Dynamic Programming I

Michael Tsai

2013/10/3

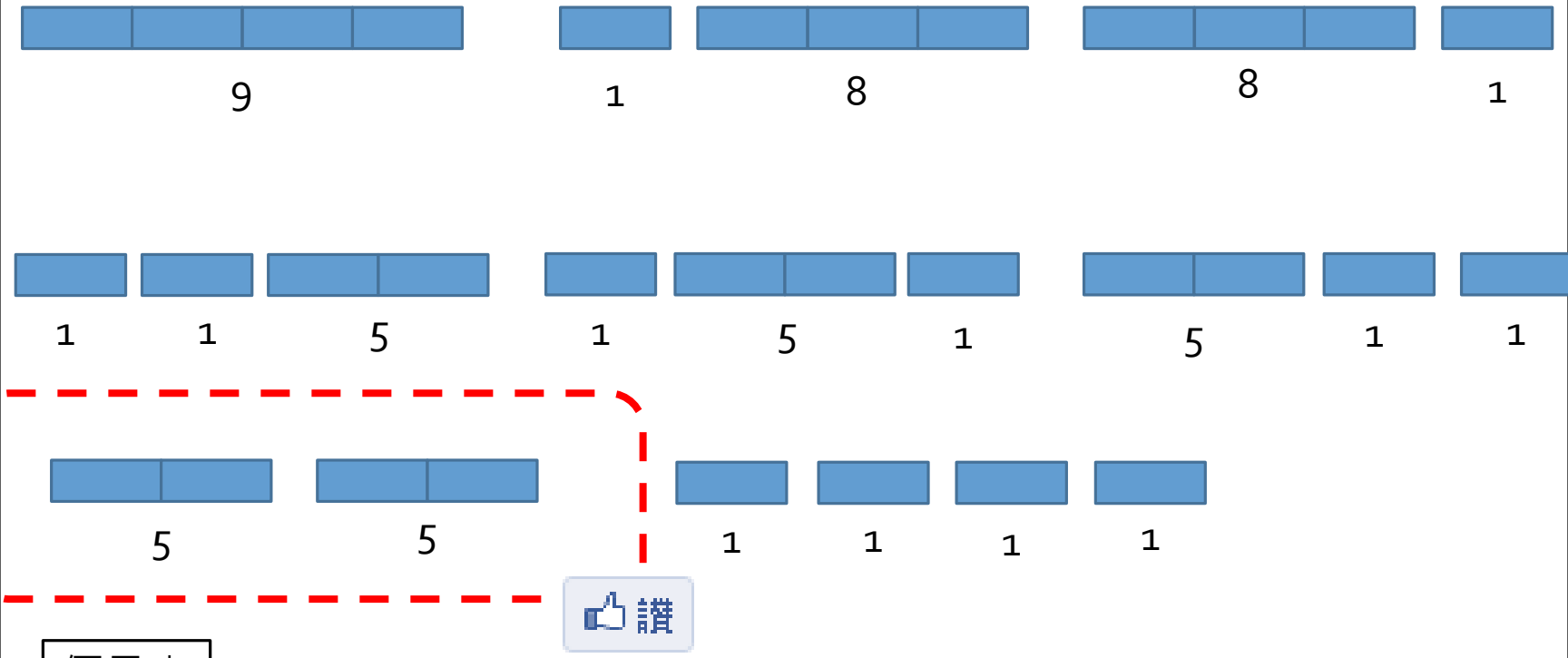
台大資訊鐵條回收公司

- 回收鐵條, 依長度算錢.
- 假設你有一條很長的鐵條, 長度為n (n為正整數)
- 可以把長鐵條砍成長度為整數的短鐵條
- 請問如何砍才能夠用這條鐵條換到最多錢?

價目表

長度	1	2	3	4	5	6	7	8	9	10
價錢	1	5	8	9	10	17	17	20	24	30

長度為4的鐵條



價目表

長度	1	2	3	4	5	6	7	8	9	10
價錢	1	5	8	9	10	17	17	20	24	30

長度為n的鐵條

- 長度為n的鐵條, 共有幾種要檢查的?
- 每一整數的地方都可以選擇要切, 或不切
- 共有n-1個"缺口"
- 因此需要檢查2^{n-1}種組合
- $\Theta(2^{n-1})$

長度為n的鐵條

- 讓情況變少種一點
- 如果我們規定鐵條只能越砍越大或一樣, 則需要檢查的不同方式可用partition function來表示
- 大約為$\frac{e^{\pi\sqrt{\frac{2n}{3}}}}{4n\sqrt{3}}$種

