動態規劃（一） Dynamic Programming I

陳俊安 Colten

2022 新化高中 x 嘉義高中 x 薇閣高中 資研社暑期培訓營隊

2022.07.08

前言

- 這一個單元對大家來說非常重要，相信是一個大家第一次接觸到會卡關的東西
- 希望能夠藉由這堂課讓大家認識動態規劃，也期許大家能學到很多，那我們就開始吧

什麼是動態規劃

什麼是動態規劃

- 一個把陣列名字叫做 dp 的技巧
- 學會了嗎？那我們下課吧

什麼是動態規劃

- 用簡單的兩句話來說
- 長江後浪催前浪，一替新人換舊人
- 資之深，則取之左右逢其原

什麼是動態規劃

- 動態規劃問題常常用於解決最佳子結構問題（或算排列組合）上
- 核心的理念是，用先前的最佳狀態來得到當前的最佳狀態，或是用先前的數量轉移出當前的數量

動態規劃的實作

動態規劃的實作

- Top-Down（通常使用遞迴）
- Bottom-Up（迴圈即可，此作法也是最常見的一種作法，且常數也比較小）

Top-Down

Top-Down

- 從上面往下去推需要計算的東西，在底層算完答案後再推回頂部

Ten Point Round Season Contest Spring pB. 奇怪的機器

現在有一個機器，如果在這一台機器放入了長度 n 的序列 a，這台機器會複製出兩組一樣的 a 出來，在這邊我們稱之為 b, c，接下來會將 b 的每一個元素都 $+1$，c 的每一個元素都 $+2$，並以 abc 這樣的順序給你一個新的長度為 $3n$ 的序列，一開始你只有一個序列 0，你會把一直重複把序列丟到機器裡面去，直到得到的序列長度超過 10^{100} 為止，現在有 q $(1 \leq q \leq 2 \times 10^5)$ 組查詢，每組查詢將查詢最後序列的第 k_i $(1 \leq k_i \leq 10^{18})$ 項的值會是誰

Ten Point Round Season Contest Spring pB. 奇怪的機器

現在有一個機器，如果在這一台機器放入了長度 n 的序列 a，這台機器會複製出兩組一樣的 a 出來，在這邊我們稱之為 b, c，接下來會將 b 的每一個元素都 $+1$，c 的每一個元素都 $+2$，並以 abc 這樣的順序給你一個新的長度為 $3n$ 的序列，一開始你只有一個序列 0，你會把一直重複把序列丟到機器裡面去，直到得到的序列長度超過 10^{100} 為止，現在有 q $(1 \leq q \leq 2 \times 10^5)$ 組查詢，每組查詢將查詢最後序列的第 k_i $(1 \leq k_i \leq 10^{18})$ 項的值會是誰

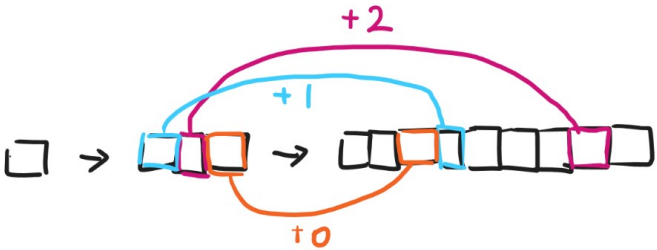
- 我們可以發現每一次序列長度都會多 3 倍，所以長度會以 3 的冪次增長
- 那我們可以先想想看，如果查詢的東西在第 k_i 項，那麼在什麼時候我們就可以知道這一項的答案了

Ten Point Round Season Contest Spring pB. 奇怪的機器

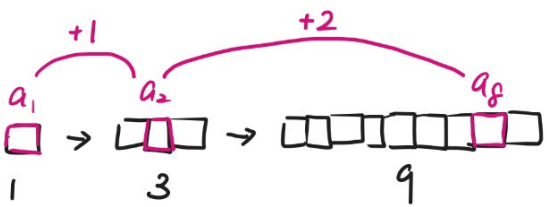
現在有一個機器，如果在這一台機器放入了長度 n 的序列 a，這台機器會複製出兩組一樣的 a 出來，在這邊我們稱之為 b, c，接下來會將 b 的每一個元素都 $+1$，c 的每一個元素都 $+2$，並以 abc 這樣的順序給你一個新的長度為 $3n$ 的序列，一開始你只有一個序列 0，你會把一直重複把序列丟到機器裡面去，直到得到的序列長度超過 10^{100} 為止，現在有 q $(1 \le q \le 2 \times 10^5)$ 組查詢，每組查詢將查詢最後序列的第 k_i $(1 \le k_i \le 10^{18})$ 項的值會是誰

- 我們可以發現每一次序列長度都會多 3 倍，所以長度會以 3 的冪次增長
- 我們可以先想想看，如果查詢的東西在第 k_i 項，那麼在什麼時候我們就可以知道這一項的答案了
- 當序列長度 $\ge k_i$ 的時候我們就可以確定第 k_i 項的值了，換句話說，假設 3^t 是第一個 $\ge k_i$ 的數字，那麼在序列長度變成 3^t 時，我們就可以知道第 k_i 項是誰了

觀察轉移關係



確定轉移式



$$\begin{cases} a_8 = a_2 + 2 \\ a_2 = a_1 + 1 \\ a_1 = 0 \end{cases}$$

統整

- 因此一開始我們只要確定第 k_i 項可以從長度多少的序列得到（也就是我們要找到 t，這個 t 是第一個會使 $3^t \geq k_i$ 的數字）

統整

- 因此一開始我們只要確定第 k_i 項可以從長度多少的序列得到（也就是我們要找到 t，這個 t 是第一個會使 $3^t \geq k_i$ 的數字）
- 接下來確定在當前這一項是從之前的哪一項轉移過來，並確定之前的那一項是前 $\frac{1}{3}$，還是中間，還是後 $\frac{1}{3}$，我們就可以確定第 k_i 項是從前面的那一項 $+0, +1$ 還是 $+2$ 轉移過來的了

統整

- 因此一開始我們只要確定第 k_i 項可以從長度多少的序列得到（也就是我們要找到 t，這個 t 是第一個會使 $3^t \geq k_i$ 的數字）
- 接下來確定在當前這一項是從之前的哪一項轉移過來，並確定之前的那一項是前 $\frac{1}{3}$，還是中間，還是後 $\frac{1}{3}$，我們就可以確定第 k_i 項是從前面的那一項 $+0, +1$ 還是 $+2$ 轉移過來的了
- 接下來我們要知道前一項是多少才可以，因此就遞迴去算前一項，並重複上面的步驟，最後就可以推回第 k_i 項了，這樣的作法就是 Top-Down 的概念

關鍵程式碼

```
 7 int solve(int n)
 8 {
 9
10     if( n == 1 ) return 0;
11
12     int total = 1 , u = n;
13
14     while( total < n ) total *= 3;
15
16     int group = total / 3;
17
18     int prev = n % ( total / 3 );
19
20     if( n % ( total / 3 ) == 0 ) prev = total / 3;
21
22     if( n > group * 2 ) return solve( prev ) + 2;
23
24     else if( n > group ) return solve( prev ) + 1;
25
26     else return solve( prev ) ;
27 }
```

