アルゴリズム論（第10回）

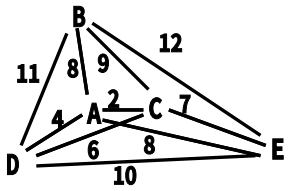
2003.12.15
櫻井　彰人
Algorithm@soft.ae.keio.ac.jp
http://www.sakurai.comp.ae.keio.ac.jp/

きょうの講義概要

- グラフ（3）（復習）
  - グラフのアルゴリズム
  - フロイトの方法
  - グラフ理論からの補足
- 動的計画法
  - LCS : 最長共通系列
  - 0-1 ナップザック問題 (次回)

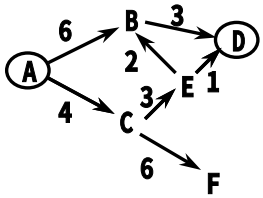
5.5 クラスカルのアルゴリズム

- すべての地点を結ぶ無向グラフでコストが最小になるもの（コスト最小の全域木）を求める（道路網、電話網など）



5.6 ダイクストラ法

- 各辺に距離のついた有向グラフ上で任意の2点間の最短経路を見つける



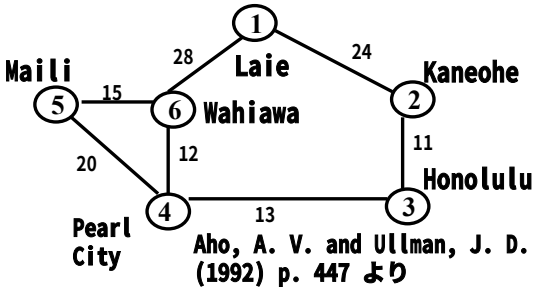
考え方：
出発の節から近い節を順次たどる。結果的には、指定された出発の節から全部の節への最短経路が求められる。

5.6 フロイトのアルゴリズム

- 特定の出発の節を決めず、すべての節間の最短距離を求める
  - すべての節に対してダイクストラ法を適用しても求まる
    - » 計算量が $O(n^3)$ に
- $a_k(i,j)$ を、$a_1, a_2, ..., a_n$ と求める
  - $a_k(i,j)$ = k番以下の節だけを経由して i から j へ至る最短経路の距離（k番目の節：ピボット）

例題

- ハワイのオアフ島の町をめぐる

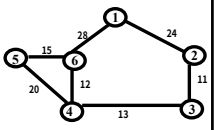


Aho, A. V. and Ullman, J. D. (1992) p. 447 より

例題の行列表現

■ はじめの行列の (i,j) 要素
– i から j へ直接つながる辺のラベル（距離）

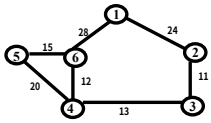
	1	2	3	4	5	6
1	0	24	∞	∞	∞	28
2	24	0	11	∞	∞	∞
3	∞	11	0	13	∞	∞
4	∞	∞	13	0	20	12
5	∞	∞	∞	20	0	15
6	28	∞	∞	12	15	0



例題の実行（1）

■ 1の節だけを経由してよいときの行列
– 1の節を経由点に

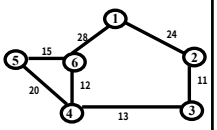
	1	2	3	4	5	6
1	0	24	∞	∞	∞	28
2	24	0	11	∞	∞	52
3	∞	11	0	13	∞	∞
4	∞	∞	13	0	20	12
5	∞	∞	∞	20	0	15
6	28	52	∞	12	15	0



例題の実行（2-1）

■ 1と2の節を経由してよいときの行列
– 2の節を新しい経由点に

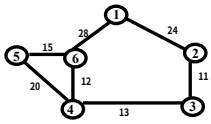
	1	2	3	4	5	6
1	0	24	35	∞	∞	28
2	24	0	11	∞	∞	52
3	35	11	0	13	∞	∞
4	∞	∞	13	0	20	12
5	∞	∞	∞	20	0	15
6	28	52	∞	12	15	0