- 仍然為指數(比任何n的多項式都長得快)

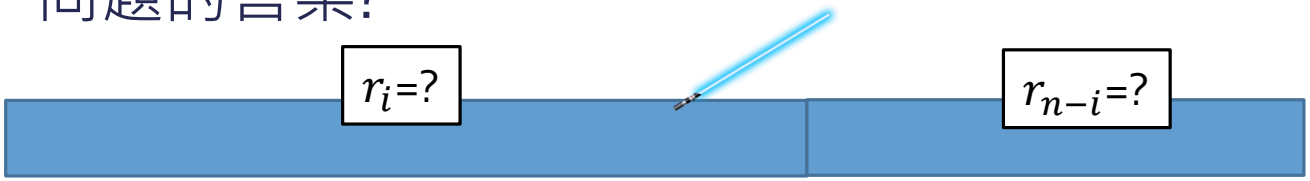
定義一些符號

- p_i: 長度為i的鐵條賣價(不再繼續砍)
- r_n: 長度為n的鐵條的最高賣價(最佳化砍法)

- 如果長度為n的鐵條最佳化砍法把鐵條砍成k段
- $n = i_1 + i_2 + \cdots + i_k$
- $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$

類divide-and-conquer思維

- "我請我的遞迴朋友們來解大小較小但同樣的問題"
- "假設我知道比較小的問題答案, 如何得出這個問題的答案?"



$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

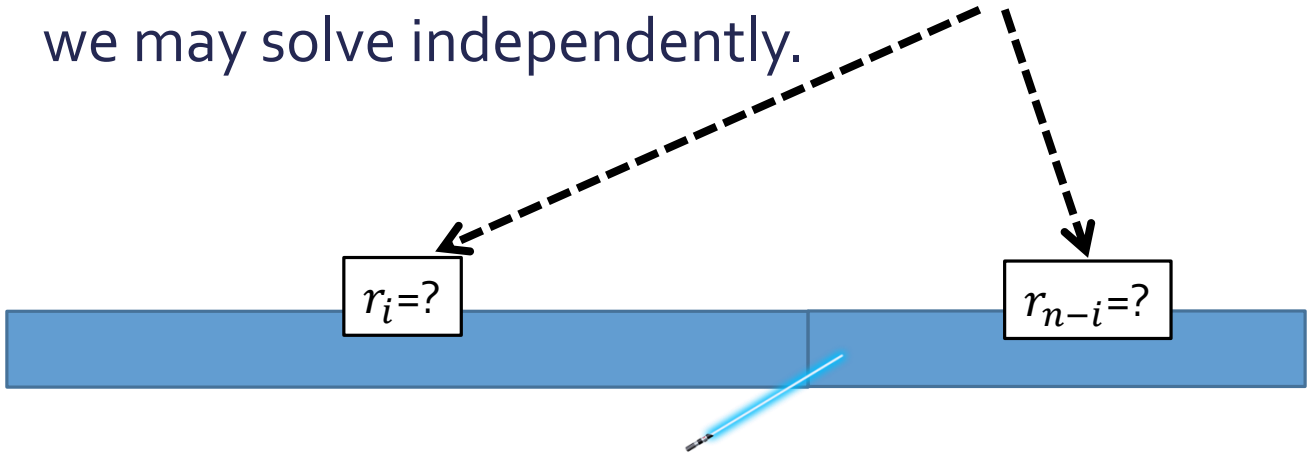
不砍刀

這一刀砍在左邊數過來第i個

最後再看哪個砍法為可得到最大值即為解.

Optimal Substructure

- Optimal solutions to a problem incorporate optimal solutions to **related subproblems**, which we may solve independently.



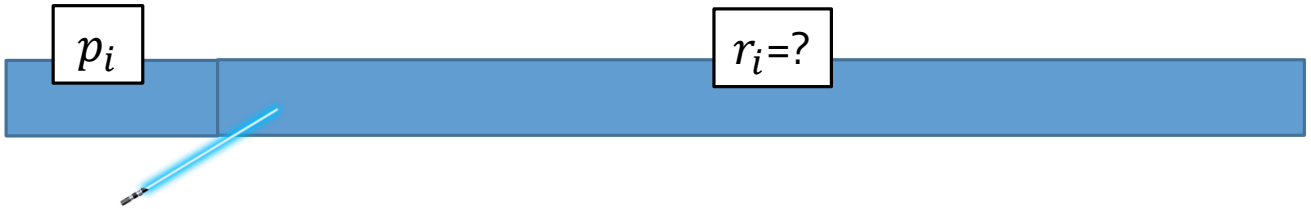
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

