algorithm 初階

edisonhello

2017 年 9 月 19 日

1 Dynamic Programming 動態規劃

當會一直要算同一件事情的時候，我們可以把之前算好的結果存起來。這樣就不用每次都要遞迴下去找答案了。

1.1 兩種 DP

1.1.1 Top-down

直接採用遞迴。遞迴時，如果這個問題的答案沒有被算過，那就依照規則算一次。如果已經被算過了，那就直接回傳答案。優點是可以不用想要怎麼訂定一個良好的順序才不會讓他爛掉，跟一些用不到的東西不會被算到；缺點是遞迴的時間可能會比較高一點。

Algorithm 1: Fibonacci numbers (Top-down)

```
1 int F[100];
2 int fib(int n){
3     if(n<=2)return !!n;
4     if(F[n])return F[n];
5     return F[n]=fib(n-1)+fib(n-2);
6 }
```

1.1.2 Bottom-up

制定一個 DP 的順序，讓計算時所需的答案在之前已經被算過。好處是時間可能快一點，但是如果順序錯了就爛了。

Algorithm 2: Fibonacci numbers (Bottom-up)

```
int F[100];
void sol_fib(int n){
    F[1]=1;
    for(int i=2;i<=n;++i){
        F[i]=F[i-1]+F[i-2];
    }
}
```

1.2 狀態、轉移、複雜度

當一個 DP 寫好之後，我們可以算出它的複雜度。以前面的費氏數列做為例子：我們一共存了 n 個答案，所以它的狀態數，剛好也是空間複雜度，為 $O(n)$。(後面會提到狀態數 $\neq$ 空間複雜度的例子。) 至於轉移，算出每個答案只需要取前兩項的答案，所以轉移的複雜度是 $O(1)$。有了狀態數跟轉移複雜度之後，這個 DP 的總複雜度就會是這兩個東西相乘。所以求費氏數列的總複雜度就是 $O(n) \times O(1) = O(n)$。

1.3 滾動 DP

這是 DP 的其中一種優化方法，目的是可以減少 DP 時使用的空間。而這就是上面所提到會使狀態數 $\neq$ 空間複雜度的例子。當 DP 時，如果一個值不會被再度用掉，那我們就可以把該位子的值覆蓋掉。

Algorithm 3: Fibonacci numbers (space optimized DP)

```
int F[2]={0,1};
int sol_fib(int n){
    for(int i=2;i<=n;++i){
        F[i&1]=F[0]+F[1];
    }
    return F[i&1];
}
```

如此一來，空間複雜度就降到了 $O(1)$。

1.4 例題們

1. 最長共同子序列 (LCS)：給兩個序列，求最長的序列長度或將其還原。複雜度 $O(NM)$。
2. 01 背包問題 (Zerojudge D637)：有 N 個物品，每個物品只有一個，且有重量跟價值。求在總重 $\leq W$ 限制下能取到的最大價值。複雜度 $O(NW)$。
3. 無限背包問題：同上，但物品有無限個。複雜度一樣。

4. 旅行推銷員問題（位元 DP）：給 n 個點的圖，求經過每個點的迴路最小長度。以 int 中每個 bit 的 1 或 0 代表走過/未走過，複雜度 $O(n^2 2^n)$。
5. Zerojudge D652：有 N 個東西，每個東西有一個值 P_i。每次刪掉一個東西 k，付出的代價是 $P_{k-1}P_kP_{k+1}$。求當剩下左右兩個東西時最小代價為多少。

2 Greedy 貪婪

2.1 to greed, or not to greed

有時候選擇當下最好的選擇就會是整體中最好的選擇。例如在台灣買東西時，最小化付出的硬幣總數。這時候在選擇硬幣時，每次都選擇面額最大的那個就可以達到想要的結果。而「每次都選擇面額最大」這件事情就是貪婪的例子。但是如果今天不在台灣，到了一個硬幣只有 1,20,24 元三種面額的國家，這時要付出 40 元就不應該先選 24 元了。所以當想到貪婪解法時，不妨多設一些例子自行驗證。如果想不到反例的話，那就交給 judge 驗證吧。

2.2 某個可以貪婪的題目

給一些直線上的線段，求最多可以取幾個線段，使所有取出來的線段互不覆蓋。下面有三個作法，但是只有一種的作法是好的。可以試著舉出例子來讓其中兩個取法爛掉，你就得到答案了。

1. 從長度最小的開始掃。如果不重複就取。
2. 依照左端點順序，從左邊開始掃。如果不重複就取。
3. 依照右端點順序，從左邊開始掃。如果不重複就取。

2.3 例題們

1. TIOJ 1072：有 N 個人各點了一道菜，每道菜各有需要的烹煮時間跟吃完時間，每個時間只能同時煮一道菜。訂定一個煮菜順序使得從第一道菜開始煮到最後一個人把菜吃完所過的時間的最小值。
2. Codeforces 665C：給一個長度為 N 的字串，求最小的 edit distance 使相鄰字元都相異。

3 搜 (?)

3.1 二分搜

當我們有一個有單調性質的函數的時候，我們可以用二分搜求出某個我們要找的值，或是他的 lower_bound 或 upper_bound，而比起一個一個搜尋，複雜度可以提升到 $O(\lg n)$。假設我們有一個遞增的函數，搜尋的範圍是 $[l,r]$，那我們可以計算出 $f(mid)$ 的值，其中 $mid=\frac{l+r}{2}$。如果這個值比想要的值還要小，那應該把區間縮減成 $(mid,r]$，反之應該要將其變成 $[l,mid]$。因為每一次的搜尋都可以讓範圍變成一半，所以複雜度就會是 $O(k\lg n)$，k 是原本函數的複雜度。

Algorithm 4: Binary search

```
1 int binarySearch(int l,int r,int x){
2     assert(r>=l);
3     while(r>l){
4         int mid=(l+r)>>1;
5         if(f(mid)>=x)r=mid;
6         else l=mid+1;
7     }
8     return f(l)==x?l:-1;
9 }
```

3.2 三分搜

如果現在的函數是一個長得凹凹的或凸凸的函數，像是 $y=x^2$ 之類的東西的話，那我們可以用類似二分搜的方式找到這個函數的極值點。通常來說，如果已知一個函數在區間 $[l,r]$ 內是先遞減再遞增，那通常可以先設兩個點 $m_1=\frac{l+l+r}{3}$ 跟 $m_2=\frac{l+r+r}{3}$，然後比較這兩個點的函數值。如果 $f(m_1)\leq f(m_2)$，那就把 r 更新成 m_2，反之把 l 更新成 m_1。

Algorithm 5: Ternary search

```
1 double TernarySearch(double l,double r){
2     assert(r>=l);
3     while(r-l>eps){
4         double m1=(l+l+r)/3,m2=(l+r+r)/3;
5         if(f(m1)<f(m2))r=m2;
6         else l=m1;
7     }
8     return l;
9 }
```

4 對答案二分搜

如果對於一個題目，可以知道類似「當代價為 x 的時候，我能否達成條件」這件事的話，且這個問題的答案有單調性，那我們就可以先設一個答案的可能範圍，然後對這個範圍去二分搜。如果達成了題目原本的條件，那代表代價可以再降低，反之必須提升，那就依照更新去調整 l,r 就好了。

5 例題們

1. TIOJ 1242
2. TIOJ 1337