例題の実行（2-2）

■ 1と2の節を経由してよいときの行列
– 2の節を新しい経由点に

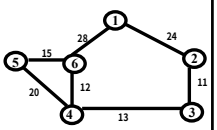
	1	2	3	4	5	6
1	0	24	35	∞	∞	28
2	24	0	11	∞	∞	52
3	35	11	0	13	∞	63
4	∞	∞	13	0	20	12
5	∞	∞	∞	20	0	15
6	28	52	63	12	15	0



例題の実行（3）

■ 1、2、3の節を経由してよいときの行列
– 3の節を経由点に

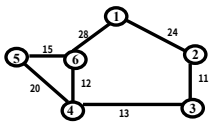
	1	2	3	4	5	6
1	0	24	35	48	∞	28
2	24	0	11	24	∞	52
3	35	11	0	13	∞	63
4	48	24	13	0	20	12
5	∞	∞	∞	20	0	15
6	28	52	63	12	15	0



例題の実行（4）

■ 1、2、3、4の節を経由してよいときの行列
– 4の節を新たな経由点に

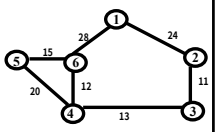
	1	2	3	4	5	6
1	0	24	35	48	68	28
2	24	0	11	24	44	36
3	35	11	0	13	33	25
4	48	24	13	0	20	12
5	68	44	33	20	0	15
6	28	36	25	12	15	0



例題の実行（5）

■ 最終的な行列

	1	2	3	4	5	6
1	0	24	35	40	43	28
2	24	0	11	24	44	36
3	35	11	0	13	33	25
4	40	24	13	0	20	12
5	43	44	33	20	0	15
6	28	36	25	12	15	0



アルゴリズム5.3 フロイトのアルゴリズム

■ 入力：グラフのn個の節間の距離を表わすn×n行列
■ 出力：各節間の最短距離を表わす行列
■ 内容
■ （1）v = 1 から n まで以下を繰り返す
– (1.1) w = 1 から n まで以下を繰り返す
» (1.1.1) dist(v,w) ← arc(v,w)

フロイトのアルゴリズム（2）

◆ (2) u = 1 から n まで以下を繰り返す
– (2.1) v = 1 から n まで以下を繰り返す
» (2.1.1) w = 1 から n まで以下を繰り返す
◆ (2.1.1.1) dist(v,u) + dist(u,w) < dist(v,w)ならば dist(v,w) ← dist(v,u) + dist(u,w)

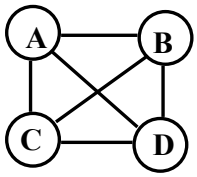
■ dist(v,w)：節v から節w までの距離
■ arc(v,w)：節v から節w への直接つながった辺の距離（入力行列の(v,w)要素）

フロイトのアルゴリズムの正当性

■ アルゴリズム5.3 において、ピボットを通るすべての経路でdist(v,w) は 節v と 節w の最短距離（証明は数学的帰納法）
■ k以下の節だけを中間の節とする v から w への経路：k-path
■ 命題S(k)：アルゴリズム5.3の (2) のループに関して、u が k のループ終了のときに、dist(v,w) は v から w への最短の k-path であるか、または経路が存在せずに ∞

5.7 グラフ理論からの補足（1）

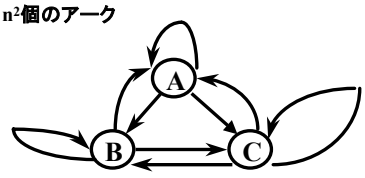
◆ 無向グラフが完全（complete）グラフ
– すべての節の間に辺がある
– n個の節の完全グラフ: K_n
» 辺の数は ${}_nC_2$（= n(n-1)/2）個