稍微調整一下

- 把subproblem數目變少一點

這一段砍下來就不再砍
成更小的了(拿去賣)

這一段是subprogram, 找遞迴朋友去解

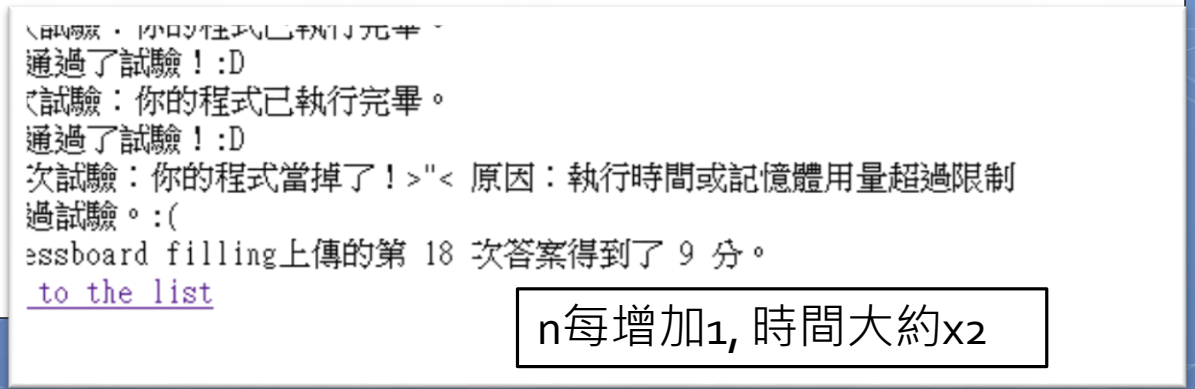


$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Cut-Rod v0.1 alpha

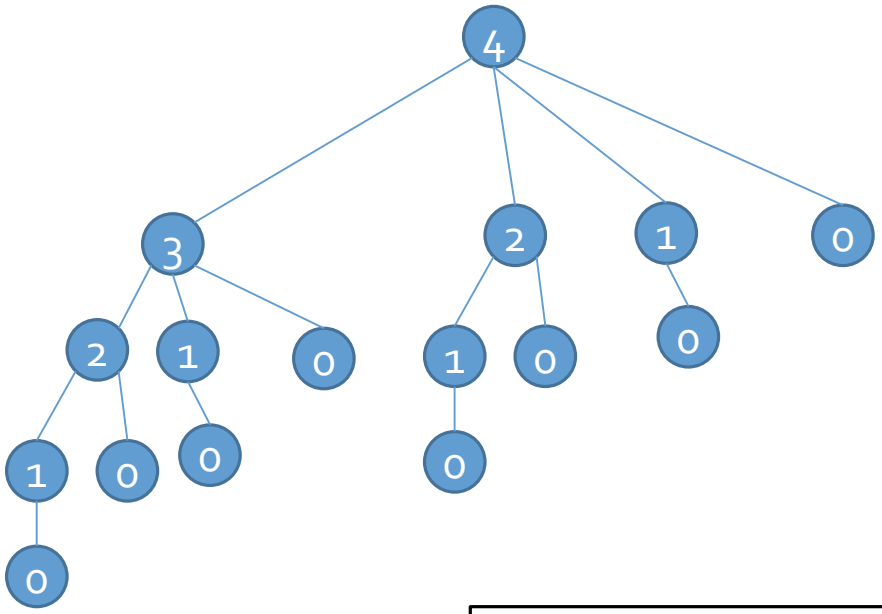
```
Cut_Rod(p,n)
if n==0
     return 0
q=-∞
for i=1 to n
     q=max(q, p[i]+Cut_Rod(p,n-i))
return q
```

猜猜看, 她會喜歡嗎?



n每增加1, 時間大約x2

Recursion tree



同樣的事情做那麼多遍...浪費時間!

