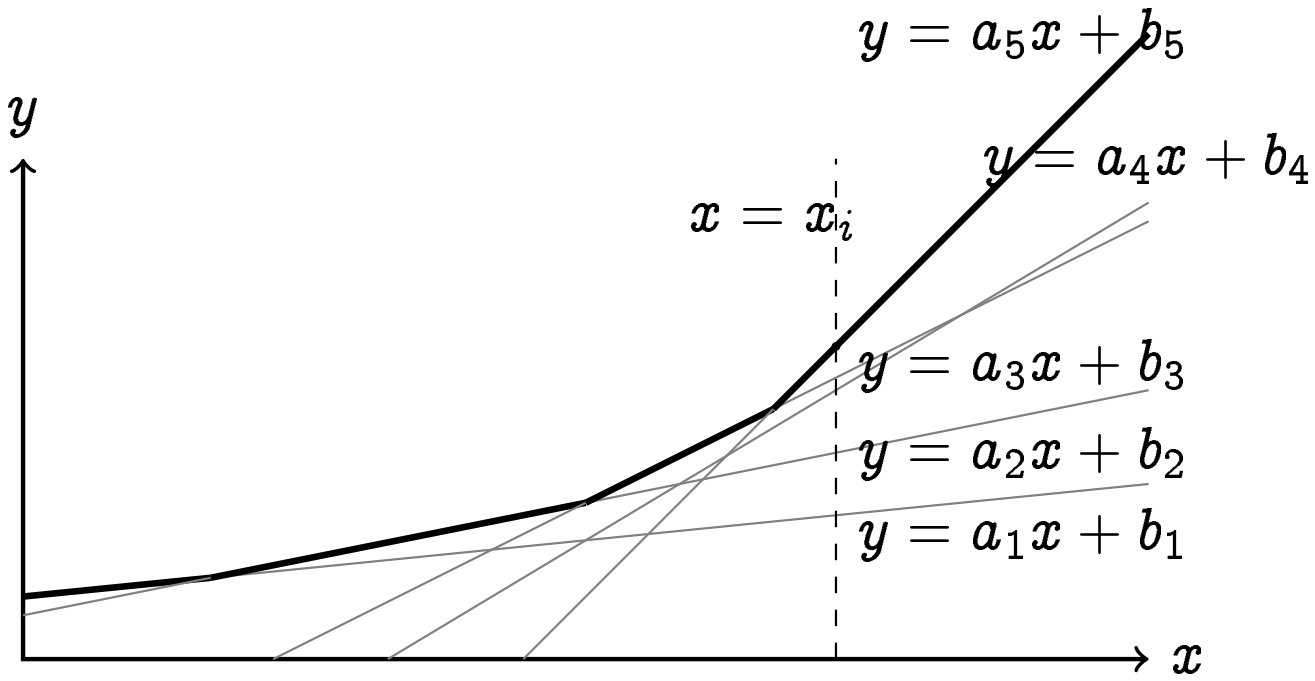


Figure: 將所有 $j \leq R_i$ 的 $y = a_jx + b_j$ 都畫在二維平面上。其中 $y = a_4x + b_4$ 並不在凸包上，不可能成為轉移來源。$x = x_i$ 的最佳轉移來源為 $y = a_5x + b_5$。

斜率優化

- 解決了查詢之後，接著必須考慮當 R_i 增加的時候，該怎麼維護這個凸包
- 首先，可以用斜率來二分搜以找到新加入的直線在凸包上的位置
- 加入了之後，有一些原本在凸包上的直線會從凸包中被移除，有一些的有效區間則會被修改，且這些被淘汰的直線原本在凸包會是一個**連續的區間**
- 具體來說，假設現在凸包中的直線斜率由小至大分別為 $(L_1, L_2, \ldots, L_k)$，且新的直線 L' 被加入凸包的位置為 L_i 以及 L_{i+1} 之間，則會存在 j 與 k 使得 $\{L_k, L_{k+1} \ldots, L_i, L_{i+1}, \ldots, L_j\}$ 被 L' 淘汰，且 L_{k-1} 與 L_{j+1} 的有效區間被更新

斜率優化

- 那要如何更新有效區間呢？
- 由於 L' 不能完全淘汰 L_{j+1} 的原因便是 L' 在完全打敗 L_{j+1} 之前就與 L_{j+1} 相交了
- 因為 L' 的斜率小於 L_{j+1}，相交就代表在交點 x_0 之後，$L'(x)$ 的值就不可能在比 $L_{j+1}(x)$ 來得大了
- 所以 L' 的有效區間的右界就到 x_0，而 L_{j+1} 的有效左界則會被改為 x_0
- 同樣的論述可以套用在 L_{k-1} 上，不過由於 L_{k-1} 的斜率小於 L'，會變成改動 L_{k-1} 的有效右界

斜率優化

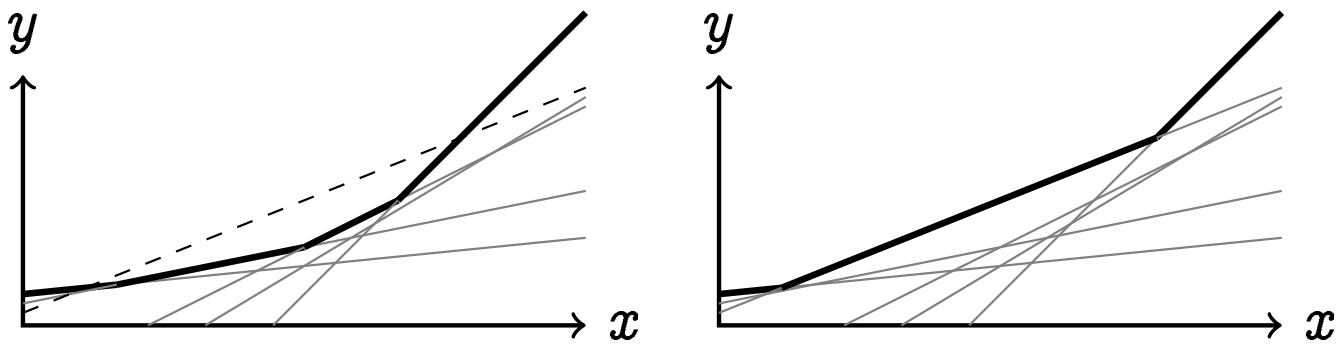


Figure: 加入新的直線後，$y = a_2x + b_2$ 以及 $y = a_3x + b_3$ 被完全淘汰。$y = a_1x + b_1$ 與 $y = a_5x + b_5$ 的有效右界以及有效左界分別被改為其與新線的交點。

斜率優化

- 那這樣的時間複雜度會是多少呢？
- 每次查詢以及插入需要使用若干次二分搜，一次為 $O(\log N)$
- 並且在插入後會刪掉一些數量不等的元素，一次刪除的複雜度也是 $O(\log N)$
- 這邊我們可以套用在上一個章節所使用的**均攤分析**來計算：每一條直線只會被加入凸包中一次，且至多只會被移除一次
- 每次加入跟移除都是 $O(\log N)$，所以均攤下來每一次操作都是 $O(\log N)$ 的
- 如此一來，我們便得到一個時間複雜度 $O(\log N)$ 的算法，相比原本的 $O(N^2)$ 是一個極大的改進

斜率優化

- 實作上，由於我們需要支援：
- 對於給定的 x，二分搜斜率最大的 L 使得 $p_L \leq x$
- 對於一條直線 L，二分搜找到它在凸包的位置並且插入
- 刪除一條直線