K_4

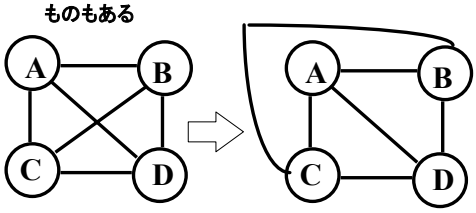
グラフ理論からの補足（2）

◆ 有向グラフが完全（complete）グラフ
– 自分自身を含めてすべての節を互いに結ぶ辺がある
– n個の節の完全グラフ
» n^2個のアーク



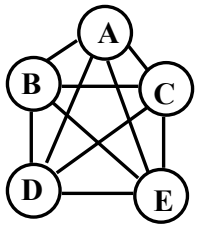
グラフ理論からの補足（3）

- 平面グラフ（A Planar Graph）
  - どの辺も交差しないで1平面上に書くことのできるグラフ
    - » 辺と節の関係を変えないで平面グラフで表現できるものもある

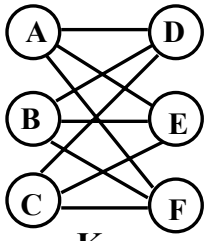


グラフ理論からの補足（4）

- 非平面グラフの例: K_5 と $K_{3,3}$



K_5



$K_{3,3}$

グラフ理論からの補足（5）

- クラトフスキー（Kuratowski）の定理
  - すべての非平面グラフは、前に示したK_5と$K_{3,3}$のうちの少なくとも一つの「コピー」を含む
- グラフの平面性の応用
  - 平面グラフと同等なグラフは、辺の交差をなくして表示したほうが分かりやすい
  - 集積回路の設計では、平面にできるものは1平面にしたほうが集積度を増すことができる

動的計画法 (Dynamic Programming)

- Bellman 1952;
  - 有名な著書 “Dynamic Programming” は1957. そこに “Dynamic” indicates that we are interested in processes
    - » in which time plays a significant role, and
    - » in which the order of operations may be crucial
  - “programming” は恐らく planning/scheduling
    - » In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, ‘programming.’ (Bellman 1984 “Eye of the hurricane”)
    - » "Program" at that time was a military term that referred not to the instruction used by a computer to solve problems, which were then called "codes," but rather to plans or proposed schedules for training, logistical supply, or deployment of combat units. (Professor George Dantzig: Linear Programming Founder Turns 80, SIAM News, November 1994,)

例: 最長共通部分系列問題

- 最長共通部分系列問題
  Longest common subsequence (LCS) problem:
  - 二つの系列 x[1..*m*] と y[1..*n*] が与えられた時, 双方に現れる部分系列の中で最長のものを見つけよ
  - 例: x = {A B C B D A B }, y = {B D C A B A}
  - {B C} や {A A} は双方の部分列
    - » ではそのような部分系列で最長のもの (LCS) は?

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

力ずくでは？

- 力ずく法: x の部分系列を一つずつ作り, それが y の部分系列になっているかを調べる
  - x の部分系列はいくつあるか?
  - 力ずく法の計算量は？
- x の部分系列は 2^m 個ある.
- その一つ一つを y の n 個の要素と照合する:
- $O(n\,2^m)$

x = {A B C B D A B }
A
B
A B
C
A C
BC
……

y = {B D C A B A}
D B
D B
D B
D
B

もっと賢くできないか？

- 実は良い方法がある: とりあえず, LCS の長さだけを見つける問題を考える
  - 見つけ終わったときに, この解からLCS の解を掘り出す方法を考える
- LCS 問題には部分構造の最適性 (*optimal substructure*) がある
  - 部分構造の最適性: 最適解が部分問題に対する最適解を含んでい ること. 例えば、AからZへの最短路はその経由地 BからYへの最短路を含んでいる
  - 部分問題: x と y の接頭辞 (*prefixes*) のLCS問題

接頭辞 (prefix)

- x[1..*m*] の接頭辞とは x[1..*j*] (1≤*j*≤*m*) .

