Dynamic Programming

ss1h2a3tw

Outline

1. What is Dynamic Programming
2. Fibonacci
3. Rod Cutting
4. LCS
5. 0/1 Knapsack Problem
6. LIS
7. Bottom-Top , Top-Bottom
8. The way of thinking in DP

What is Dynamic Programming

Must have these 2 attributes

Optimal Substructure (最優子結構)

大問題的答案可以透過小問題的答案湊出來

Overlapping Subproblem (重複子問題)

有很多重複的小問題

Fibonacci

```
f(1)=1

f(2)=2

f(n)=f(n-1)+f(n-2)


1,1,2,3,5,8,13,21
```

Fibonacci

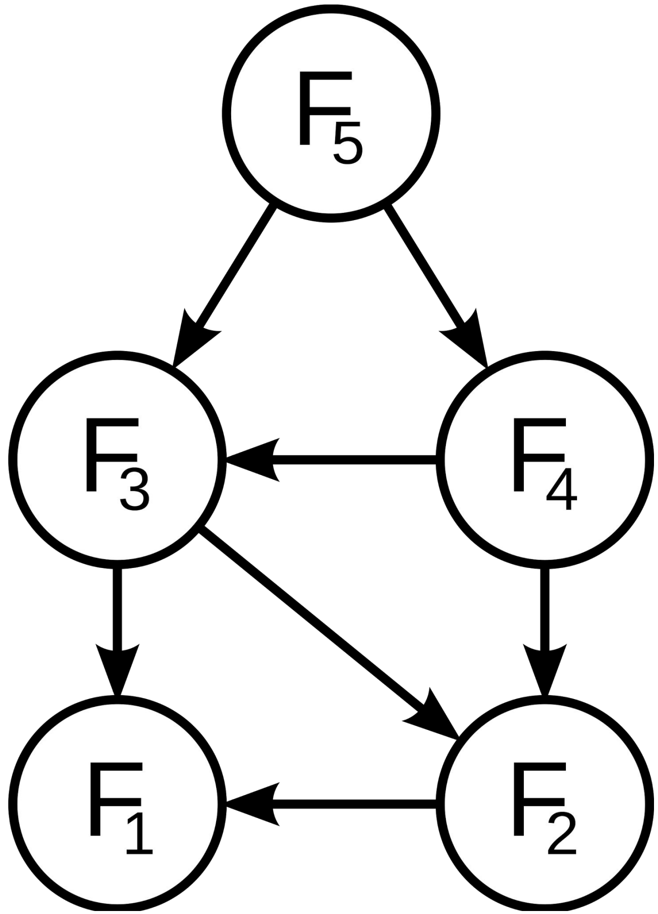
f(1)=1

f(2)=2

f(n)=f(n-1)+f(n-2) **Optimal Substructure**

1,1,2,3,5,8,13,21

Fibonacci-Overlapping Subproblem



https://upload.wikimedia.org/wikipedia/commons/0/06/Fibonacci_dynamic_programming.svg

Fibonacci - code

```
int fib[10];

fib[1]=1;

fib[2]=1;

for(int i = 3 ; i<10 ; i ++)

    fib[i]=fib[i-1]+fib[i-2];
```

Rod cutting - UVA 10003 Cutting Sticks

你想切木條,可是切木條的成本跟那個木條的長度一樣(無論切哪裡都一樣)

今天給你木條域定要切割的位置,求最小成本

木條長度<1000 , 要切的位置是整數,切的位置數量<50

https://pastebin.com/S5Q3TLsS

Rod cutting-Optimal Substructure

先把切玩後的小木條上編號 0,...,n

定義len(i,j) 為 [i..j] 的長度和

定義f(i,j) 為假設要處理[i..j]連續接起來的區間木條所需的最小成本

對於所有合法i f(i,i) = 0

f(i,j) = 對於所有i<=k<j 取

f(i,k)+f(k+1,j)+len(i,j) 的min

Optimal Substructure

Rod cutting-Overlapping Subproblem

`f(i,j)` 會需要 `f(i,i)`到`f(i,j-1)`與 `f(j,j)`到`f(i+1,j)`

`f(i+1,j)` 會需要 `f(i+1,i+1)`到`f(i+1,j-1)` 與 `f(j,j)`到`f(i+2,j)`

Rod cutting-code

```
for(int d = 1 ; d <= n ; d ++){
    for(int i = 0 ; i+d <= n ; i ++){
        dp[i][i+d]=1e9;
        for(int j = i ; j < i+d ; j ++){
            dp[i][i+d]=min(dp[i][j]+dp[j+1][i+d],dp[i][i+d]);
        }
        dp[i][i+d]+=psum[i+d+1]-psum[i];
    }
}
```