斜率優化

- 一個理想的結構為一個平衡二元樹
- 由於實作方便，通常都是使用 `std::set` 配合自定義的直線 `Line`
- 不過在我們要支援的操作中，有兩種二分搜的基準（斜率、有效區間），直接套用 `std::set` 的話可能會有一些小問題
- 一個直覺的處理方法是維護兩個 `std::set`，不過這樣不僅會讓 code 量變很大而且還會讓常數變大，並不是一個實惠的做法。以下介紹兩種處理的方法，讀者可以自行參考使用
  1. 使用一個全域的變數 `flag` 來記錄現在要使用哪一種二分搜的基準，並且在 `Line` 的比較運算子中根據 `flag` 的值來調整內容
  2. 重載 `Line` 的比較運算子：一個與 `Line` 比較（斜率）、一個與整數比較（有效區間）
- 範例程式碼使用第二種方法，並且直接繼承 `std::multiset` 以減少實作負擔
- 值得注意的是，在大部分的問題中查詢都是整數，因此有效

斜率優化 – 例題

題目 (CSES 2085 Monster Game II)

現在你要過 N 個關卡，關卡編號為 $1, 2, \ldots, N$，每一個關卡都有一隻怪獸，第 i 個關卡的怪獸的能力值為 s_i。在還沒進入任何關卡之前，你有一個初始的技能值 x。
在第 $1, 2, \ldots, N-1$ 關卡中，你可以選擇打倒怪獸，或者是直接逃跑。如果你選擇打倒第 i 隻怪獸，你要花 $f \times s_i$ 的時間打倒怪獸，其中 f 是你當前的技能值。在打倒怪獸後，你的新的能力值會變成 f_i。
你的最終目標是打倒第 N 隻關卡的怪獸，請問在這個過程中，你最少需要花費多少時間。

- $N \leq 2 \times 10^5$

斜率優化

- 假設 $f_0 = 0, dp(0) = 0$，應該不難列出以下的 DP 式子：
- $dp(i) = \min_{0 \le j < i}(dp(j) + f_j \times s_i)$
- 為了之後講解方便，我們先把所有的 f_i 設成 $-f_i$，我們就可以把 DP 式子轉換成取最大值的形式：
- $dp(i) = \max_{0 \le j < i}(dp(j) + f_j \times s_i)$
- 並且最後的答案為 $-dp(N)$
- 而跟前面那個例題不同是，這邊的斜率、查詢不再是單調的。因此，就需要使用上面那個技巧

斜率優化 – 用 CDQ 解斜率非單調的斜率優化

- 在上個小章節，我們成功用 $O(N \log N)$ 的時間內，解決了斜率非單調的斜率優化
- 不過，動態凸包其實沒有想像中那麼好寫
- 但是，相對來說，如果斜率、詢問都是單調的話，世界會變得非常的美好，實作的難易度也會下降很多
- 我們有沒有辦法把非單調的東西變成單調呢？
- 答案是肯定的

斜率優化 – 用 CDQ 解斜率非單調的斜率優化

- 考慮以下分治法：
- 如果 $L == R$，結束遞迴
- 否則，依序執行以下步驟：
  - 遞迴計算左半部 $[L, mid]$ 的 DP 值。
  - 用左半部 $[L, mid]$ 算出來的 DP 值，更新右半部 $[mid + 1, R]$ 的 DP 值。
  - 遞迴計算右半部 $[mid + 1, R]$ 的 DP 值。

斜率優化 – 用 CDQ 解斜率非單調的斜率優化

- 應該不難發現，每個 DP 值在計算的過程中，**都會被所有可能的轉移點嘗試轉移**
- 因此，上述演算法的正確性是可以保證的
- 但是，時間複雜度呢？
- 如果左半部 DP 值更新右半部的部份，我們單純使用 $O(N^2)$ 轉移，會讓複雜度變成 $T(N) = 2T(\frac{N}{2}) + O(N^2)$，會使得最後的複雜度為 $O(N^2)$
- 但，「用左半部更新右半部」這個部份，**其實就是個斜率單調、詢問單調的問題**

斜率優化 – 用 CDQ 解斜率非單調的斜率優化

- 因為我們**可以把左半部的線按照斜率排序、把右半部的詢問按照詢問大小排序**
- 可以這麼做的理由是因為，我們已經知道所有左半邊的 DP 值（也就是說，我們已經知道所有的線），同時我們也可以知道所有右半部的詢問
- 因此，我們就可以**自由的決定他們的順序**，得到一個好的順序之後，我們就可以好好的方便做事了！
- 也因此，「用左半部更新右半部」這個部份，我們可以花 $O(N \log N)$ 的時間排序，並且花 $O(N)$ 的時間內解決。因此，整體複雜度就變成 $T(N) = 2T(\frac{N}{2} + O(N \log N))$，得到 $T(N) = O(N \log^2 N)$ 的複雜度
- 可以參考講義上面的範例程式碼～

凹凸単調優化

凹凸單調優化 – 前言

- 從這個章節開始，要介紹更多的 DP 優化
- 這章先介紹常見的 1D / 1D 優化
- 所謂的 xD/yD 的 DP，代表說狀態是 $O(n^x)$，轉移是 $O(n^y)$ 的 DP

凹凸單調優化 – 問題定義

定理 ((1D/1D))

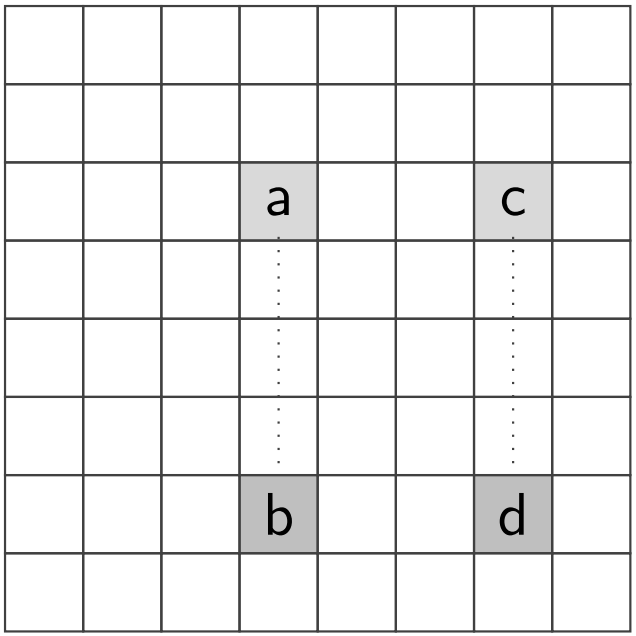
要求的是一個一維代價 $dp[j]$, $\forall\ 1 \leq j \leq n$。給一可在常數時間內求得的轉移代價函數 (transition cost function) $f(i, j, dp[i])$, $\forall\ 0 \leq i < j \leq n$、以及初始化邊界 $dp[0]$，求：

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

接下來為了討論方便，我們把 $f(i, j, dp[i])$ 簡單寫作 $f(i, j)$

不過討論時須注意 $f(i, j)$ 事實上和 $dp[i]$ 有關，因此他是**在線 (online)** 的，也就是 $f(i, j)$ 的值必須在 $dp[i]$ 被計算出來後才能得知

凹凸單調優化 – 單調性



凹單調 (concave totally monotone)：

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

凸單調 (convex totally monotone)：

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

凹凸單調優化 – 單調性

- 假設 $i_1 < i_2, j_1 < j_2$：
- 凹單調：若 $A[i_1][j_1] \leq A[i_2][j_1]$ 則 $A[i_1][j_2] \leq A[i_2][j_2]$
- 凸單調：若 $A[i_1][j_1] \geq A[i_2][j_1]$ 則 $A[i_1][j_2] \geq A[i_2][j_2]$
- 代表最佳選擇具有某種程度的「淘汰性」，也就是某個時間點後一個選擇一旦輸給了某個順序在他之前/後的選擇，他就再也不可能是最佳的選擇