Bottom-Up

Bottom-Up

- 從底下往上推答案的做法，也是動態規劃最常見的一種做法，比起 Top-Down，Bottom Up 的時間複雜度常數小了非常多

轉移式

轉移式

- 動態規劃最重要的元素
- 就像是一個公式一樣，可以想像成我們必須定義出正確的公式才能正確的算出答案

轉移式

- 動態規劃最重要的元素
- 就像是一個公式一樣，可以想像成我們必須定義出正確的公式才能正確的算出答案
- 設計轉移式前，我們必須先定義轉移式，正確的定義轉移式表示的狀態我們才能設計出正確的轉移式（例如： 定義 `dp[i]` 表示前 `i` 項的最大值，這就是定義狀態的例子）

轉移式

- 動態規劃最重要的元素
- 就像是一個公式一樣，要可以想像成我們必須定義出正確的公式才能正確的算出答案
- 設計轉移式前，我們必須先定義轉移式，正確的定義轉移式表示的狀態我們才能設計出正確的轉移式（例如： 定義 `dp[i]` 表示前 `i` 項的最大值，這就是定義狀態的例子）
- 直接來看題目大家應該就能更了解了

Atcoder DP Contest pA. Frog

現在有 N $(1 \leq N \leq 10^5)$ 個石頭，有一隻青蛙一開始在第一個石頭上，每一次青蛙可以跳一格或兩格，假設青蛙從 i 跳到了 k，那麼所需要的花費是 $|a_i - a_k|$，請問當青蛙跳到第 N 個石頭的最小花費

動態規劃初探

Atcoder DP Contest pA. Frog

現在有 N $(1 \leq N \leq 10^5)$ 個石頭，有一隻青蛙一開始在第一個石頭上，每一次青蛙可以跳一格或兩格，假設青蛙從 i 跳到了 k，那麼所需要的花費是 $|a_i - a_k|$，請問當青蛙跳到第 N 個石頭的最小花費

- 這次我先來帶大家完整的做一次動態規劃的流程吧

動態規劃初探

Atcoder DP Contest pA. Frog

現在有 N $(1 \leq N \leq 10^5)$ 個石頭，有一隻青蛙一開始在第一個石頭上，每一次青蛙可以跳一格或兩格，假設青蛙從 i 跳到了 k，那麼所需要的花費是 $|a_i - a_k|$，請問當青蛙跳到第 N 個石頭的最小花費

- 這次我先來帶大家完整的做一次動態規劃的流程吧
- 如果我們定義 `dp[i]` 表示青蛙跳到 i 的最小花費

Atcoder DP Contest pA. Frog

現在有 N $(1 \leq N \leq 10^5)$ 個石頭，有一隻青蛙一開始在第一個石頭上，每一次青蛙可以跳一格或兩格，假設青蛙從 i 跳到了 k，那麼所需要的花費是 $|a_i - a_k|$，請問當青蛙跳到第 N 個石頭的最小花費

- 這次我先來帶大家完整的做一次動態規劃的流程吧
- 如果我們定義 $dp[i]$ 表示青蛙跳到 i 的最小花費
- 由於青蛙每一次都只能跳一格或兩格，所以我們如果現在在第 i 格，那我們一定是從第 $i-1$ 格或第 $i-2$ 格跳過來的，因此我們就可以得到轉移式
- $dp[i] = max(dp[i-2] + |a_i - a_{i-2}|, dp[i-1] + |a_i - a_{i-1}|)$

動態規劃初探

- **轉移式：** $dp[i] = max(dp[i-2] + |a_i - a_{i-2}|, dp[i-1] + |a_i - a_{i-1}|)$
- **一開始我們必須將 dp 陣列的所有的位置都先設定成一個超大的數字，否則轉移出來的結果都會是 0，這一個動作叫做初始化**

動態規劃初探

- **轉移式：** $dp[i] = max(dp[i-2] + |a_i - a_{i-2}|, dp[i-1] + |a_i - a_{i-1}|)$
- **一開始我們必須將 dp 陣列的所有的位置都先設定成一個超大的數字，否則轉移出來的結果都會是 0，這一個動作叫做初始化**
- **接下來我們還必須定義初始狀態，轉移出來的結果才會是對的**
- **以這一題來說，初始狀態會是 $dp[1] = 0, dp[2] = |a_1 - a_0|$，因為一開始在第一個石頭時我們完全沒有任何花費，而在第二個石頭時我們一定只能從第一個石頭跳過來**

動態規劃初探

```cpp
int n;

cin >> n;

vector<int>a(n+1),dp(n+1,(int)1e9);

for(int i=1;i≤n;i++) cin >> a[i];

dp[1] = 0 , dp[2] = abs(a[2]-a[1]);

for(int i=3;i≤n;i++)
{
    dp[i] = min( dp[i-1] + abs(a[i]-a[i-1]) , dp[i-2] + abs(a[i]-a[i-2]) );
}

cout << dp[n] << "\n";
```

動態規劃初探

費氏數列

有 Q $(1 \leq Q \leq 2 \times 10^5)$ 組查詢，每組查詢將查詢費氏數列的第 k_i $(1 \leq k_i \leq 10^6)$ 項是多少

先讓大家想想看這一題應該怎麼做吧

費氏數列

有 Q $(1 \leq Q \leq 2 \times 10^5)$ 組查詢，每組查詢將查詢費氏數列的第 k_i $(1 \leq k_i \leq 10^6)$ 項是多少

- 這一題會用到建表的概念，我們可以一開始就先把所有項的答案都算出來，接下來每次查詢都可以在 $O(1)$ 的時間查詢

費氏數列

有 Q $(1 \leq Q \leq 2 \times 10^5)$ 組查詢，每組查詢將查詢費氏數列的第 k_i $(1 \leq k_i \leq 10^6)$ 項是多少

- 這一題會用到建表的概念，我們可以一開始就先把所有項的答案都算出來，接下來每次查詢都可以在 $O(1)$ 的時間查詢
- 定義轉移式 $dp[i]$ 表示費氏數列的第 i 項的值，那我們一開始就有了初始狀態 $dp[0] = 0, dp[1] = 1$，就可以很輕鬆地列出轉移式 $dp[i] = dp[i-1] + dp[i-2]$

多種選擇（？ 的動態規劃

多種選擇（？ 的動態規劃

Atcoder DP Contest pC. Vacation

Gino 接下來有 N $(1 \leq N \leq 10^5)$ 天的暑假，每一天你可以選擇要游泳、抓蟲還是寫暑假作業，如果在第 i 天游泳將得到 a_i 的快樂度、如果抓蟲將得到 b_i 的快樂度，寫暑假作業得到 c_i 的快樂度，Gino 不會連續兩天做同一件事情，請問他能得到的最大快樂度會是多少（雖然現實中暑假作業一定是在最後幾天才做，但這邊我們要先假裝我們是個好學生）

多種選擇（？ 的動態規劃

Atcoder DP Contest pC. Vacation

Gino 接下來有 N $(1 \leq N \leq 10^5)$ 天的暑假，每一天你可以選擇要游泳、抓蟲還是寫暑假作業，如果在第 i 天游泳將得到 a_i 的快樂度、如果抓蟲將得到 b_i 的快樂度，寫暑假作業得到 c_i 的快樂度，Gino 不會連續兩天做同一件事情，請問他能得到的最大快樂度會是多少（雖然現實中暑假作業一定是在最後幾天才做，但這邊我們要先假裝我們是個好學生）

