Exam Preparation

- Variables
  - w: 需要準備的剩餘資料量
  - n: 有 n 天可以準備
  - a: 每次讀書，剩餘資料量- a
  - b: 連續 x 天讀書，剩餘資料量+b (第一、二天選擇讀書: w = w – 2a + b + 2b)
  - d[i]: 第 i 天選擇睡覺，剩餘資料量+ d[i]
- dp table:
  - dp[i][0]表示第 i 天選擇睡覺的最好情況(最小剩餘量);

    dp[i][1]代表第 i 天選擇讀書的最好情況。
  - dp[i][0] = min(dp[i-1][0], dp[i-1][1]) + d[i] 由於選擇睡覺的最小剩餘量只跟前一天的選擇有關，因此 dp[i][0]會是前一天的最小剩餘量+d[i]
  - dp[i][1] = min(dp[i][1], dp[j][0] + (i – j)(i – j + 1)/2 * b – (i – j) * a) for j belongs to [0, i – 1]. 因為連續讀書會等差級數增加剩餘量，因此最好情況不會只跟前一天有關，min 的後項表示從第 j 天睡覺、在第 j+1 天~第 i 天讀書
  - 邊界為 dp[0][0] = dp[1][0] = w, 因為在第 0 天所以甚麼都還沒做(剩餘量 = 初始量 = w)

- dp 斜率優化

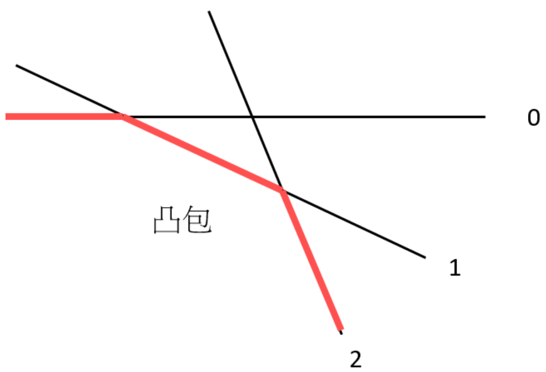
dp[i][1] = min(dp[i][1], dp[j][0] + (i – j)(i – j + 1)/2 * b – (i – j) * a) for j belongs to [0, i – 1]. 這裡我們 min 的前項 dp[i][1]主要是為了記下算過最小值，也就是我們想要的是 dp[j][0] + (i – j)(i – j + 1)/2 * b – (i – j) * a 的最小值。

dp[j][0] + (i – j)(i – j + 1)/2 * b – (i – j) * a
= dp[j][0] + 1/2 * b * (j^2 - j) + ja – ijb + 1/2 * b * (i^2 + i) – ia

其中綠色部分只跟 i 有關，也就是迴圈跑到 i 時 O(1)可算完的東西，視為常數。藍色部分只跟 j 有關，是在迴圈跑到 i 前就可算完的東西，視為常數。紅色部分可以發覺隨著 i 變大，這個值會越來越小(單調遞減)。因此把紅色部分跟藍色部分一起看，可以當成是個以 i 為未知數的直線方程式。

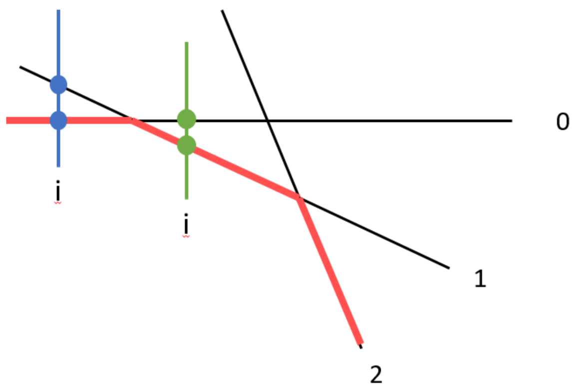
y = mx + b’
x = i
m = -jb
b’ = dp[j][0] + 1/2 * b * (j^2 – j) + ja

如果我們把它想成是一條條直線(數字為加入順序)