凹凸單調優化 – 凹單調

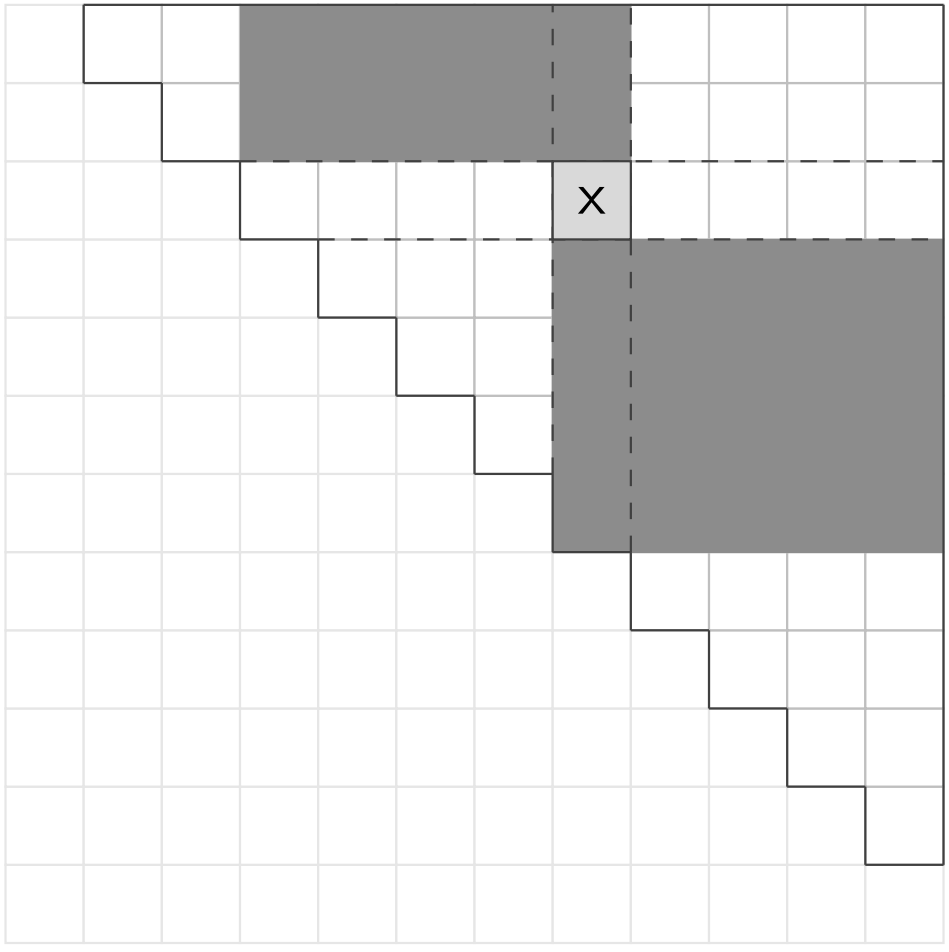


Figure: 凹單調時，被 x 殺死的元素

凹凸單調優化 – 凹單調

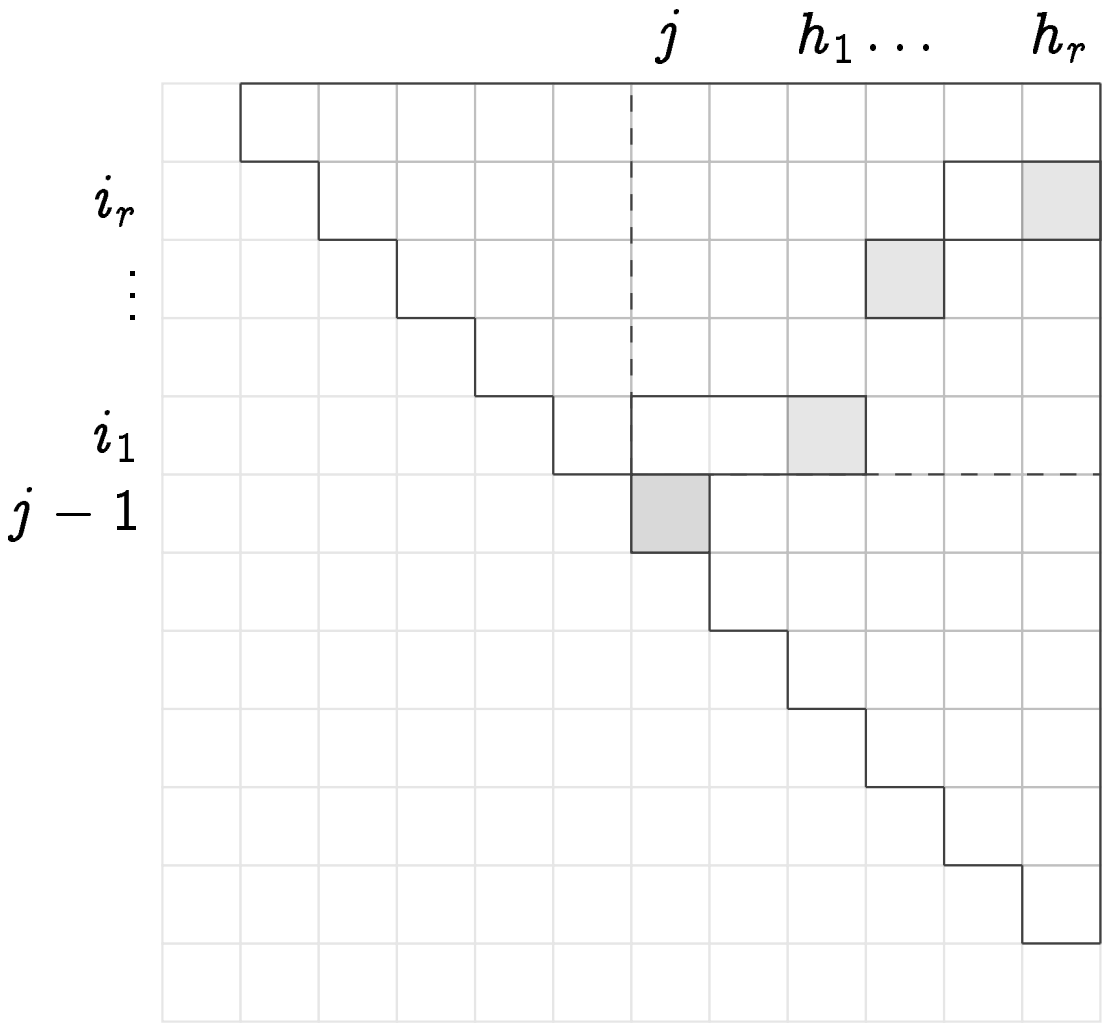


Figure: 凹單調性 DP

凹凸單調優化 – 凹單調

- 用**列區間 (row segment)** 來代表這些人選
- $S = ([i_1, j : h_1], [i_2, h_1 + 1 : h_2], \ldots, [i_k, h_{k-1} + 1 : h_k])$
- 其中 $[i, j_1 : j_2]$ 代表第 i 列上，第 j_1 到第 j_2 行所形成的列區間，h_t 是第 t 個列區間的最右元素

凹凸單調優化 – 凹單調

```
1  stack<Seg> sta;
2  sta.push(Seg(0, 1, m));
3  for (int j = 1; j <= m; ++j) {
4    while (!sta.empty() && sta.top().r < j) sta.pop(); //
     ↪  把過期的最佳解丟掉
5    dp[j] = cal(sta.top().pos, j) - a[j]; // 平常的題目應該是
     ↪  不用 -a[j] 這項的，這邊 -a[j] 是因應題目要求
6
7    while (!sta.empty() && cal(sta.top().pos, sta.top().r)
     ↪  > cal(j, sta.top().r)) {
8      // 把被完全覆蓋的線段刪掉
9      sta.pop();
10   }
11   if (sta.empty()) {
12     sta.push(Seg(j, j + 1, m));
13   }
```

凹凸單調優化 – 凹單調

```
    else {
      Seg seg = sta.top(); sta.pop();
      // 二分搜斷點
      int l = seg.l - 1, r = seg.r;
      while (r - l > 1) {
        int mid = (l + r) >> 1;
        if (cal(seg.pos, mid) >= cal(j, mid)) l = mid;
        else r = mid;
      }
      sta.push(Seg(seg.pos, r, seg.r));
      if (j + 1 <= l) sta.push(Seg(j, j + 1, l));
    }
}
```