- 轉移式感覺很複雜？ 狀態特別多？，有時候轉移式不一定會是一維！

多種選擇（？ 的動態規劃

Atcoder DP Contest pC. Vacation

Gino 接下來有 N $(1 \leq N \leq 10^5)$ 天的暑假，每一天你可以選擇要游泳、抓蟲還是寫暑假作業，如果在第 i 天游泳將得到 a_i 的快樂度、如果抓蟲將得到 b_i 的快樂度，寫暑假作業得到 c_i 的快樂度，Gino 不會連續兩天做同一件事情，請問他能得到的最大快樂度會是多少（雖然現實中暑假作業一定是在最後幾天才做，但這邊我們要先假裝我們是個好學生）

- 轉移式感覺很複雜？ 狀態特別多？，有時候轉移式不一定會是一維！
- 定義 $dp[k][i]$ 表示在第 i 天做第 k 件事情的最大快樂度

多種選擇（？ 的動態規劃

Atcoder DP Contest pC. Vacation

Gino 接下來有 N $(1 \leq N \leq 10^5)$ 天的暑假，每一天你可以選擇要游泳、抓蟲還是寫暑假作業，如果在第 i 天游泳將得到 a_i 的快樂度、如果抓蟲將得到 b_i 的快樂度，寫暑假作業得到 c_i 的快樂度，Gino 不會連續兩天做同一件事情，請問他能得到的最大快樂度會是多少（雖然現實中暑假作業一定是在最後幾天才做，但這邊我們要先假裝我們是個好學生）

- 轉移式感覺很複雜？ 狀態特別多？，有時候轉移式不一定會是一維！
- 定義 $dp[k][i]$ 表示在第 i 天做第 k 件事情的最大快樂度
- 接下來讓大家想想看，這樣定義的話，轉移式會是什麼呢

多種選擇（？ 的動態規劃

- 每一天我們都有三種選擇，如果第 i 天選擇了第 1 種，那麼我們第 $i-1$ 天就只能做第 2 種事情或第 3 種事情，因此就可以得到轉移式
  $dp[1][i] = \max(dp[2][i-1], dp[3][i-1])$

多種選擇 (? 的動態規劃

- 每一天我們都有三種選擇，如果第 i 天選擇了第 1 種，那麼我們第 $i-1$ 天就只能做第 2 種事情或第 3 種事情，因此就可以得到轉移式
  $dp[1][i] = \max(dp[2][i-1], dp[3][i-1]) + a_i$
- 照這樣的概念我們就可以推出所有的轉移式

$$dp[k][i] = \begin{cases} a_i, & \text{if } i = 1, k = 1 \\ b_i, & \text{if } i = 1, k = 2 \\ c_i, & \text{if } i = 1, k = 3 \\ \max(dp[2][i-1], dp[3][i-1]) + a_i, & \text{if } k = 1 \\ \max(dp[1][i-1], dp[3][i-1]) + b_i, & \text{if } k = 2 \\ \max(dp[1][i-1], dp[2][i-1]) + c_i, & \text{otherwise} \end{cases} \tag{1}$$

最後答案就會是 $\max(dp[1][N], dp[2][N], dp[3][N])$，時間複雜度 $O(N)$

多種選擇 (? 的動態規劃

```
25      cin >> n;
26
27      for(int i=1;i≤n;i++)
28      {
29          for(int k=1;k≤3;k++)
30          {
31              cin >> a[i][k];
32          }
33      }
34
35      dp[1][1] = a[1][1] , dp[1][2] = a[1][2] , dp[1][3] = a[1][3];
36
37      for(int i=2;i≤n;i++)
38      {
39          dp[i][1] = max(dp[i-1][2],dp[i-1][3]) + a[i][1];
40          dp[i][2] = max(dp[i-1][1],dp[i-1][3]) + a[i][2];
41          dp[i][3] = max(dp[i-1][1],dp[i-1][2]) + a[i][3];
42      }
43
44      cout << max({dp[n][1],dp[n][2],dp[n][3]}) << "\n";
```

多種選擇（? 的動態規劃

TNFSH OJ 470. 2020 台南一中 x 台南女中聯合寒訓 pD. 公假無雙

接下來將有 n $(1 \leq n \leq 10^6)$ 天，如果在第 i 天請公假將得到 a_i 的訓練成效，但公假不能連續請兩天，請問最大的訓練成效總和會是多少

- 這題有一維的轉移式也有二維的轉移式
- 對一維的沒有想法建議先想二維的

多種選擇（？ 的動態規劃

- 定義 $dp[i]$ 表示第 1 天到第 i 天的最大訓練成效總和

多種選擇（？ 的動態規劃

- 定義 $dp[i]$ 表示第 1 天到第 i 天的最大訓練成效總和
- 接下來有兩種選擇，一種是在第 i 天請公假，一種是不請
- 如果在第 i 天請公假就表示第 $i-1$ 天一定不能請公假，所以最大訓練成效總和會是 $dp[i-2]+a_i$

多種選擇（？ 的動態規劃

- 定義 $dp[i]$ 表示第 1 天到第 i 天的最大訓練成效總和
- 接下來有兩種選擇，一種是在第 i 天請公假，一種是不請
- 如果在第 i 天請公假就表示第 $i-1$ 天一定不能請公假，所以最大訓練成效總和會是 $dp[i-2]+a_i$
- 那如果這天不請公假的最大訓練成效總和就會是 $dp[i-1]$

多種選擇（？ 的動態規劃

- 定義 $dp[i]$ 表示第 1 天到第 i 天的最大訓練成效總和
- 接下來有兩種選擇，一種是在第 i 天請公假，一種是不請
- 如果在第 i 天請公假就表示第 $i-1$ 天一定不能請公假，所以最大訓練成效總和會是 $dp[i-2]+a_i$
- 那如果這天不請公假的最大訓練成效總和就會是 $dp[i-1]$
- 綜合以上就可以得到轉移式了！

$$dp[i] = \begin{cases} \max(0, a_i), & \texttt{if } i = 1 \\ \max(dp[i-2] + a_i, dp[i-1]), & \texttt{otherwise} \end{cases} \tag{2}$$

多種選擇 (? 的動態規劃

TNFSH OJ 489 第七屆大台南高一程式設計排名賽第六題. 狗狗大隊

現在有 n $(1 \leq n \leq 2 \times 10^5)$ 隻狗狗，每一隻狗狗有一個戰鬥力 a_i，接下來把這些狗狗進行分組，每一隻狗狗不一定都要被分組，但如果第 i 隻狗狗有被分組，那麼跟他相鄰的狗狗不能跟這隻狗狗不同組，每一組狗狗的數量一定都要是 k $(1 \leq k \leq n)$，請問最後有被分組的狗狗的戰鬥力總和最高可以是多少

這題給大家自己想看看 >< 有一點點小小小小的難

多種選擇（？ 的動態規劃

- 每一隻狗狗會有兩種選擇，第一種是被分組，另外一種是不被分組，那我們定義 $dp[i]$ 表示考慮第 $1 \sim i$ 隻狗狗的最大戰鬥力總和

多種選擇（？ 的動態規劃

- 每一隻狗狗會有兩種選擇，第一種是被分組，另外一種是不被分組，那我們定義 $dp[i]$ 表示考慮第 $1 \sim i$ 隻狗狗的最大戰鬥力總和
- 如果第 i 隻狗狗不被分組，那我們的最大戰鬥力總和就會是 $dp[i-1]$

多種選擇（？ 的動態規劃

- 每一隻狗狗會有兩種選擇，第一種是被分組，另外一種是不被分組，那我們定義 $dp[i]$ 表示考慮第 $1 \sim i$ 隻狗狗的最大戰鬥力總和
- 如果第 i 隻狗狗不被分組，那我們的最大戰鬥力總和就會是 $dp[i-1]$
- 如果第 i 隻狗要被分組，那麼就表示第 $i-k$ 這隻狗不能被分組，因此我們就可以知道最大的戰鬥力總和會是 $dp[i-k-1] + \sum_{j=i-k+1}^{i} a_j$