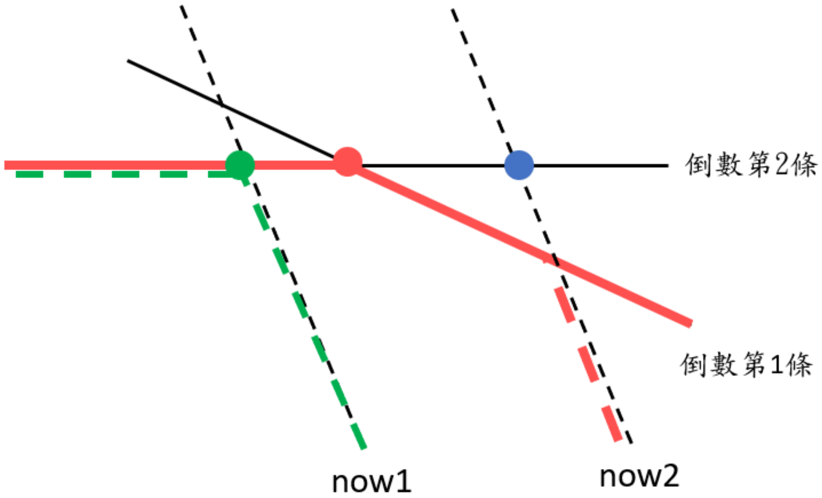


由於我們要的是最小值，因此得到一個凸包，我們用以下 2 種方式來維護凸包性值。

(1) 由圖可知 i 一開始可以由代入第 0 條線求出 dp[i][1], 但當 i 大於某個值後由於第 1 條線的交點比第 0 條線的交點值還要小，因此之後都不會用到第 0 條線(單調遞減)，因此可以把不需要的線移除，這樣 dp[i][1]的值永遠都能從第 0 條線轉移過來。(檢查 i 在第 1 條線和第 0 條線的值，如果第 1 條線的值比較小，則移除第 0 條線，直到第 0 條線的值為現在的最小值為止)



(2) 如圖，我們現在要加入代表 i 的直線(給之後的更大的 i 使用)(以虛線表示)，如果是 now1 的情況，會發現我們根本不需要倒數第一條線就能形成凸包(綠色在紅色左邊)，而 now2 的情況則會保留倒數第一條線(藍色在紅色右邊)，因此可以得知在我們要加入新線條前要先檢查倒數第二條線與倒數第一條線的交點是否在倒數第二條與我們現在要加入這條線的交點的右邊，如果是就要刪除倒數第一條線，直到現在只有 1 條線或是倒數第二條線與倒數第一條線的交點在倒數第二條與我們現在要加入這條線的交點的左邊為止。



- Pseudo code:

```
 Struct line:
        m <- 0
        b <- 0
// substituting i into line l
// for conveience, we also compute i part(green) inside
 func foo(line l, i, a, b): val , val is long long
        return l.m * i + l.b + b * i * (i + 1) / 2 - a * i

 func vertex(line1, line2): x, x is double
        return (line2.b – line1.b) / (line1.m / line2.m) // the intersection
of l1 and l2

func main:
 declare a deque dq
 dp[0][0] <- w
 dp[0][1] <- w
 for i (1 to n) do
        dp[i][0] <- min(dp[i-1][0], dp[i-1][1]) + d[i]
        // step 1
        while dq.size > 1 and foo(dq[1], i, a, b) < foo(dq[0], i, a, b) do
                dq.pop_front
        end while
```

```
        dp[i][1] <- foo(dq[0], i, a, b)
        // step 2
        Let tmp be line(-ib, dp[i][0] + 1/2 * b*(i^2 – i) + ia)
        While dq.size >= 2 and vertex(dq[dq.size-1], dq[dq.size-2]) >
vertex(tmp, dq[dq.size – 2]) do
            dq.pop_back
        end while
        dq.push_back(tmp)
  end for
  print min(dp[n][0], dp[n][1]) // answer
```

- 複雜度（原本的）
  - dp table reset 花 O(n)
  - dp 迴圈部分花 T(n) = 1 + 2 + 3 + 4 +… + n = O(n^2)
  - 時間複雜度 = O(n^2)
- 複雜度(斜率優化後）
  - 由於 while 迴圈裡的動作是在刪除線，而我們最多加進去 n 條線，也就是整個過程中這 while 迴圈不會跑超過 n 次，如果他平均分散在每次更新 i 值的時候，時間複雜度為 O(1)，如果集中在某一次 n(更新 i) + n - 2(while) = O(n)仍不超過 O(n)，且更新 dp[i][0], dp[i][1]本身迴圈就是 O(n)，因此 Time complexity = O(n)