凹凸單調優化 – 凸單調

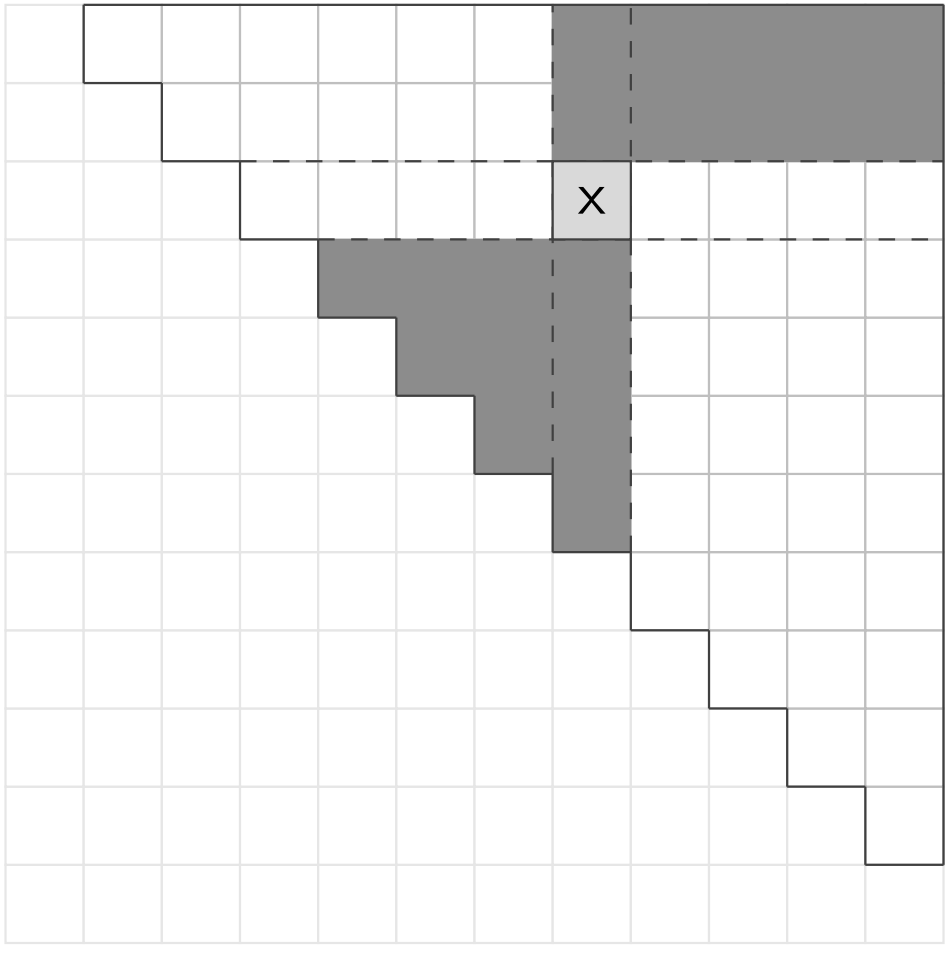


Figure: 凸單調時，被 x 殺死的元素

凹凸單調優化 – 凸單調

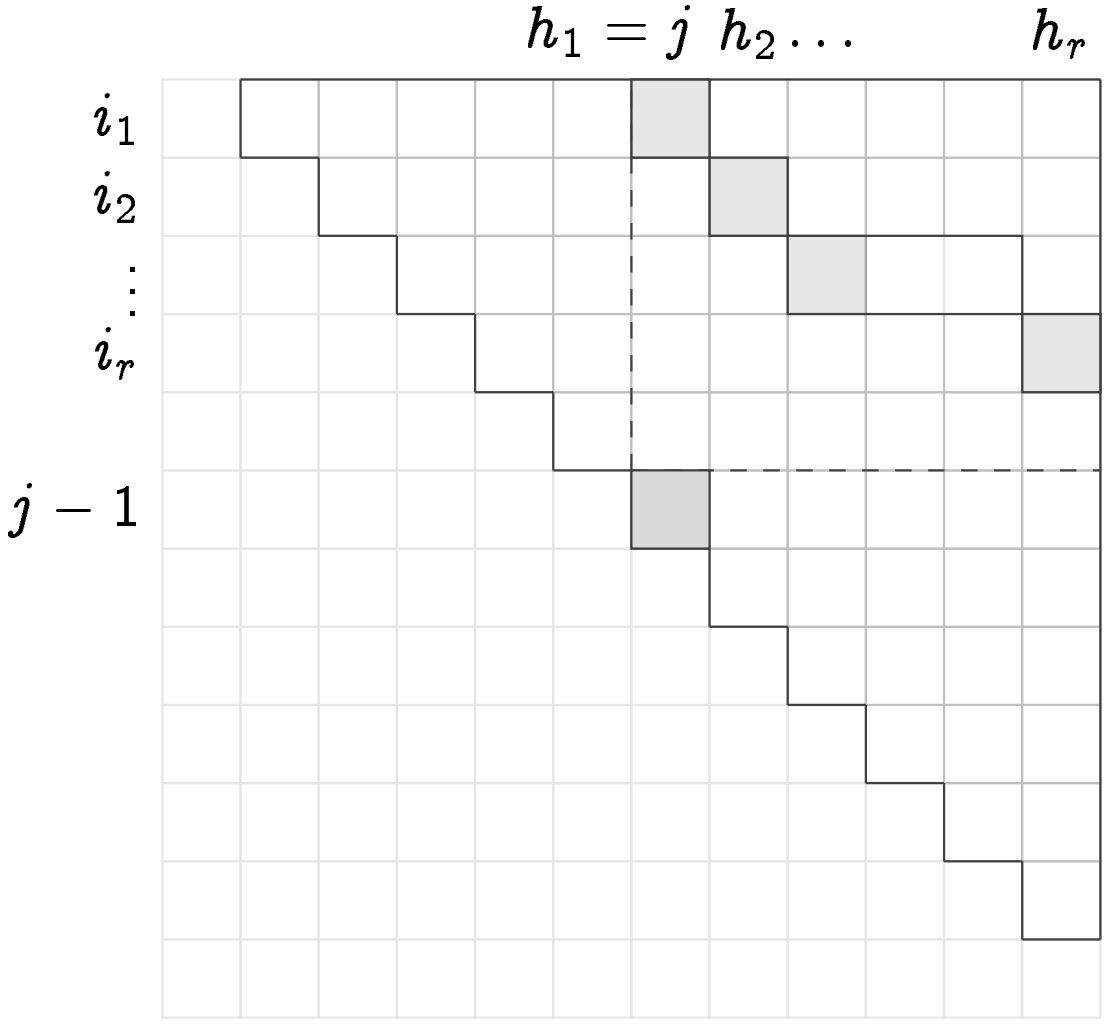


Figure: 凸單調性 DP

凹凸單調優化 – 凸單調

- 用**列區間 (row segment)** 來代表這些人選
- $S = ([i_1, j : h_1], [i_2, h_1 + 1 : h_2], \ldots, [i_k, h_{k-1} + 1 : h_k])$
- 其中 $[i, j_1 : j_2]$ 代表第 i 列上，第 j_1 到第 j_2 行所形成的列區間，h_t 是第 t 個列區間的最右元素

凹凸單調優化 – 凸單調

```
deque<Seg> dq;
dq.push_back(Seg(0, 1, n));
for (int j = 1; j <= n; ++j) {
  while (!dq.empty() && dq[0].r < j) dq.pop_front(); // 把過期的最佳解丟掉
  dp[j] = cal(dq[0].pos, j);
  while (!dq.empty() && cal(dq.back().pos, dq.back().l) > cal(j, dq.back().l)) {
    // 把被完全覆蓋的線段刪掉
    dq.pop_back();
  }
  if (dq.empty()) {
    dq.push_back(Seg(j, j + 1, n));
  }
```

凹凸單調優化 – 凸單調

```
    else {
      Seg seg = dq.back(); dq.pop_back();
      // 二分搜斷點
      int l = seg.l, r = seg.r + 1;
      while (r - l > 1) {
        int mid = (l + r) >> 1;
        if (cal(seg.pos, mid) > cal(j, mid)) r = mid;
        else l = mid;
      }
      dq.push_back(Seg(seg.pos, seg.l, l));
      if (l != n) dq.push_back(Seg(j, l + 1, n));
    }
}
```

凹凸單調優化 – Monge Condition

定理 (Monge condition)

給一個 $m \times n$ 矩陣 B，若 $\forall 1 \leq i_1 < i_2 \leq m, 1 \leq j_1 < j_2 \leq n$ 我們有：

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

那我我們說他符合 convex (concave) Monge condition.