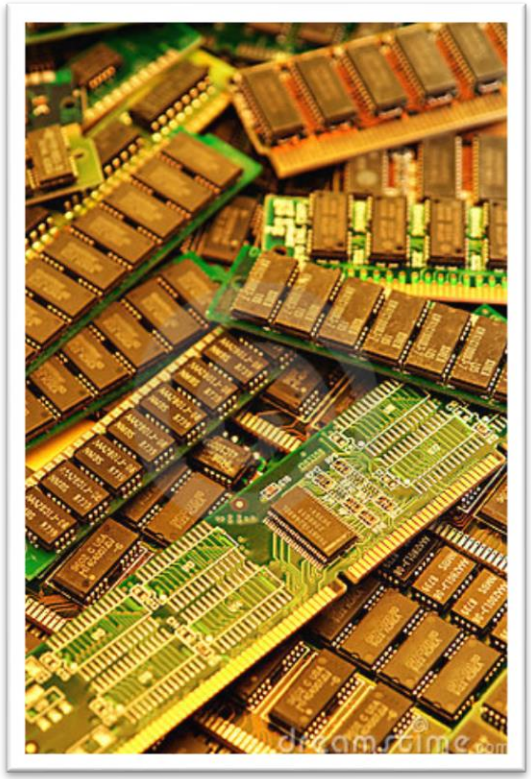
遞迴式又來拉~

- $T(0) = 1$
- $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$
- 結果:
- $\mathrm{T}(n) = 2^n$ (如我們所料)
- <Homework when you have time> :P
- Exercise 15.1-1

假如我們有點記性的話...

- 空間換取時間
- 做過的事情就不要再做
- "記得結果"就好: 填表與查表
- Dynamic programming: 這裡的programming指填表, 不是寫程式
- 當
  - 所有不同的subprogram數目是polynomial, 及
  - 解掉subprogram的時間也是polynomial,

dynamic programming方法可以在polynomial time內完成



Dynamic programming的做法

- Top-down with memoization
  - 還是用遞迴的方式來做
  - 但是每次做之前就先檢查是不是做過一樣的
  - 如果沒做過就遞迴下去, 但是要記錄結果
  - 如果做過就用查表取得之前的結果
- Bottom-up method
  - 把所有的subprogram從小排到大
  - 然後從小排到大來解
  - 解一個subproblem時候, 所有它所需要的subsubproblem都已經被解完了
- 請問哪一個快?
- 課本p.365有答案.

Cut-Rod v0.1 beta (Top-down)

```
Memoized_Cut_Rod(p,n)
let r[0..n] be a new array
for i=0 to n
      r[i]=-∞
return Memoized_Cut_Rod_Aux(p,n,r)
```

p: 價目表 n: 鐵條長度

```
Memoized_Cut_Rod_Aux(p,n,r)
if r[n] ≥ 0
      return r[n]
if n==0
      q=0
else q=-∞
      for i=1 to n
            q=max(q,p[i]+Memoized_Cut_Rod_Aux(p,n-i,r))
r[n]=q
return q
```

p: 價目表 n: 鐵條長度
r: 存最佳解的陣列

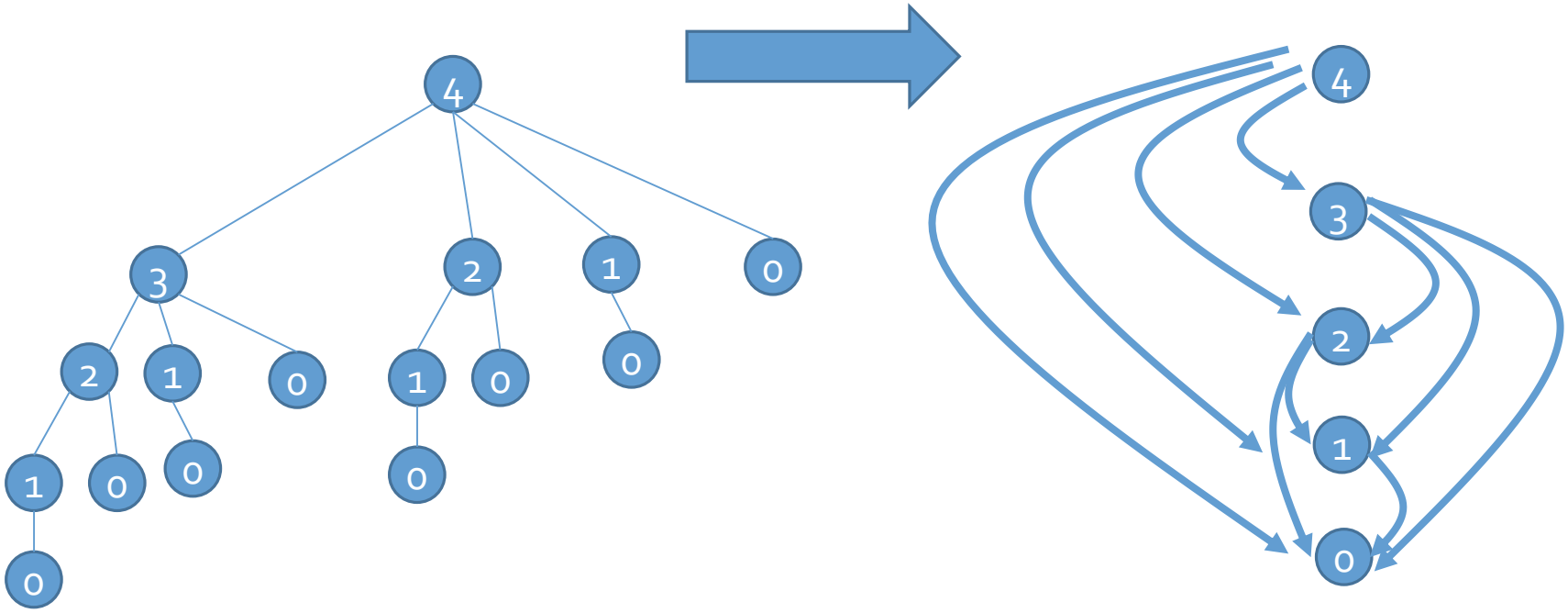
Cut-Rod v0.1 gamma (Bottom-up)

```
Bottom_Up_Cut_Rod(p,n)
let r[0..n] be a new array
r[0]=0
for j=1 to n
     q=-∞
     for i=1 to j
          q=max(q,p[i]+r[j-i])
     r[j]=q
return r[n]
```