  x = A **B** **C** **B** D **A** B

  y = **B** D **C** A **B** **A**

- 全体のLCSは
  - x[1..2] と y[1..1],
  - x[1..3] と y[1..3],
  - x[1..4] と y[1..5], 等

  のLCSを含む

LCS 長を見つけよう

- c[*i,j*] を x[1..*i*] と y[1..*j*] の LCS とする
  - そのとき、x と y のLCSの長さはどう表せるか?
- 定理:

$$c[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ c[i-1,j-1]+1 & \text{if } x[i]=y[j],\\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise}\end{cases}$$

- これはどんな意味か?

LCS の再帰解

$$c[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ \cdots & \\ \cdots & \end{cases}$$

- $i = j = 0$ から始める(x と y の空部分列)
- x[1..0] と y[1..0] は空列であるから, そのLCSは空列 (i.e. $c[0,0] = 0$)
- 空列と他のどんな列とのLCSは空列であるから, 全ての i と j について: $c[0, j] = c[i,0] = 0$

LCS の再帰解

$$c[i,j]=\begin{cases}\cdots & \\ c[i-1,j-1]+1 & \text{if } x[i]=y[j],\\ \cdots & \end{cases}$$

- $c[i, j]$ を計算するにあたって2つの場合がある:
- **場合1:** $x[i]=y[j]$: さらに一つの文字が一致, そこで、x[1..*i*] と y[1..*j*] のLCSの長さは, x[1..*i*–1] と y[1..*j*–1] のLCSの長さに1加えたもの

  x = A **B** **C** | **B** D **A** B

  y = **B** D **C** A | **B** **A**

LCS の再帰解

$$c[i,j]=\begin{cases}\cdots & \\ \cdots & \\ \max(\,c[i,j-1],\,c[i-1,j]\,) & \text{otherwise}\end{cases}$$

- **場合2:** $x[i] != y[j]$
- 二つの記号は一致しないので, 今持っている解は改善しない. すなわち, LCS(x[1..*i*], y[1..*j*]) の長さは以前と同じ (LCS(x[1..*i*–1], y[1..*j*]) の長さと LCS(x[1..*i*], y[1..*j*–1]) の長さの最大値). なぜ, LCS(x[1..*i*–1], y[1..*j*–1]) の長さではないのか？

x = A **B** **C** **B** D **A** B (*x*[3])

y = **B** D **C** A **B** **A** (*y*[4])

LCS(x[1..*3*], y[1..*4*])="BC"
LCS(x[1..2], y[1..*4*])="B"
LCS(x[1..*3*], y[1..*3*])="BC"
LCS(x[1..*2*], y[1..*3*])="B"

LCS の長さを求めるアルゴリズム

```
LCS-Length(x, y)
1. m = length(x)  // x の長さを求める
2. n  = length(y)  // y の長さを求める
3. for i = 1 to m      c[i,0] = 0      // y が空
4. for j = 1 to n      c[0,j] = 0      // x が空
5. for i = 1 to m                      // 全ての x[i] につき
6.       for j = 1 to n                        // 全 y[j]
7.              if ( x[i] == y[j])
8.                      c[i,j] = c[i-1,j-1] + 1
9.              else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

LCS の例

次の例に LCS長アルゴリズムを適用しよう:

- x = ABCB
- y = BDCAB

x と y のLCSは何か？

LCS(x, y) = BCB
x = A **B** **C** **B**
y = **B** D **C** A **B**

データは バージニア大 David Luebke

例 (0)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		y[j]	**B**	**D**	**C**	**A**	**B**
0	x[i]						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

x = ABCB; m = |x| = 4
y = BDCAB; n = |y| = 5
配列 c[0..5,0..4] の確保

例 (1)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		y[j]	**B**	**D**	**C**	**A**	**B**
0	x[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m c[i,0] = 0
for j = 1 to n c[0,j] = 0

例 (2)

ABCB
BDCAB

i	j	0	**1**	2	3	4	5
		y[j]	(**B**)	**D**	**C**	**A**	**B**
0	x[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	(**A**)	**0**	**0**				
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (3)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		y[j]	**B**	**D**	**C**	**A**	**B**
0	x[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (4)

ABCB
BDCAB

i	x[i]	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B	
0			0	0	0	0	0	0
1	A		0	0	0	0	1	
2	B		0					
3	C		0					
4	B		0					

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (5)

ABCB
BDCAB

i	x[i]	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B	
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0					
3	C		0					
4	B		0					

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (6)

ABCB
BDCAB

i	x[i]	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B	
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1				
3	C		0					
4	B		0					

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (7)

ABCB
BDCAB

i	x[i]	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B	
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	
3	C		0					
4	B		0					

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (8)

ABCB
BDCAB

i	x[i]	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B	
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0					
4	B		0					