Longest Common Subsequence

最長共同子序列

子序列:一個序列中,照順序選擇子元素所產生的序列

[a,b,c,d,e,f,g]

[a,e,g] 是他的子序列

[d,c,f] 不是

給兩個序列 詢問他們的最長共同子序列

LCS-Optimal Substructure

想像有兩個序列 X=[......,a] 與 Y=[.......,b]

他們的LCS一定不是 [.......,a] 與 [.....] 一樣 或者是 [........] , [.......,b]

因為a,b不同 所以假設X中的a會跟Y中的某a配對,那麼某a的後面必定不能取

所以可以拔掉在Y尾巴的b

反之亦然

讓我們定義X,Y分別是第一序列從頭到第i項的子序列 , Y則是第二序列到第j項

```
if(a[i]!=b[j])
    f(i,j) = max(f(i-1,j),f(i,j-1))
```

LCS-Optimal Substructure

如果一樣呢？

[........,a] [..........,a] 的結果就是 [.......] [.......] 的結果+1

　　因為如果這著時候不湊一對,可能試想說還有其他機會

　　但是就算又其他機會也只能+1,不如現在就湊成一對,兩邊的還有+1的機會更大

```
if(a[i]==b[j])
    f(i,j) = f(i-1,j-1)+1
```

LCS-Overlapping Subproblem

`f(i,j)` 有可能會被 `f(i+1,j)` `f(i,j+1)` `f(i+1,j+1)`用到

LCS-code

```
for(int i = 0; i < lena ; i ++){
    for(int j = 0 ; j < lenb ; j ++){
        if(a[i]==b[j]){
            dp[i+1][j+1]=dp[i][j]+1;
        }
        else{
            dp[i+1][j+1]=max(dp[i+1][j],dp[i][j+1])
        }
    }
}
```

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	**0**	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	**1**	1	1	1	1	1	1
3	J	0	1	1	**2**	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	**3**	3	3	3
6	U	0	1	1	2	3	3	3	**4**
7	Z	0	1	2	2	3	3	3	4

0/1 Knapsack Problem

給若干個物品每個物品都有他的重量與價值每個都只有一個,給你一個可以裝某重量的背包,請問可以最多裝總共多少價值進去?

0/1 Knapsack Problem - Optimal Substructure

先把物品編號0....n-1

定義f(i,j) 為在0...i中選擇一些物品且重量總和不大於j的最大價值

f(i,j) = max(f(i-1,j),f(i-1,j-w[i])+v[i])

0/1 Knapsack Problem - Overlapping Subproblem

`f(i,j)` 會被 `f(i+1,j)` `f(i+1,j+w[i+1])` 用到

0/1 Knapsack Problem - Code

```
for(int i = 0 ; i < n ; i ++){

    for(int j = 0; j <= cap ; j ++){

        if(j-w[i]>=0)

            dp[i+1][j]=max(dp[i][j],dp[i][j-w[i]]+v[i]);

        else

            dp[i+1][j]=dp[i][j];

    }

}
```

Longest Increasing Subsequence

跟之前的子序列的定義一樣

但是這次裡面裝的是數字

求最常的遞增子序列

LIS-Optimal Substructure

定義f(i)為當一定要拿第i項時 0~i的LIS長度

f(0) = 1

f(i) = max(1,f(j)+1) 對於所有j<i且a[j]<a[i]

ans=max(f(i)) 0<=i<n

LIS-Overlapping Subproblem

`f(i)`會被所有`j>i`且`a[j]>a[i]`用到

LIS-Code

```
for(int i = 0 ; i < n ; i ++){

    dp[i]=1;

    for(int j = 0 ; j < i ; j ++){

        if(a[i]>a[j])dp[i]=max(dp[i],dp[j]+1);

    }

}
```

Bottom-Top , Top-Bottom

我們從剛剛到現在都是在填陣列

就是從小的case填到大的

有時反過來從大的開始會比較好寫

可是不是要先有小的嘛？

沒關係 我們來進行遞迴呼叫即可,再加上紀錄化搜索

Rod Cutting

```
int mem[N][N];

int psum[N+1]; // Assume we initailized it

int solve(int s,int e){

     if(s==e)return 0;

     if(mem[s][e])return mem[s][e];

     mem[s][e]=1e9;

     for(int i = s ; i < e ; i ++){

          mem[s][e]=min(solve(s,i)+solve(i+1,e)+mem[s][e]);

     }

     mem[s][e]+=psum[e+1]-psum[s];

     return mem[s][e];

}
```

The Way of Thinking in DP

找出子問題,列舉切割法

看測資範圍,觀察在複雜度上合理的DP方式

在切割子問題時,有時給子問題加上好的限制會讓DP式很漂亮