$\Theta(n^2)$

Subproblem graphs

Subproblem x必須要考慮subproblem之optimal solution→ <x,y> directed edge



Bottom-up method: Reverse Topological Sort

Top-down method: Depth First Search

Subproblem graphs

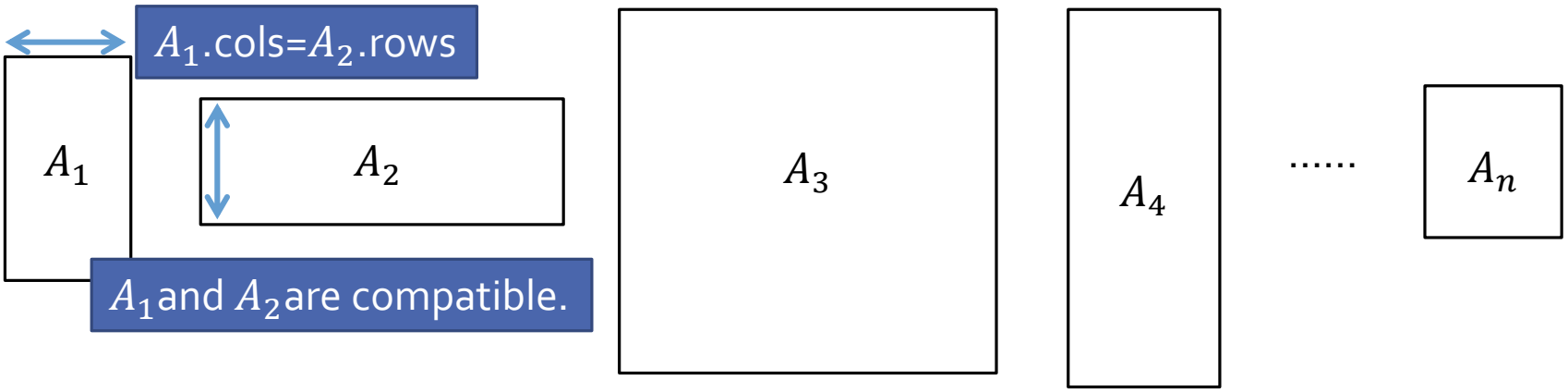
- Subproblem graph的大小讓我們可以估計dynamic programming演算法的執行時間
- 大小?
- $G = (V, E)$. 我們指$|E|$和$|V|$.
- $|E|$=有幾個要解的subproblem (因為有記結果, 所以一個subproblem只要做一次)
- $|V|$=每個subproblem需要幾個比它小的subproblem的結果
- 大致的估計: 和$|E|$和$|V|$成線性

最後的小問題

- 剛剛的程式只把可以拿多少錢算出來
- 沒有真的印出要怎麼切
- 如何解決?
- 用另外一個陣列(大小$\Theta(n)$)紀錄optimal solution應該切的大小.
- 最後依序印出結果即可.
- 課本p. 369有解答.