多種選擇 (? 的動態規劃

- 每一隻狗狗會有兩種選擇，第一種是被分組，另外一種是不被分組，那我們定義 $dp[i]$ 表示考慮第 $1 \sim i$ 隻狗狗的最大戰鬥力總和
- 如果第 i 隻狗狗不被分組，那我們的最大戰鬥力總和就會是 $dp[i-1]$
- 如果第 i 隻狗要被分組，那麼就表示第 $i-k$ 這隻狗不能被分組，因此我們就可以知道最大的戰鬥力總和會是 $dp[i-k-1] + \sum_{j=i-k+1}^{i} a_j$
- 綜合以上，我們就可以統整出轉移式

$$dp[i] = \begin{cases} 0, & \texttt{if } i < k \\ \sum_{j}^{i} a_j, & \texttt{if } i = k \\ \texttt{max}(dp[i-1], dp[i-k-1] + \sum_{j=i-k+1}^{i} a_j), & \texttt{otherwise} \end{cases} \quad (3)$$

多種選擇 (? 的動態規劃

```
21      cin >> n >> k;
22
23      for(int i=1;i≤n;i++) cin>> a[i] , b[i] = b[i-1] + b[i];
24
25      dp[k] = b[k];
26
27      for(int i=k+1;i≤n;i++)
28      {
29          dp[i] = max(dp[i-1] , dp[i-k-1] + s[i] - s[i-k]);
30      }
31
32      cout << dp[n] << '\n';
```

排列組合動態規劃

排列組合動態規劃

- 動態規劃也常常用於計算排列組合的題目上
- 這裡需要有高中排列組合的一些簡單先備知識，像是加法原理跟乘法原理這些基本的排列組合知識
- 相信數論講師這一個部分教的非常好 （o

加法原理

嚴格的定義來說，如果有多個集合 $S_1, S_2, ..., S_n$ 互斥，則滿足

$$\sum_{i=1}^{n} |S_i| = |S_1| \cup |S_2| \cup ... \cup |S_n|$$

Examples

Koying 決定要請所有營隊的學員喝飲料，每個學員只能選奶茶或紅茶其中一種，如果選奶茶則可以選三種配料，選紅茶可以選五種配料，每個配料只能選其中一種，假設一定要選配料，則會有 $3 + 5 = 8$ 種選擇

乘法原理

乘法原理

如果現在有數個集合 $S_1, ..., S_n$，乘法原理滿足 $|S_1| \cdot ... \cdot |S_n| = |S_1 \times ... \times S_n|$
這邊的 $\times$ 指的是笛卡兒積 (Cartesian product)

Examples

Koying 決定要請營隊的所有人喝飲料，你可以從 3 個飲料品項當中選一個，再從 7 種配料當中選一個做搭配，假如一定要選飲料跟搭配配料，那麼依照乘法原理，總共會有 $3 \times 7 = 10$ 種組合

CSES Problem Set Dice Combinations

現在有很多個六面骰，你會一個一個的骰這些骰子數次，請問骰出的總和為 n $(1 \leq n \leq 10^6)$ 共有幾種可能

舉例： 如果 $n = 3$ 那麼會有以下幾種可能

- 1 + 1 + 1 = 3
- 1 + 2 = 3
- 2 + 1 = 3
- 3

排列組合動態規劃

- 定義 $dp[i]$ 表示湊出點數 i 共有幾種可能

排列組合動態規劃

- 定義 $dp[i]$ 表示湊出點數 i 共有幾種可能
- 在點數 i 時共有 6 種可能，分別是從 $i-1, i-2, i-3, i-4, i-5, i-6$ 這六種點數再分別骰出點數 $1, 2, 3, 4, 5, 6$ 得來

排列組合動態規劃

- 定義 $dp[i]$ 表示湊出點數 i 共有幾種可能
- 在點數 i 時共有 6 種可能，分別是從 $i-1, i-2, i-3, i-4, i-5, i-6$ 這六種點數再分別骰出點數 $1, 2, 3, 4, 5, 6$ 得來
- 依照加法原理，我們可以得到轉移式
- $dp[i] = dp[i-1] + dp[i-2] + dp[i-3] + dp[i-4] + dp[i-5] + dp[i-6]$
- 題目有要求要將答案 `mod` $10^9 + 7$ 別忘記了
- `mod` 的數字建議用 `const int` 宣告，因為這一個數字永遠固定，用這樣的宣告方式會提升程式的效率

排列組合動態規劃

```
17    int n;
18    cin >> n;
19
20    dp[0] = 1;
21
22    for(int i=1;i≤n;i++)
23    {
24        for(int k=1;k≤min(i,6LL);k++)
25        {
26            dp[i] += dp[i-k];
27            dp[i] %= mod;
28        }
29    }
30
31    cout << dp[n] << "\n";
```

排列組合動態規劃

CSES Problem Set Coin Combinations I

現在你有 n $(1 \leq n \leq 100)$ 個硬幣，每一個硬幣有不同的幣值 C_i，請問要湊出幣值總和為 x $(1 \leq x \leq 10^6)$ 共有幾種組合

- 2+2+5 與 5+2+2 算是不同種組合

這題讓大家先想看看 ><

排列組合動態規劃

- 定義 $dp[i]$ 表示湊出 i 的組合數

排列組合動態規劃

- 定義 $dp[i]$ 表示湊出 i 的組合數
- 在 i 幣值會有 n 種可能，從 $i - C_1, i - C_2, .., i - C_n$ 這些幣值得來

排列組合動態規劃

- 定義 $dp[i]$ 表示湊出 i 的組合數
- 在 i 幣值會有 n 種可能，從 $i - C_1, i - C_2, .., i - C_n$ 這些幣值得來
- 藉由這個概念與加法原理我們就可以得到轉移式
- $dp[i]$ += $\sum_{k=1}^{n} dp[i - C_k]$
- 記得 mod $10^9 + 7$
- 還有一開始要把 $dp[0] = 1$

排列組合動態規劃

```
16    int n,m;
17    cin >> n >> m;
18
19    vector <int> a(n);
20
21    for(int i=0;i<n;i++)
22    {
23        cin >> a[i];
24    }
25
26    dp[0] = 1;
27    for(int i=1;i≤m;i++)
28    {
29        for(int k=0;k<n;k++)
30        {
31            if( i - a[k] ≥ 0 )
32            {
33                dp[i] += dp[i-a[k]];
34                dp[i] %= mod;
35            }
36        }
37    }
38    cout << dp[m] << "\n";
```

CSES Problem Set Grid Paths

你有一個 $n \times n$ $(1 \leq n \leq 1000)$ 的網格，'.' 是可以走的格子，'*' 是不可以走的格子，定義左上角為 $(1,1)$、右下角為 (n,n)，每一步只能往右走或往下走，請問從 $(1,1)$ 走到 (n,n) 共有幾種走法

排列組合動態規劃

- 定義 $dp[i][k]$ 表示 $(1,1)$ 走到 (i,k) 的走法總數

排列組合動態規劃

- 定義 $dp[i][k]$ 表示 $(1,1)$ 走到 (i,k) 的走法總數
- 在 (i,k) 時只有兩種可能，一種是從 $(i-1,k)$ 走到 (i,k)，另外一種是從 $(i,k-1)$ 走到 (i,k)