```
if (x[i] == y[j])
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (10)

ABCB
BDCAB

i	x[i]	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B	
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1			
4	B		0					

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (11)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B
0	x[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2		
4	B	0					

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (12)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B
0	x[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0					

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (13)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B
0	x[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1				

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (14)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B
0	x[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

例 (15)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		y[j]	B	D	C	A	B
0	x[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

```
if (x[i] == y[j] )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

LCS 長アルゴリズムの実行時間

- LCS長アルゴリズムは配列 $c[m,n]$ の各要素を一度だけ計算する
- 従って、実行時間は

  $O(m*n)$

  というのも各 $c[i,j]$ の計算は一定時間だし, 要素数は $m*n$

LCSそのものの見つけ方

- 先ほどのアルゴリズムでは、LCS長はわかったがLCSそのものはわからない.
- しかしながら、ちょっとの工夫で、LCS自体を求めることができる.

各 $c[i,j]$ が依存するのは $c[i–1,j]$ と $c[i,j–1]$

とかまたは $c[i–1, j–1]$

各 c[i,j] についてどうやってそれを得たかがわかる:

2	2
2	3

例えば, ここでは
$c[i,j] = c[i–1,j–1] +1 = 2+1=3$

LCSそのものの見つけ方

- 再帰的定義は

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- そこで $c[m,n]$ から始めて戻ればよい
- $c[i,j] = c[i\text{-}1,j\text{-}1]+1$ ならばいつでも, $x[i]$ を記憶 (なぜなら $x[i]$ はLCSの一部分)
- i=0 または j=0 (i.e. すなわち端に辿り着いたとき), 記憶した文字列を逆順に出力

LCS の見つけ方

i	j	0	1	2	3	4	5
		y[j]	**B**	**D**	**C**	**A**	**B**
0	x[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

LCSの見つけ方 (2)

i	j	0	1	2	3	4	5
		y[j]	**B**	**D**	**C**	**A**	**B**
0	x[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

LCS (逆順): **B C B**

LCS (正順): **B C B**

（回文になったのは偶然）

動的計画法

- 解が、再帰的に、部分問題の解を用いて、表現できる (部分構造の最適性 *optimal substructure* がある) ときに使える
- 分割統治法 (*divide and conquer*) とそっくりであるが, 部分解を何回も再利用することで効率向上
  - 分割統治は、部分問題が共通でないときのアイデア
- すなわち、動的計画法のアルゴリズムは、部分問題の解を見つけるとそれを記憶し、次に同じ部分問題が表れると、それを使用する

まとめ

- 動的計画法 (dynamic programming) が非常に有効となる問題群がある
- 解が、部分問題の解から、再帰的に構成されるとき、これら部分解を記憶し、必要に応じて再利用する
- 実行時間比較 (動的計画法 vs. 力ずく):
  - LCS: $\mathbf{O(m*n)}$ vs. $\mathbf{O(n * 2^m)}$
  - 0-1 ナップザック問題: $\mathbf{O(W*n)}$ vs. $\mathbf{O(2^n)}$