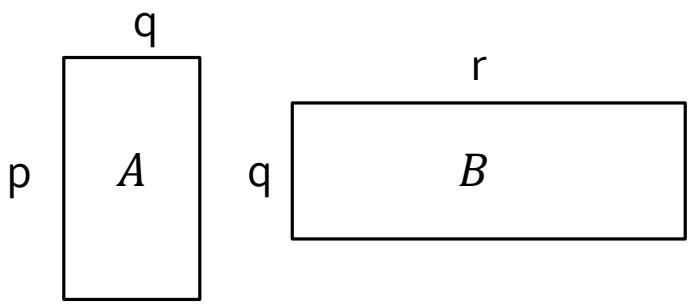
連串矩陣相乘問題

題目: 求$A_1A_2 \ldots A_n$矩陣相乘之解.



- Matrix multiplication is associative.
- $(((A_1A_2)A_3)A_4)$
- $((A_1(A_2A_3))A_4)$
- $((A_1A_2)(A_3A_4))$
- $(A_1((A_2A_3)A_4))$
- $(A_1(A_2(A_3A_4)))$
- 以上算出來答案都一樣

連串矩陣相乘問題

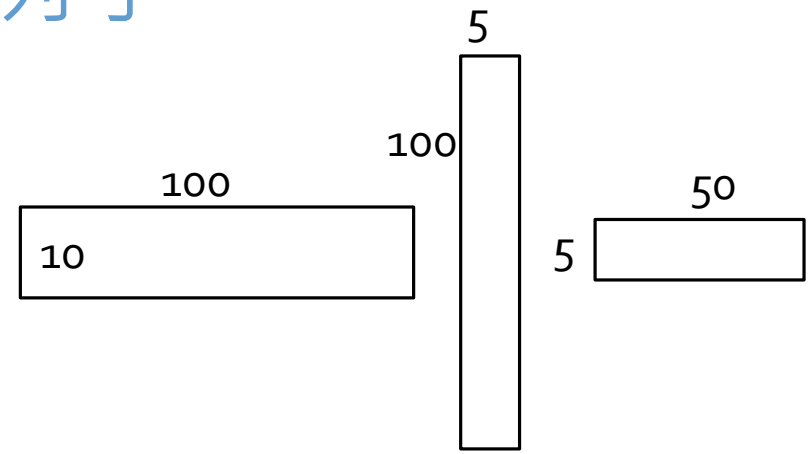



```
Matrix-Multiply(A,B)
if A.columns != B.rows
      error “incompatible dimensions”
else let C be a new A.rows x B.cols matrix
      for i=1 to A.rows
            for j=1 to B.cols
                  c_ij=0
                  for k=1 to A.cols
                        c_ij = c_ij + a_ik · b_kj
      return C
```

主要花費時間在這邊!

共花費pqr次乘法的時間

看一個例子



- $((A_1A_2)A_3)$花費之乘法數目=$10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$
- $(A_1(A_2A_3))$花費之乘法數目=$100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$
- 差十倍!!

連串矩陣相乘問題-正式版

- 給一連串的矩陣$\langle A_1, A_2, \ldots, A_n \rangle$, 其中矩陣$A_i$的大小為$p_{i-1} \times p_i \ (i = 1, 2, \ldots, n)$, 找出一種乘法可以使計算時的乘法數目最少
- 沒有真的要算結果, 而只是找出能最快算出結果的方法.
- 因為算"怎麼算比較快"多花的時間, 比"用爛方法直接算"多花的時間少很多

暴力法有多暴力

- 全部到底有幾種算法呢?
- 用遞迴定義:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

假設先這樣分:$(A_1 A_2 \dots A_k)(A_{k+1} A_{k+2} \dots A_n)$

- (n)之解為Catalan numbers, $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$, or is also $\Omega(2^n)$

所以不要暴力了.