排列組合動態規劃

- 定義 $dp[i][k]$ 表示 $(1,1)$ 走到 (i,k) 的走法總數
- 在 (i,k) 時只有兩種可能，一種是從 $(i-1,k)$ 走到 (i,k)，另外一種是從 $(i,k-1)$ 走到 (i,k)
- 我們就可以很輕鬆的列出轉移式

$$dp[i][k] = \begin{cases} 0, & \texttt{if } (i,k) \texttt{ is *} \\ dp[i-1][k] + dp[i][k-1], & \texttt{otherwise} \end{cases} \tag{4}$$

排列組合動態規劃

```
37      for(int i=1;i≤n;i++)
38      {
39          for(int k=1;k≤n;k++)
40          {
41              if( i == 1 && k == 1 )
42              {
43                  if( mp[i][k] == '*' ) dp[1][1] = 0;
44                  else dp[1][1] = 1;
45
46                  continue;
47              }
48
49              if( mp[i][k] == '*' ) continue;
50
51              dp[i][k] = dp[i-1][k] + dp[i][k-1];
52              dp[i][k] = ADD(dp[i][k]);
53          }
54      }
55
56      cout << dp[n][n] << "\n";
```

動態規劃 - 滾動優化

動態規劃-滾動優化

- 節省記憶體的技巧
- 通常用於發現轉移點只會用到上一次的狀態時會使用這樣的優化，藉此降低空間複雜度
- 其概念是如果轉移的狀態只會用到上一次的資訊，那麼上上次以前的資訊其實我們都可以丟掉不用

動態規劃-滾動優化

CSES Problem Set Coin Combinations II

你有 n $(1 \leq n \leq 100)$ 種硬幣，每一種硬幣有不同的幣值 C_i，請問要湊出 x $(1 \leq x \leq 10^6)$ 的幣值總和共有幾種可能

- 1+1+2 與 2+1+1 視為同一種

這題如果你用沒有優化的二維的轉移式，空間複雜度會是 $O(nx)$，那我們應該怎麼做呢？

動態規劃-滾動優化

- 我們一樣先列出沒有優化的轉移式

動態規劃-滾動優化

- 我們一樣先列出沒有優化的轉移式
- 定義 $dp[i][k]$ 表示只使用前 i 種硬幣湊出幣值 k 的組合數量

動態規劃-滾動優化

- 我們一樣先列出沒有優化的轉移式
- 定義 $dp[i][k]$ 表示只使用前 i 種硬幣湊出幣值 k 的組合數量
- 在幣值 k 時對於第 i 個硬幣會有兩種可能，一種是使用第 i 個硬幣，另外一種是不使用第 i 個硬幣

動態規劃-滾動優化

- 我們一樣先列出沒有優化的轉移式
- 定義 $dp[i][k]$ 表示只使用前 i 種硬幣湊出幣值 k 的組合數量
- 在幣值 k 時對於第 i 個硬幣會有兩種可能，一種是使用第 i 個硬幣，另外一種是不使用第 i 個硬幣
- 如果不使用第 i 種硬幣要湊出幣值 k 會有 $dp[i-1][k]$ 種可能

動態規劃-滾動優化

- 我們一樣先列出沒有優化的轉移式
- 定義 $dp[i][k]$ 表示只使用前 i 種硬幣湊出幣值 k 的組合數量
- 在幣值 k 時對於第 i 個硬幣會有兩種可能，一種是使用第 i 個硬幣，另外一種是不使用第 i 個硬幣
- 如果不使用第 i 種硬幣要湊出幣值 k 會有 $dp[i-1][k]$ 種可能
- 如果使用第 i 種硬幣要湊出幣值 k 則會有 $dp[i][k-C_i]$ 種可能（因為可以重複拿，所以轉移點會是 $dp[i][k-C_i]$）

動態規劃-滾動優化

- 我們一樣先列出沒有優化的轉移式
- 定義 $dp[i][k]$ 表示只使用前 i 種硬幣湊出幣值 k 的組合數量
- 在幣值 k 時對於第 i 個硬幣會有兩種可能，一種是使用第 i 個硬幣，另外一種是不使用第 i 個硬幣
- 如果不使用第 i 種硬幣要湊出幣值 k 會有 $dp[i-1][k]$ 種可能
- 如果使用第 i 種硬幣要湊出幣值 k 則會有 $dp[i][k-C_i]$ 種可能（因為可以重複拿，所以轉移點會是 $dp[i][k-C_i]$）
- 綜合以上我們得到轉移式
- $dp[i][k] = dp[i-1][k] + dp[i][k-C_i]$
- 記得一開始要把 $dp[0][0] = 1$

動態規劃-滾動優化

- $dp[i][k] = dp[i-1][k] + dp[i][j - C_i]$
- **我們先觀察一下轉移式，有什麼特性**

動態規劃-滾動優化

- $dp[i][k] = dp[i-1][k] + dp[i-1][j - C_i]$
- 我們先觀察一下轉移式，有什麼特性
- 你會發現每一次 $dp[i][k]$ 的狀態都只跟 $dp[i-1][k]$ 有關，因此我們每次只需要知道上一次的資訊就足夠得到這一次的答案了

動態規劃-滾動優化

- $dp[i][k] = dp[i-1][k] + dp[i][k-C_i]$
- 我們先觀察一下轉移式，有什麼特性
- 你會發現每一次 $dp[i][k]$ 的狀態都只跟 $dp[i-1][k]$ 有關，因此我們每次只需要知道上一次的資訊就足夠得到這一次的答案了
- 那如果我們現在用奇數與偶數來區分資訊，那如果當 `dp[i][k]` 的 `i` 是奇數時，我們就可以確定，上一層的資訊在偶數那邊，反之，當 `i` 是偶數，我們就可以知道上一層的資訊在奇數那邊

動態規劃-滾動優化

- $dp[i][k] = dp[i-1][k] + dp[i-1][k-C_i]$
- 我們先觀察一下轉移式，有什麼特性
- 你會發現每一次 $dp[i][k]$ 的狀態都只跟 $dp[i-1][k]$ 有關，因此我們每次只需要知道上一次的資訊就足夠得到這一次的答案了
- 那如果我們現在用奇數與偶數來區分資訊，那如果當 `dp[i][k]` 的 `i` 是奇數時，我們就可以確定，上一層的資訊在偶數那邊，反之，當 `i` 是偶數，我們就可以知道上一層的資訊在奇數那邊
- 因此我們可以把 `dp` 陣列開成 `dp[2][1000001]` 的大小，並將轉移式改寫成 $dp[i\ \%2][k] = dp[(i+1)\%2][k] + dp[i\ \%2][j-C_i]$
- 如此一來就可以成功的節省掉了非常多的記憶體

再稍微小小的優化？

- **轉移式：** $dp[i \ \%2][k] = dp[(i+1)\%2][k] + dp[i \ \%2][j - C_i]$

再稍微小小的優化?