定理 (Monge condition (等價形式))

給一個 $m \times n$ 矩陣 B，若 $\forall 1 \leq i < m, 1 \leq j < n$ 我們有：

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

那我我們說他符合 convex (concave) Monge condition.

凹凸單調優化 – Monge Condition

要注意的是，Monge condition 事實上是比較嚴格的，因此**反方向的推論並不成立**，不符合 Monge condition 也不代表不具有單調性。

Knuth Optimization

Knuth Optimization – 前言

- 前面的章節，幾乎都是介紹 1D / 1D 的 DP 優化
- 現在要來介紹 2D / 1D 相關的優化

Knuth Optimization – 問題定義

定理 ((2D/1D))

要求的是一個二維代價 $dp[i][j], \forall\ 1 \leq i < j \leq n$。給一可在常數時間內求得的轉移代價函數（transition cost function）$w(i,j), \forall\ 1 \leq i < j \leq n$、及初始化邊界 $dp[i][i] = 0, \forall\ 1 \leq i \leq n$；求：

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

Knuth Optimization – 問題定義

- 我們知道 2D/1D 問題有顯然的 $O(N^3)$ 作法
- 亦即對於要求 $dp[i][j]$，枚舉 $1 \leq k < j$ 作為轉移的決策點並找到最小的轉移來源
- 我們稱在這轉移中的最佳選擇 k 為 $K_{i,j} = \text{argmin}_{i \leq k < j} \{dp[i][k] + dp[k+1][j]\}$
- 這邊我們提出一個性質：Optimal Split Point Monotonicity

Knuth Optimization – 重要性質

定理 (Monge condition implies Monotonic Optimal Split Point)

當上述 2D/1D 問題的代價 C 滿足凸 convex Monge condition，亦即：

$$dp[i][j] + dp[i+1][j+1] \geq dp[i][j+1] + dp[i+1][j]$$

則決策點具有單調性：

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

白話來說：$dp[i][j]$ 的決策點界於 $dp[i][j-1]$ 與 $dp[i+1][j]$ 的決策點之間

Knuth Optimization

- 光這麼看好像看不出什麼端睨，$K_{i,j-1}$ 到 $K_{i+1,j}$ 究竟有多大也不清楚
- 不過這時依然可以使用均攤分析！就讓我們一次分析填完一整 (斜) 排 DP 所需要花的時間吧：

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

Knuth Optimization

- 斜著做 DP 的話，每一列要考慮的數量和不超過 N
- 總複雜度為 $O(2N \times N) = O(N^2)$

Knuth Optimization – 範例程式碼

```
1  int kl = K[i][j - 1];
2  int kr = K[i + 1][j];
3  for (int k = kl; k <= kr; ++k) {
4    if (w(i, j) + dp[i][k] + dp[k + 1][j] < dp[i][j]) {
5      dp[i][j] = w(i, j) + dp[i][k] + dp[k + 1][j];
6      K[i][j] = k;
7    }
8  }
```

2D / 1D 凹優化

- 基本上就只是做 N 次 1D / 1D 凹優化
- 非常不常見，這邊就不細講

轉移點單調優化

轉移點單調優化 – 例題

題目 (ZJ a146 Sliding Window)

令 $F(i,j)$ 為一個對所有 $1 \le i \le j \le N$ 的正整數 i,j 都有定義的函數。對於所有的 $1 \le j \le N$，求 $\min_{i \le j} F(i,j)$ 或 $\max_{i \le j} F(i,j)$。

這樣看起來有點抽象，我們來實際看一個例子：

轉移點單調優化 – 例題

題目 (ZJ a146 Sliding Window)

有 N 個位於一維數線上的餐廳，由左至右編為 1 到 N 號，其中 i 號餐廳與 $i-1$ 號餐廳的距離為 A_i。你有 M 張餐卷，第 i 張可以兌換第 i 種食物且只能使用一次。已知這 N 家餐廳都有販售這 M 種食物，且第 i 家餐廳的第 j 種食物的好吃度為 $B_{i,j}$。你可以從任意間餐廳開始，請問「吃到食物的好吃度總和 $-$ 總行走距離」的最大值為何。

- $1 \leq N \leq 5000$
- $1 \leq M \leq 200$

轉移點單調優化 – 例題

- 首先可以知道，當決定了要在哪些餐廳買食物之後，最佳的策略一定是從最左的直接走到最右邊的餐廳，不會來回行走
- 因此，定義 $F(i,j)$ 為從 i 號餐廳開始，結束在 j 號餐廳的最大價值。則 $F(i,j)$ 可以被寫成
- $F(i,j) = \left(\sum_{f=1}^{M} \max_{i \le k \le j} B_{k,f}\right) - \left(\sum_{k=i+1}^{j} A_k\right)$
- 其中 i 與 j 之間各種食物的最大好吃度可以透過 RMQ 預處理，i 與 j 的距離也可以透過前綴和達到 $O(1)$，因此可以在 $O(M)$ 的時間內計算好 $F(i,j)$

轉移點單調優化 – 例題

那該如何有效率的計算 $\max_{i \le j} F(i,j)$，或是更一般的，對所有餐廳 j 求出以它當終點的最大價值呢 V_j？這裡我們必須用到一個性質：

定理 (性質)

令 H_j 為以 j 當終點情況下的最佳起點，亦即：

$$H_j = \arg\max_{i \le j} F(i,j)$$

則 $H_j \le H_{j+1}$。也就是說當終點向右時，最佳的起點只會往右或不動

轉移點單調優化 – 演算法

- 有了這個性質之後，可以考慮下列的分治算法：
- 假設現在我們想要計算所有 $L \le j \le R$ 的 V_j，並且我們知道所有在範圍內的 H_j 都滿足 $L' \le H_j \le R'$
- 令 $M = \lfloor \frac{L+R}{2} \rfloor$，那我們在 $O(R' - L')$ 次計算 F 值之內，枚舉 $[L', \min(R', M)]$ 之間的 i 並找出最大的 $F(i, j)$ 以及對應的 H_M
- 根據單調性，我們知道 $[L, M-1]$ 的最佳起點會落在 $[L', H_M]$ 、$[M+1, R]$ 的最佳起點會落在 $[H_M, R']$。因此我們分別遞迴計算子問題 $\{L, M-1, L', H_M\}$ 及 $\{M+1, R, H_M, R'\}$
- 複雜度為 $O(N \log N \times F())$，其中 F 是算一個 $F(i, j)$ 的值的複雜度

轉移點單調優化 – 演算法

```
1  void dc(int L, int R, int best_L, int best_R) {
2    if (L > R) return;
3    int mid = (L + R) >> 1;
4    for (int i = best_L; i <= best_R; ++i) {
5      if (h[mid] == 0 || f(mid, i) > f(mid, h[mid]))
       ↪  h[mid] = i;
6    }
7    dc(L, mid - 1, best_L, h[mid]);
8    dc(mid + 1, R, h[mid], best_R);
9  }
```

轉移點單調優化 – CDQ 配轉移點單調

- 有了轉移點單調這個好東西後，我們可以回頭看一次 1D / 1D 凸單調
- 應該不難發現，**1D / 1D 凸單調也是具有轉移點單調**這個性質的，但是在有些時候，**轉移來源必須是在線的**，導致我們無法快樂直接套用轉移點單調
- 但其實，這個世界並沒有這麼不美好。這個問題可以用類似斜率優化 CDQ 的思維，來多花一點時間解決！假設我們現在正在算 $[L, R]$ 的答案，算法過程如下：