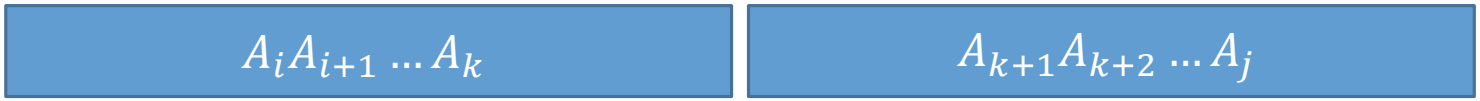
- 使用dynamic programming
- 正規步驟:

1. 找出最佳解的"結構"
2. 使用遞迴來定義最佳解的花費
3. 計算最佳解的花費
4. 使用已經計算的資訊來構築最佳解

找出最佳解的”結構”

$i \le j$

在k切一刀

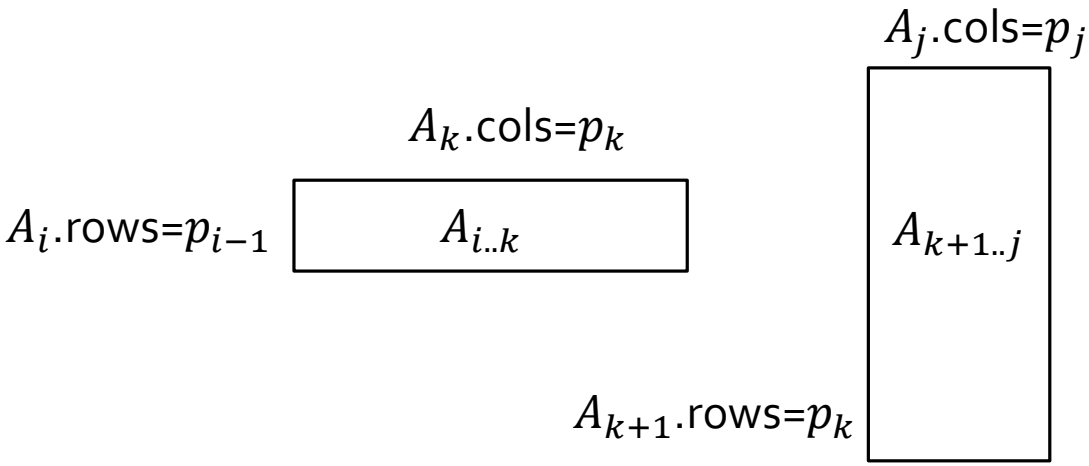


$i \le k < j$

- 總花費=$A_i..A_k$的花費+$A_{k+1}..A_j$的花費+把$A_i..A_k$和$A_{k+1}..A_j$乘起來的花費
- 最佳解的結構: 假設有$A_i..A_j$的最佳解, 此一方法為在k切一刀. 則在此$A_i..A_j$最佳解中, $A_i..A_k$的相乘方法一定是$A_i..A_k$的最佳相乘方法
- 假設$A_i..A_k$不是最佳解, 則我們可以在$A_i..A_j$中把$A_i..A_k$換成更好的方法, 則可以找到一個更好的$A_i..A_j$的相乘方法→矛盾.
- 子問題的最佳解可以導出大問題的最佳解!
- 最後結論: 分成兩個子問題, 並嘗試所有可以切分的地方(k值)

使用遞迴來定義最佳解的花費

- 遞迴定義:使用子問題最佳解的cost來定義大問題最佳解的cost
- 定義: $m[i,j]$為$A_i..A_j$所需花的最少乘法數
- $A_{i..k}A_{k+1..j}$所花乘法數=$p_{i-1}p_kp_j$



使用遞迴來定義最佳解的花費

- $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
- 但是我們不知道最佳解中的k的值
- 因此我們必須找所有$k = i, i+1, \ldots, j-1$
- 最後版本:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j. \end{cases}$$

計算最佳解的花費

- 純recursive解法: exponential time
- 使用前面教的Bottom-up填表方法: 每個不同的subprogram僅需解1次
- 有幾個不同的問題?
- $1 \le i \le j \le n$, 幾個i和j的組合?
- 答案: $\binom{n}{2} + \mathrm{n} = \Theta(n^2)$

$i \neq j$ $i = j$

計算最佳解的花費

- 如何決定填表的順序呢? (這次有i,j兩個變數)

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j. \end{cases}$$

$k - i + 1$個矩陣相乘　　$j - k$個矩陣相乘

都小於$j - i + 1$個

我們可以把j-i+1(也就是要相乘的matrix個數)
當作problem size的定義

計算最佳解的花費

```
n=p.length-1
let m[1..n,1..n] and s[1..n-1,2..n] be new tables
for i=1 to n
      m[i,i]=0
for l=2 to n
      for i=1 to n-l+1
            j=i+l-1
            m[i,j]=∞
            for k=i to j-1
                  q=m[i,k]+m[k+1,j]+p_{i-1} p_k p_j
                  if q<m[i,j]
                        m[i,j]=q
                        s[i,j]=k
return m and s
```

邊界條件先設好.