- **轉移式：** $dp[i\ \%2][k] = dp[(i+1)\%2][k] + dp[i\ \%2][j - C_i]$
- **觀察一下，我們可以發現每一個** $dp[i][k]$ **的資訊都直接是上一層的** $dp[i-1][k]$ **加上** $dp[i][k - C_i]$

再稍微小小的優化？

- **轉移式：** $dp[i\ \%2][k] = dp[(i+1)\%2][k] + dp[i\ \%2][j - C_i]$
- **觀察一下，我們可以發現每一個** $dp[i][k]$ **的資訊都直接是上一層的** $dp[i-1][k]$ **加上** $dp[i][k - C_i]$
- **既然每一次都是直接拿上一層的東西加上** $dp[i][k - C_i]$ **那我們就乾脆直接把上一次的資訊覆蓋掉就好了啊！轉移式就可以直接變成一維的呢**
- $dp[k]$ **+=** $dp[k - C_i]$

動態規劃-滾動優化

```
21      dp[0] = 1;
22
23      for(int i=0;i<n;i++)
24      {
25          int input;
26
27          cin >> input;
28
29          for(int k=input;k≤m;k++)
30          {
31              dp[k] += dp[k-input];
32
33              if( dp[k] ≥ mod ) dp[k] -= mod;
34          }
35      }
36
37      cout << dp[m] << "\n";
```

背包問題

背包問題

Atcoder Beginner Contest pD. Knapsack 1

現在有 N 個物品，每一個物品有一個重量 w_i $(1 \leq w_i \leq 10^5)$ 與價值 v_i，有一個背包可以裝下總重量 W $(1 \leq W \leq 10^5)$，請問這個背包最多可以裝下多少價值的物品

背包問題

Atcoder Beginner Contest pD. Knapsack 1

現在有 N 個物品，每一個物品有一個重量 w_i $(1 \leq w_i \leq 10^5)$ 與價值 v_i，有一個背包可以裝下總重量 W $(1 \leq W \leq 10^5)$，請問這個背包最多可以裝下多少價值的物品

- 定義 $dp[i][k]$ 表示考慮前 i 種物品且在背包容量為 k 時可以裝到的最大價值

Atcoder Beginner Contest pD. Knapsack 1 (0/1 背包問題)

現在有 N 個物品，每一個物品有一個重量 w_i $(1 \leq w_i \leq 10^5)$ 與價值 v_i，有一個背包可以裝下總重量 W $(1 \leq W \leq 10^5)$，請問這個背包最多可以裝下多少價值的物品

- 定義 $dp[i][k]$ 表示考慮前 i 種物品且在背包容量為 k 時可以裝到的最大價值
- 在考慮第 i 個物品且背包容量為 k 時，第 i 個物品有兩種選擇，一種是拿、另外一種是不拿，如果拿了，那麼在考慮前 $i-1$ 個物品時，背包最多只能是 $k-w_i$ 的容量，否則裝下第 i 個物品就超出 k 容量了，因此可以得到價值 $dp[i-1][k-w_i]+v_i$，不拿第 i 個物品我們可以得到的最大價值為 $dp[i-1][k]$，綜合以上我們就可以得到轉移式
- $dp[i][k] = \max(dp[i-1][k], dp[i-1][k-w_i]+v_i)$

背包問題-滾動優化

- $dp[i][k] = \max(dp[i-1][k], dp[i-1][k-w_i] + v_i)$
- 觀察一下你會發現，背包問題的轉移式也都是只跟上一次的資訊有關，因此是可以用滾動優化的技巧來降低空間複雜度的，實作的方法一樣也是利用 i 的奇偶性

背包問題-滾動優化

- $dp[i][k] = \max(dp[i-1][k], dp[i-1][k-w_i] + v_i)$
- 觀察一下你會發現，背包問題的轉移式也都是只跟上一次的資訊有關，因此是可以用滾動優化的技巧來降低空間複雜度的，實作的方法一樣也是利用 i 的奇偶性
- 但是，背包問題其實是可以只用一維陣列做到的

背包問題-滾動優化

- $dp[i][k] = \max(dp[i-1][k], dp[i-1][k-w_i] + v_i)$
- 觀察一下你會發現，背包問題的轉移式也都是只跟上一次的資訊有關，因此是可以用滾動優化的技巧來降低空間複雜度的，實作的方法一樣也是利用 i 的奇偶性
- 但是，背包問題其實是可以只用一維陣列做到的
- 觀察一下轉移式你會發現，其實我們也可以直接把上一次的資訊拿來覆蓋，如果要這樣的話我們的轉移式會變成
- 定義 $dp[i]$ 表示背包容量為 i 時的最大價值
- 第 j 個物品時 $dp[i] = max(dp[i], dp[i-w_j] + v_j)$

一維的背包轉移

- 定義 $dp[i]$ 表示背包容量為 i 時的最大價值
- 第 j 個物品時 $dp[i] = max(dp[i], dp[i - w_j] + v_j)$
- 觀察一下你會發現，如果我們在轉移的過程迴圈的順序是 $i = w_j, w_j + 1, ..., W$，這樣的話會發生什麼事情呢？

一維的背包轉移

- 定義 $dp[i]$ 表示背包容量為 i 時的最大價值
- 第 j 個物品時 $dp[i] = \max(dp[i], dp[i - w_j] + v_j)$
- 觀察一下你會發現，如果我們在轉移的過程迴圈的順序是 $i = w_j, w_j + 1, ..., W$，這樣的話會發生什麼事情呢？
- 假設當前 $i = 5, w_j = 3$，那麼轉移狀態會是 $dp[5] = \max(dp[5], dp[5 - 3] + v_j)$，但是當 $i = 8$ 時，轉移狀態會是 $dp[8] = \max(dp[8], dp[8 - 3] + v_j)$，而 $8 - 3 = 5$，但是 $dp[5]$ 有可能已經拿過第 j 個物品了，你會發現這不符合題目的要求，同一種物品只能拿最多一次

一維的背包轉移

- 定義 $dp[i]$ 表示背包容量為 i 時的最大價值
- 第 j 個物品時 $dp[i] = \max(dp[i], dp[i - w_j] + v_j)$
- 觀察一下你會發現，如果我們在轉移的過程迴圈的順序是 $i = w_j, w_j + 1, ..., W$，這樣的話會發生什麼事情呢？
- 假設當前 $i = 5, w_j = 3$，那麼轉移狀態會是 $dp[5] = \max(dp[5], dp[5 - 3] + v_j)$，但是當 $i = 8$ 時，轉移狀態會是 $dp[8] = \max(dp[8], dp[8 - 3] + v_j)$，而 $8 - 3 = 5$，但是 $dp[5]$ 有可能已經拿過第 j 個物品了，你會發現這不符合題目的要求，同一種物品只能拿最多一次
- 那我們應該怎麼做呢？
- 如果我們把迴圈順序變成從最後面回來，也就是 $i = W, W - 1, ..., w_j$ 這樣子的順序，你會發現就解決掉這個問題了耶！聽起來很棒對吧

一維的背包轉移

- 照原本的迴圈順序會變成物品可以被重複拿，就會變成無限背包問題的解法
- 如果每種物品只能拿一次，那一維時迴圈的順序就必須反過來，這樣才會是 0/1 背包問題的解法

背包問題範例

```
23      int n,w;
24
25      cin >> n >> w;
26
27      for(int i=1;i≤n;i++)
28      {
29          cin >> u[i] >> v[i];
30
31          for(int k=w;k≥u[i];k--)
32          {
33              dp[k] = max(dp[k],dp[k-u[i]] + v[i]);
34          }
35      }
36
37      cout << dp[w] << "\n";
38  }
```

不一樣的背包問題

Atcoder DP Contest pE Knapsack 2

與 Knapsack 1 題目一樣，只是物品重量與背包最大承受重量最大來到了 10^9，但現在物品的價值最多只有 10^3

如果照原本的轉移式的話，陣列可開不了這麼大唷 ...