轉移點單調優化 – CDQ 配轉移點單調

- 如果 L $==$ R，結束遞迴
- 否則，依序執行以下步驟：
  - 遞迴計算左半部 $[L, mid]$ 的 DP 值。
  - 用左半部 $[L, mid]$ 算出來的 DP 值，更新右半部 $[mid + 1, R]$ 的 DP 值。
  - 遞迴計算右半部的 $[mid + 1, R]$ DP 值。
- 複雜度為 $O(N \log^2 N)$
- 範例 code 麻煩參考講義～

轉移點單調優化 – 基於轉移點單調的唬爛

- 使用這個方法之前，要先決定一個魔法常數 $magic$，接著，該方法的執行過程如下：
- 假設現在在計算 $dp(i)$，並已知 $dp(i-1)$ 的轉移點為 p
- 嘗試從 $[p, p+magic]$ 轉移到 $dp(i)$，如果發現一個比 p 好的點，把 p 更新成新的轉移點，並重新執行這部份
- 嘗試從 $[i-magic, i]$ 轉移到 $dp(i)$，如果發現一個比 p 好的點，就更新 p
- 複雜度為 $O(magic \times N)$，**但是並沒有保證正確性！**
- 但是不失為一個**唬爛好方法**

SMAWK 演算法

SMAWK 演算法 – 前言

- 前面的轉移點單調，已經讓複雜度從 $O(N^2)$ 的東西壓到 $O(N \log N)$
- 現在，要來進一步的壓到 $O(N)$

SMAWK 演算法 – 定義

定理

定義一個 2×2 矩陣 $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ 是單調的，如果以下兩個條件都有成立：

- 如果 $c < d$，那麼 $a < b$
- 如果 $c = d$，那麼 $a \leq b$

而一個 $N \times M$ 的矩陣是 **完全單調矩陣**，若且唯若每個 2×2 的子矩陣（submatrix）都是單調矩陣。注意到子矩陣的列（row），行（column）不一定要是連續的。

定義 $h(i)$ 是第 i 列中，最左邊最小值發生的位置。而如果 $N \times M$ 的矩陣 A 是一個完全單調矩陣，那麼會有 $h(1) \leq h(2) \leq \cdots \leq h(N)$ 這個性質

SMAWK 演算法

- 而 SMAWK 演算法做的事情，就是給定一個 $N \times M$ 大小的完全單調矩陣，SMAWK 演算法會在 $O(N+M)$ 的時間內，找到 $h(1), h(2), \ldots, h(N)$
- 這個演算法分成的架構如下：
  - 若 $\max(N, M) \leq 2$，暴力算出所有的 $h(i)$。
  - 若 $N \geq M$，則呼叫 `interpolate()` 算法，遞迴計算 $\frac{N}{2} \times M$ 矩陣的答案後，找出剩下 $\frac{N}{2}$ 列的答案。
  - 若 $N < M$，則呼叫 `reduce()` 算法，把一些不重要的行 (column) 刪掉後，遞迴呼叫剩下 $N \times N$ 矩陣的答案。

SMAWK 演算法 – interpolate

- interpolate 的想法很簡單：把 $N \times M$ 的矩陣分成奇數列與偶數列，把偶數列構成的矩陣拿去遞迴算答案
- 當我們有偶數列的答案後，我們就可以根據偶數列的答案，算出奇數列的答案
- 根據前面提到的 $h(1) \leq h(2) \leq \cdots \leq h(N)$ 的性質，假設我要算奇數列 $h(i)$ 的答案，我們可以發現：$h(i-1) \leq h(i) \leq h(i+1)$。也就是說，這個奇數列答案的位置，**是被前後兩個偶數列的答案位置限制住的**。所以，我們只要跑過 $h(i-1)$ 到 $h(i+1)$ 的所有值，就可以知道第 i 行的最小值發生的位置
- 也因此，我們就可以花 $O(N+M)$ 加上額外遞迴的時間，完成 interpolate 這個操作。

SMAWK 演算法 – reduce

- reduce 這個函數的想法，就是利用矩陣是完全單調的性質，刪掉至少 $M - N$ 個列，讓新的矩陣變成至多 $N \times N$ 後，拿去跑 interpolate
- 基本上，會開一個 stack S 來維護要選擇的列。接著，我們會由左而右的看每個列的狀況。假設我們正在看列 C，而矩陣的第 i 行第 j 列的值為 $A_{i,j}$。演算法進行過程如下：
- 如果 stack 是空的，直接把 C 放進去 stack 裡面。
- 否則，執行以下迴圈直到 S 是空的，或者是 C 被擊敗了：
  - 假設 stack 裡面有 x 個列，最後一個為 c_x。
  - 如果 $A_{x,c_x} > A_{x,C}$，那麼把 c_x 這列從 stack 中丟掉，因為 c_x 不可能包含最佳解了。
  - 否則，代表 C 被擊敗了，離開迴圈。
- 如果這時候 stack 裡面的列數量小於 N 個，那麼就把 C 丟進 stack 裡面。

SMAWK 演算法 – 範例

10	20	13	19	35
20	29	21	25	37
28	33	24	28	40
42	44	35	38	48
48	49	39	42	48
56	55	44	44	49
75	73	59	57	53

SMAWK 演算法 – 範例

20	29	21	25	37
42	44	35	38	48
56	55	44	44	49

SMAWK 演算法 – 範例

20		21	25	
42		**35**	38	
56		**44**	44	

SMAWK 演算法 – 範例

10	20	13	19	35
20	29	21	25	37
28	33	24	28	40
42	44	**35**	38	48
48	49	39	42	48
56	55	**44**	44	49
75	73	59	57	53

SMAWK 演算法 – 範例

10	20	13	19	35
20	29	21	25	37
28	33	**24**	28	40
42	44	**35**	38	48
48	49	**39**	42	48
56	55	**44**	44	49
75	73	59	57	**53**

SMAWK 演算法 – 時間複雜度

- 假設我們拿到一個 $N \times M$ 的矩陣，可以發現大致上需要按照 reduce, interpolate, reduce, interpolate 的順序來執行 SMAWK 演算法
- 遞迴式為 $T(N, M) = O(N + M) + T(\frac{N}{2}, N)$
- 展開之後可以發現 $T(N, M) = O(M) + 2(O(N) + O(\frac{N}{2}) + O(\frac{N}{4}) + \dots)$，可以發現這是一個 $O(N + M)$ 複雜度的演算法

SMAWK 演算法 – 應用

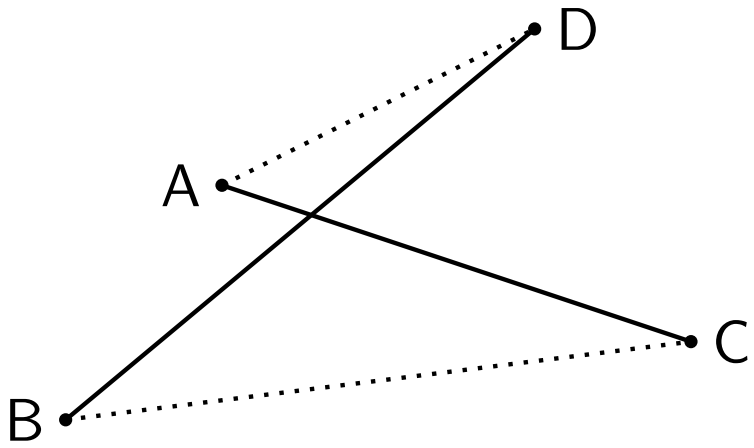
- 注意到完全單調矩陣的證明，有的時候要直接證很難。但是，別忘記在前面章節提到的，**如果那個矩陣符合 monge condition，那麼那個矩陣也會是完全單調矩陣**
- 讓我們來看一個例題，底下的例題同時也是 SMAWK 演算法被提出的論文中，論文作者提出來的題目