大problem的解只會用到小problem的解.因此慢慢往上長.

把同樣problem size的所有i,j組合都依序做過

使用遞迴式找出最佳切點k

$\Theta(n^3)$

計算最佳解的花費

- 用一個例子來trace code比較快

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 x 35	35 x 15	15 x 5	5 x 10	10 x 20	20 x 25

使用已經計算的資訊來構築最佳解

- 前面只印出了花費, 不是真正的解
- 怎麼乘才是最後的解
- 使用s陣列的資訊

使用已經計算的資訊來構築最佳解

```
Print-Optimal-Parens(s,i,j)
if i==j
      print A_i
else
      print "("
      Print-Optimal-Parens(s,i,s[i,j])
      Print-Optimal-Parens(s,s[i,j]+1,j)
      print ")"
```

看了兩個例子以後...

- 問: 一個問題要有什麼要件才能使用dynamic programming?
- 答:

1. Optimal substructure
2. Overlapping Subproblems

什麼是Optimal substructure?

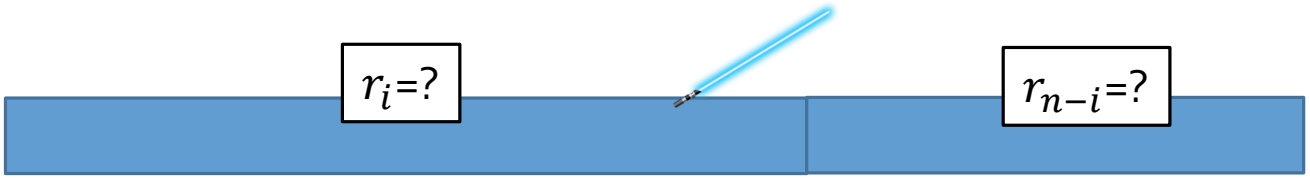
- Definition: A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems.

- 怎麼尋找optimal substructure呢?

怎麼尋找optimal substructure呢?

1. 要得到問題的解答有許多選擇(砍在哪邊, 切在哪邊), 而做這個選擇之後, 我們有一些subproblem要解決.
2. 我們假設對於一個問題, 我們可以找到那個選擇
3. 知道這個選擇以後, 我們找出哪個subproblem可以被拿來應用, 及剩下的問題(沒有對應到subproblem的)怎麼解決
4. 證明大問題的最佳解中可以直接應用(剪下貼上)子問題的最佳解.

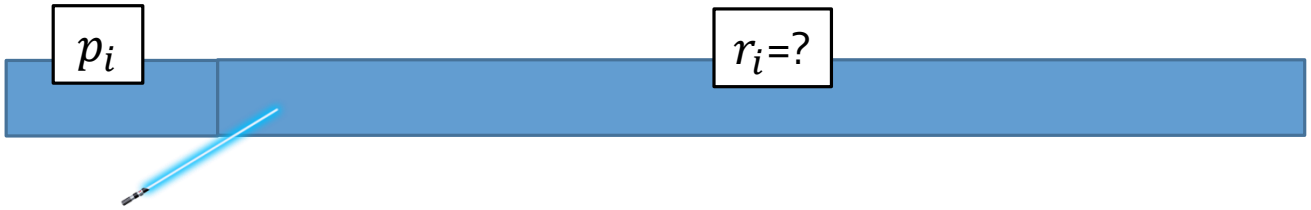
Optimal substructure越簡單越好



versus

這一段砍下來就不再砍成更小的了(拿去賣)

這一段是subproblem, 找遞迴朋友去解



Optimal substructure越簡單越好

假設把問題定義成$A_1 \dots A_j$就好 (少一個變數)

$1 \le j$

在k切一刀

$A_1 A_{i+1} \dots A_k$ | $A_{k+1} A_{k+2} \dots A_j$

$1 \le k < j$

此為一個子問題

此不為一個子問題!

除非k一直都是j-1, 否則...

Optimal substructure的變化

1. 原始問題的最佳解用了多少個子問題
2. 大問題有多少選擇(選擇用不同的子問題們來獲得最佳解)

大略來說, 以上兩者決定dynamic programming algorithm的執行時間.

(之前說的Subproblem graphs是另外一種算法)

	多少個子問題	有多少選擇	執行時間
鐵條資源回收	$\Theta(n)$	n	$O(n^2)$
連串矩陣相乘	$\Theta(n^2)$	n-1	$O(n^3)$