不一樣的背包問題

- 觀察到價值的值域非常小，所以我們要針對價值來下手

不一樣的背包問題

- 觀察到價值的值域非常小，所以我們要針對價值來下手
- 定義 $dp[i]$ 表示湊出價值 i 所需要的最小重量

不一樣的背包問題

- 觀察到價值的值域非常小，所以我們要針對價值來下手
- 定義 $dp[i]$ 表示湊出價值 i 所需要的最小重量
- 如此一來轉移式自然就會出來了！
- 對於第 j 個物品 $dp[i] = \texttt{min}(dp[i], dp[i - v_j] + w_j)$

不一樣的背包問題

- 觀察到價值的值域非常小，所以我們要針對價值來下手
- 定義 $dp[i]$ 表示湊出價值 i 所需要的最小重量
- 如此一來轉移式自然就會出來了！
- 對於第 j 個物品 $dp[i] = \min(dp[i], dp[i - v_j] + w_j)$
- 這樣的轉移式也要特別注意在 0/1 背包問題時，記得迴圈的順序要從最大價值往回跑，才不會算成無限背包哦！

不一樣的背包問題

```
23    fill(dp,dp+100001,(int)1e18);
24
25    int n,w;
26
27    cin >> n >> w;
28
29    dp[0] = 0;
30
31    for(int i=1;i≤n;i++)
32    {
33        cin >> u[i] >> v[i];
34
35        for(int k=100000;k≥v[i];k--)
36        {
37            dp[k] = min(dp[k],dp[k-v[i]]+u[i]);
38        }
39    }
40
41    int ans = 0;
42
43    for(int i=0;i≤100000;i++) if( dp[i] ≤ w ) ans = max(ans,i);
44
45    cout << ans << "\n";
```

最長遞增子序列

最長遞增子序列

給定一個長度為 N $(1 \leq N \leq 1000)$ 的序列 a，請你找出一些位置 $(p_1, ..., p_k)$，滿足 $a_{p_1} < a_{p_2} < ... < a_{p_k}$，請問 k 最大可以是多少

最長遞增子序列

給定一個長度為 N $(1 \leq N \leq 1000)$ 的序列 a，請你找出一些位置 $(p_1, ..., p_k)$，滿足 $a_{p_1} < a_{p_2} < ... < a_{p_k}$，請問 k 最大可以是多少

- 定義 $dp[i]$ 表示以第 i 個位置為結尾的最長遞增子序列長度

最長遞增子序列

最長遞增子序列

給定一個長度為 N $(1 \leq N \leq 1000)$ 的序列 a，請你找出一些位置 $(p_1, ..., p_k)$，滿足 $a_{p_1} < a_{p_2} < ... < a_{p_k}$，請問 k 最大可以是多少

- 定義 $dp[i]$ 表示以第 i 個位置為結尾的最長遞增子序列長度
- 那我們接下來想要得到 $dp[i]$ 就要去檢查 $< i$ 的所有位置，如果 $j < i$ 且 $a_j < a_i$ 則表示當前以 a_j 為結尾的最長遞增子序列可以讓 a_i 接上來，最長遞增子序列長度則會變成 $dp[j] + 1$，由於有可能有很多位置都滿足上面這個條件，但我們要取最好的，因此就可以得到轉移式

最長遞增子序列

給定一個長度為 N $(1 \leq N \leq 1000)$ 的序列 a，請你找出一些位置 $(p_1, ..., p_k)$，滿足 $a_{p_1} < a_{p_2} < ... < a_{p_k}$，請問 k 最大可以是多少

- 定義 $dp[i]$ 表示以第 i 個位置為結尾的最長遞增子序列長度
- 那我們接下來想要得到 $dp[i]$ 就要去檢查 $< i$ 的所有位置，如果 $j < i$ 且 $a_j < a_i$ 則表示當前以 a_j 為結尾的最長遞增子序列可以讓 a_i 接上來，最長遞增子序列長度則會變成 $dp[j] + 1$，由於有可能有很多位置都滿足上面這個條件，但我們要取最好的，因此就可以得到轉移式
- $dp[i] = \texttt{max}(dp[i], dp[j] + 1)$ **if** $j < i$ 且 $a_j < a_i$

最長遞增子序列

給定一個長度為 N $(1 \leq N \leq 1000)$ 的序列 a，請你找出一些位置 $(p_1, ..., p_k)$，滿足 $a_{p_1} < a_{p_2} < ... < a_{p_k}$，請問 k 最大可以是多少

- 定義 $dp[i]$ 表示以第 i 個位置為結尾的最長遞增子序列長度
- 那我們接下來想要得到 $dp[i]$ 就要去檢查 $< i$ 的所有位置，如果 $j < i$ 且 $a_j < a_i$ 則表示當前以 a_j 為結尾的最長遞增子序列可以讓 a_i 接上來，最長遞增子序列長度則會變成 $dp[j] + 1$，由於有可能有很多位置都滿足上面這個條件，但我們要取最好的，因此就可以得到轉移式
- $dp[i] = \texttt{max}(dp[i], dp[j] + 1)$ **if** $j < i$ 且 $a_j < a_i$
- 這麼做的時間複雜度會是 $O(N^2)$，但其實有 $O(N \log N)$ 的做法，這個動態規劃（二）的課程會提到

最長共同子序列

最長共同子序列

Atcoder DP Contest pF. LCS

給兩個長度不超過 1000 的字串，請你輸出其中一組最長共同子序列
子序列的定義是如果第一個字串中的某些字元在第二個字串都有出現，且相對順序不變，則我們稱這一個字串是第一個字串與第二個字串的共同子序列

最長共同子序列

- 定義 $dp[i][k]$ 表示第一個字串的前 i 個字元與第二個字串的前 k 個字元的最長共同子序列長度

最長共同子序列

- 定義 $dp[i][k]$ 表示第一個字串的前 i 個字元與第二個字串的前 k 個字元的最長共同子序列長度
- 如果第一個字串的第 i 個字元與第 k 個字元一樣，那就表示最長共同子序列的長度會是 $dp[i][k] = dp[i-1][k-1] + 1$

最長共同子序列

- 定義 $dp[i][k]$ 表示第一個字串的前 i 個字元與第二個字串的前 k 個字元的最長共同子序列長度
- 如果第一個字串的第 i 個字元與第 k 個字元一樣，那就表示最長共同子序列的長度會是 $dp[i][k] = dp[i-1][k-1] + 1$
- 如果不一樣則看第一個字串少一個字元時比較好還是第二個字串少一個字元時比較好因此 $dp[i][k] = \max(dp[i-1][k], dp[i][k-1])$

最長共同子序列

- 定義 $dp[i][k]$ 表示第一個字串的前 i 個字元與第二個字串的前 k 個字元的最長共同子序列長度
- 如果第一個字串的第 i 個字元與第 k 個字元一樣，那就表示最長共同子序列的長度會是 $dp[i][k] = dp[i-1][k-1] + 1$
- 如果不一樣則看第一個字串少一個字元時比較好還是第二個字串少一個字元時比較好因此 $dp[i][k] = \mathbf{max}(dp[i-1][k], dp[i][k-1])$
- 統整一下轉移式

$$dp[i][k] = \begin{cases} dp[i-1][k-1] + 1, & \textbf{if } S_{1,i} = S_{2,k} \\ \mathbf{max}(dp[i-1][k], dp[i][k-1]), & \textbf{otherwise} \end{cases} \tag{5}$$