題目

給定一個 N 個點的凸多邊形。對於凸多邊形上的每個點 i，請求出離點 i 最遠的點的編號。

- $N \leq 5 \times 10^5$

SMAWK 演算法 – 應用



假設 A, B, C, D 這四個點是凸多邊形上按照逆時針順序的四個點。藉由觀察上圖，可以發現
$dis(A,C)+dis(B,D) \geq dis(A,D)+dis(B,C)$。而如果假設那四個點分別是凸多邊形上的第 x_1, x_2, y_1, y_2 個點，就可以把式子寫成 $dis(x_1,y_1)+dis(x_2,y_2) \geq dis(x_2,y_1)+dis(x_1,y_2)$。即可發現這題有「Monge Condition」這個性質

Aliens 優化

Aliens 優化 – 前言

- Aliens 優化又稱 WQS 二分搜，據說在很久之前就在大陸被發明了
- 不過使它發揚光大的是 IOI 2016 一道名為 Aliens 的題目
- 近幾年基於一些不明的原因開始在臺灣競程界流行了起來，所以在這邊簡單的介紹一下

Aliens 優化 – 例題

題目

給定一棵 N 個節點的帶權樹以及一個常數 K，請找出 K 條邊使得邊權總和最大且不存在兩條邊有共同頂點。

- $N \leq 2.5 \times 10^5$
- $K \leq N - 1$

Aliens 優化 – 例題

先考慮沒有 K 這個限制下要怎麼處理，也就是說找出一些邊使得邊權總和最大且沒有共端點

這是一個經典在樹上 DP 的題目：令（為了方便，定義一個點被選了就是有一條相鄰的邊被選了）

$dp(v, 0) :=$ v 的子樹在 v 還沒有被選的情況下的最大答案

$dp(v, 1) :=$ v 的子樹在 v 被選的情況下的最大答案

Aliens 優化 – 例題

- $dp(v, 0) = \sum_{(u,w)\in G(v)} \max(dp(u, 0), dp(u, 1))$
- $dp(v, 1) = \max_{(u,w)\in G(v)}\{dp(u, 0) + w + \sum_{(u',w')\in G(v),u'\neq u} \max(dp(u', 0), dp(u', 1))\}$
- 其中式子一代表不取 v 向下延伸的邊的 case，所以並不在乎小孩有沒有被選到。而式子二則是枚舉要取哪條邊，所以對應的小孩一定不能被選到，而其他則無所謂。透過一些預處理可以使得這個 DP 算法的複雜度壓到 $O(N)$
- 那要怎麼處理有限制個數的狀況呢？其中一個方法是在 DP 狀態多記一個維度代表取了幾條邊，不過會使得複雜度變成至少 $O(NK)$，並不是一個足夠有效率的方法

Aliens 優化 – 性質

- 我們需要以下的性質來幫助解題

定理 (性質)

令 $f(K)$ 為恰取 K 條邊的最大邊權總和，則 f 是一個凹函數 (concave function) [1]。亦即，這個函數滿足 $f(K+1) - f(K) \leq f(K) - f(K-1)$。

- 也就是 f 的差分遞減。再白話一點的話，就是取第 $K+1$ 條邊的**淨獲益**不會比取第 K 條邊的淨獲益來得高
- 有了這個性質以後，原本的問題就可以被轉為下面的問題

[1] 為了方便說明，這邊指的 concave function 都只定義在整數上面。

Aliens 優化 – 性質

題目

等價的問題 現在有一個凹函數 f，我們想知道 $f(K)$ 的值是多少。可惜的是，沒有好的演算法可以直接計算 $f(K)$，可以知道的只有給定 p，$f(x) - px$ 的最大值以及最大值發生的位置。要如何運用這些資訊求出 $f(K)$ 呢？

令 $M(p)$ 為 $f(x) - px$ 的最大值，$V(p)$ 為最大值發生的位置，若有多個最大值，則定義為 x 值最小的那個最大值。不難發現，假設現在我們幸運的找到了一個 p，使得 $V(p) = K$，那麼 $f(K)$ 就可以直接用 $f(K) = M(p) + pK$ 算出來了，所以我們的目標就是快速找到這樣的 p

Aliens 優化 – 性質

- 不妨把 $f(x)$ 的圖形在二維平面上畫出來
- 雖然我們不能直接知道 $f(x)$ 具體長怎樣，不過由於我們已經知道 f 是凹函數，所以可以考慮任意的凹函數
- 觀察一下後可以知道，f 的長相會長得像一個**上凸包**，且 $V(p) = x$ 就代表存在一條斜率為 p 的直線與 f 的圖形相切於 $(x, f(x))$
- 原因是一條斜率為 p 的直線切在 x 上就代表通過 x 這點且斜率為 p 的直線的截距是所有點中最大的
- 令切線為 $g(x) = px + b$，則帶入 $(x, f(x))$ 後得到截距 $b = f(x) - px$，恰好與 $M(p)$ 的定義吻合
- 不僅如此，當 x 越大時，切在 $(x, f(x))$ 的切線斜率是遞減的。這就告訴我們其實我們可以透過二分搜 p 來找到 $V(p) = K$，再藉此推算 $f(K)$

Aliens 優化 – 性質

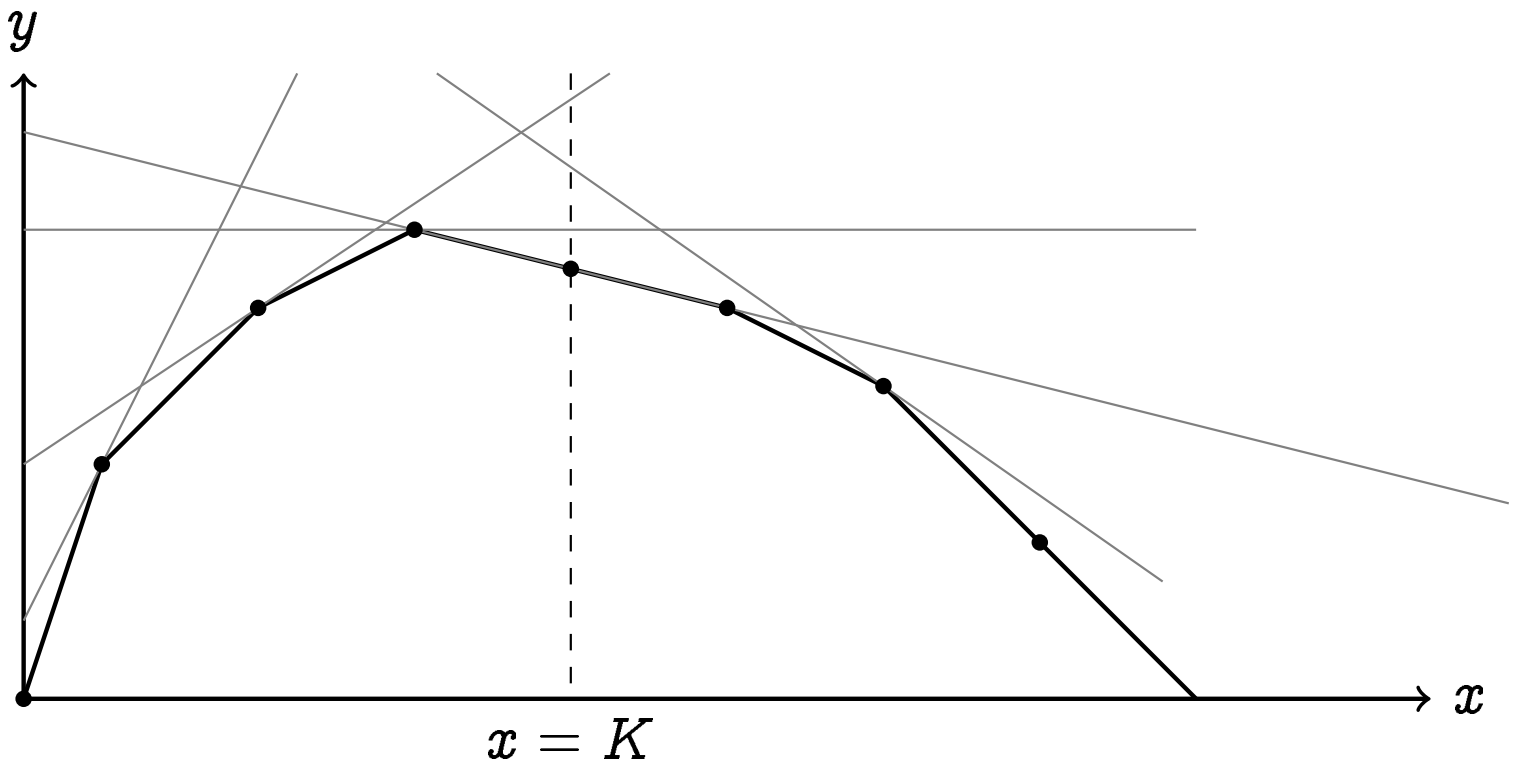


Figure: 一個可能的 $f(x)$ 以及切線們。注意到切線斜率隨著 x 遞增而遞減，且 $f(x)$ 在 $x = K$ 之處等差。

Aliens 優化 – 性質

- 這邊其實有一個問題，那就是我們可以透過二分搜找到最小的 p 使得 $V(p) \leq k$，但這不保證就真的有那麼一個 p 使得 $V(p) = k$
- 不過這件事只會發生在當 f 在 $K \in [V(p), V(p+1)]$ 之間是等差的情形
- 所以一個解決方法是分別求出 $f(V(p))$ 與 $f(V(p+1))$ 之後內插得到 $f(K)$，也就是說
  $f(K) = f(V(p)) + \frac{f(V(p+1)) - f(V(p))}{V(p+1) - V(p)} (K - V(p))$

Aliens 優化 – 性質

- 另一種不需要內插的做法用到了 $V(p)$ 定義為最小的最大值發生點以及我們找到的是最小滿足條件的 p，在這個情況下，由於 $f(x) - px$ 的頂部是平的，所以我們會有 $f(V(p)) - pV(p) = M(p) = f(K) - pK \implies f(K) = M(p) + pK$
- 因此只要找到 $M(p)$ 之後便可以直接找出 $f(K)$ 的值
- 一般來說，遇到的題目大部分的 f 都是整數到整數的函數，在這個情況底下 f 形成的凸包的斜率也都會是整數，所以上面講的二分搜以及內插全部都可以在整數上完成，比起使用浮點數計算，在效率以及誤差上會有很大的提升
- 另外，這邊並沒有講到合理的二分搜上下界要定在哪，這部分請讀者自行推導一下

Aliens 優化 – 性質

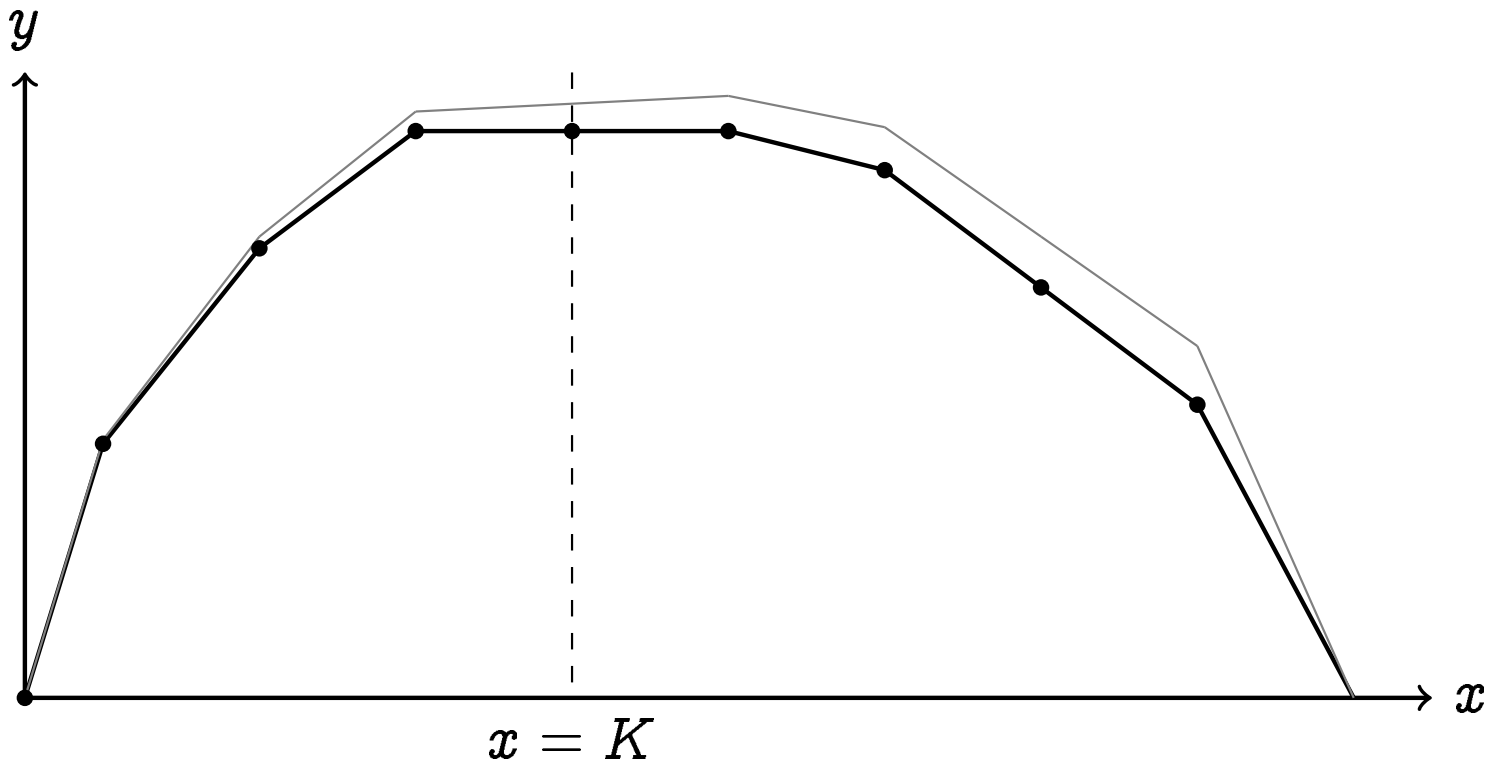


Figure: $f(x) - px$ 的圖形，其中 p 為最小的 p 使得 $V(p) \leq K$。$x = K$ 之處變為平的。注意到如果二分搜的方向反了（找最大的 p 使得 $V(p) \geq K$ ）且 $V(p)$ 的定義不變的話，$f(x) - px$ 會變成淺色的樣子，$x = K$ 的地方就不是平的，直接推 $f(K)$ 便會出錯。

Aliens 優化 – 性質

- 回到原題，由於我們已經證明了這題的 $f(x)$ 是凹函數，所以可以套用 Aliens 優化
- 而一開始提到的 DP 可以很輕易的被修改成計算 $f(x) - px$ 最大值的形式，只要在取邊相對應的轉移式那邊多扣 p 即可
- 至此，我們已經把原本看似高維的問題轉為了一維的問題，而這也是 Aliens 優化最常被使用的地方
- 最後要提到的是，大部分的情況下，題目中函數的凹性證明並不會像例題那樣簡單
- 據說 IOI 官方也沒有給 Aliens 的證明。所以大部分的時候都是靠感覺，或是本機寫暴力對拍跑跑看小 case 來確認性質
- 另外，由於凸函數（convex function）就是凹函數取負，所以上述的所有推導在差分遞增的情況也都成立，修改細節留給讀者自行練習

感謝大家的聆聽～