但是這一題要構造其中一組解

- 不用緊張，我們只要記錄每一個位置 (i, j) 是從哪一個位置轉移過來的就可以了

但是這一題要構造其中一組解

- 不用緊張，我們只要記錄每一個位置 (i, j) 是從哪一個位置轉移過來的就可以了
- 如果位置 (i, j) 是從 $(i-1, j-1)$ 所轉移過來的，我們就可以知道 $S_{1,i}$ 或 $S_{2,j}$ 會是我們最長共同子序列的字元

但是這一題要構造其中一組解

- 不用緊張，我們只要記錄每一個位置 (i, j) 是從哪一個位置轉移過來的就可以了
- 如果位置 (i, j) 是從 $(i-1, j-1)$ 所轉移過來的，我們就可以知道 $S_{1,i}$ 或 $S_{2,j}$ 會是我們最長共同子序列的字元
- 因此我們只要轉移時有先記錄好每一個點是被誰轉移過來的，就可以從 (n, m)（這邊指的 n 是第一個字串的長度，m 是第二個字串的長度）開始回朔回去，一直往往回跑，往回跑的同時順便紀錄路上的最長共同子序列的字元，就可以構造出一組解了

範例程式

```
25      string s1,s2;
26
27      cin >> s1 >> s2;
28
29      for(int i=0;i<s1.size();i++)
30      {
31          for(int k=0;k<s2.size();k++)
32          {
33              if( s1[i] == s2[k] )
34              {
35                  dp[i+1][k+1] = dp[i][k] + 1;
36                  pre[i+1][k+1] = {i,k};
37              }
38              else if( dp[i+1][k] ≥ dp[i][k+1] )
39              {
40                  dp[i+1][k+1] = dp[i+1][k];
41                  pre[i+1][k+1] = {i+1,k};
42              }
43              else
44              {
45                  dp[i+1][k+1] = dp[i][k+1];
46                  pre[i+1][k+1] = {i,k+1};
47              }
48          }
49      }
```

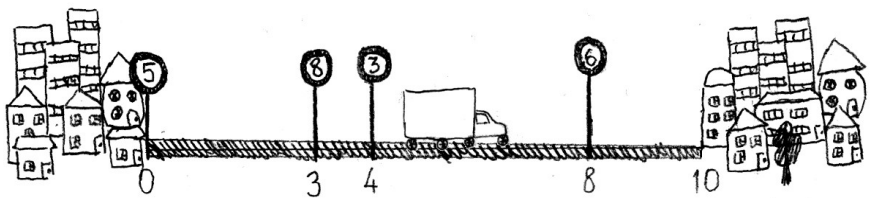
範例程式

```
53    int index1 = s1.size() , index2 = s2.size();
54
55    string ans;
56
57    while( index1 ≥ 1 && index2 ≥ 1  )
58    {
59        if( pre[index1][index2] == make_pair(index1-1,index2-1) )
60        {
61            ans += s1[index1-1];
62        }
63
64        pair <int,int> u = pre[index1][index2];
65
66        index1 = u.first;
67        index2 = u.second;
68    }
69
70    for(int i=ans.size()-1;i≥0;i--) cout << ans[i];
```

枚舉與動態規劃

Codeforces Round #765 pC. Road Optimization

給你 n $(1 \le n \le 500)$ 個速度牌，每一個速度牌上有一個數字 s_i，且位於數線上的某一點 d_i $(d_1 = 0)$，現在起點到終點的距離為 l $(1 \le l \le 10^5)$，每經過一個速度牌你就必須改變當前的速度，如果經過的速度牌標示的速度為 t，則接下來每一單位的距離你會花費 t 分鐘來走，現在你覺得這樣遵守速度規定很沒有意義，所以你決定最多拆除 k $(1 \le k \le n-1)$ 個速度牌來幫你加速，拆太多會被發現所以拆一些就好了，請問從起點到達終點的最少時間是多少



枚舉與動態規劃

- 這是一個標準的在比賽中會出現的動態規劃題
- 我們定義 $dp[i][j]$ 表示總共拆除 j 個速度牌到達第 i 個速度牌所需花費的最少時間

枚舉與動態規劃

- 這是一個標準的在比賽中會出現的動態規劃題
- 我們定義 $dp[i][j]$ 表示總共拆除 j 個速度牌到達第 i 個速度牌所需花費的最少時間
- 接著我們想想看轉移式，到達 i 這一個速度牌有好幾種方法，有可能是從第 $i-1$ 個速度牌過來的也有可能是從第 $i-2$ 個速度牌來的（如果你選擇拆掉第 $i-1$ 個速度牌的話），共有 k 種可能
- 有 k 種可能的話我們就必須枚舉是從哪一個速度牌過來的，有了這一個概念後我們就可以列出轉移式了

枚舉與動態規劃

- 這是一個標準的在比賽中會出現的動態規劃題
- 我們定義 $dp[i][j]$ 表示總共拆除 j 個速度牌到達第 i 個速度牌所需花費的最少時間
- 接著我們想想看轉移式，到達 i 這一個速度牌有好幾種方法，有可能是從第 $i-1$ 個速度牌過來的也有可能是從第 $i-2$ 個速度牌來的（如果你選擇拆掉第 $i-1$ 個速度牌的話），共有 k 種可能
- 有 k 種可能的話我們就必須枚舉是從哪一個速度牌過來的，有了這一個概念後我們就可以列出轉移式了
- 對於 $dp[i][j]$ 來說，如果從第 p $(1 \leq p < i)$ 個速度牌過來的話，轉移式就會是以下這一個式子
- $dp[i][j] = \texttt{min}(dp[i][j], dp[p][j-(i-p-1)] + (d_i - d_p) \times s_p)$
- 由於從 p 到 i 需要拆除 $i-p-1$ 個速度牌，因此在計算 $dp[i][j]$ 時，必須從 $dp[p][j-(i-p-1)]$ 轉移

枚舉與動態規劃

```
22    int n,m,k;
23    cin >> n >> m >> k;
24    vector<int> a(n+1,0),b(n+1,0);
25    for(int i=0;i<n;i++) cin >> a[i]; // 速度牌的位置
26    for(int i=0;i<n;i++) cin >> b[i]; // 標示的速度
27    for(int i=1;i≤n;i++) // 初始化
28    {
29        for(int j=0;j≤k;j++) dp[i][j] = 1e18;
30    }
31    for(int i=0;i≤k;i++) dp[0][i] = 0;
32
33    a[n] = m; // 幫終點開一個點，等等計算會比較方便
34
35    for(int i=1;i≤n;i++)
36    {
37        for(int j=0;j≤k;j++)
38        {
39            for(int p=i-1;p≥0;p--) // 枚舉從哪邊過來 i
40            {
41                if( j - ( i - p - 1 ) < 0 ) break; // 一定要這一行，不然陣列會溢位
42
43                dp[i][j] = min(dp[i][j],dp[p][j-(i-p-1)]+b[p] * (a[i]-a[p]));
44            }
45        }
46    }
47
48    cout << *min_element(dp[n],dp[n]+k+1) << "\n";
```

結尾

- 不知道大家對動態規劃有沒有更加地了解了
- 大家覺得動態規劃有趣嗎？
- 這一堂課有讓你收穫到一些不一樣的知識嗎
- 這一堂課就這樣結束了，希望大家喜歡
- 我的下一堂課是資料結構（二），到時候我會帶領大家探索資料結構的奇幻世界，大家就好好期待吧
- 也謝謝大家這次這麼認真的聽我上課還有跟我互動，有問題的也非常歡迎找我討論唷！