アルゴリズム論（第11回）

2003.12.22
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

- **動的計画法**
  - **LCS : 最長共通系列（復習）**
  - **0-1 ナップザック問題**

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,)

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

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**

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

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

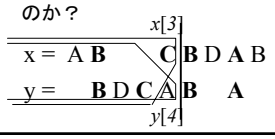
- $c[i,j]$ を計算するにあたって2つの場合がある:
- **場合1:** $x[i]=y[j]$: さらに一つの文字が一致, そこで、x[1..*i*] と y[1..*j*] のLCSの長さは, x[1..*i*–1] と y[1..*j*–1] のLCSの長さに1加えたもの

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?



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		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	
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		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					
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		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				
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		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	
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		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					
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		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			
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

- 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）

例題

- 次の文字列の最長部分列を求めなさい
  - abcdgh aedfhr
- 次の列の最長部分列を求めなさい

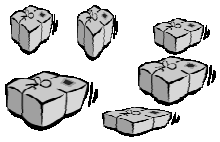
Sequence1: 

Sequence2: 

ナップザック問題 (Knapsack problem)

最大積載量のあるナップサックに荷物を詰める問題
・荷物はいくつかある
・荷物はそれぞれ重さが違う
・荷物はそれぞれ価値が違う
荷物の価値の合計が最大になる詰め込み方を求めよ
例:

荷物番号	重さ	価値
1	1	8
2	3	6
3	5	5



ナップザック問題

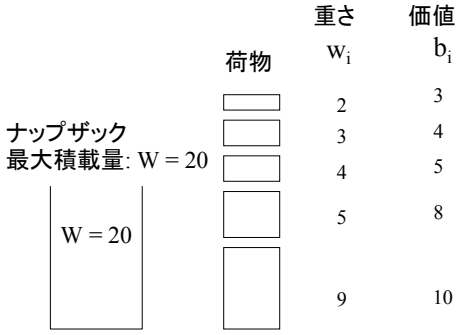
2種類ある:
(1) "0-1 ナップザック問題" と
(2) "実数ナップザック問題"

(1) 各アイテムは分割できない. すなわち、そのアイテムを取るか取らないかしか方法がない . 動的計画法 を用いる
(2) 各アイテムは分割可能. 各アイテムの好きなだけとれる. *greedy* アルゴリズムを用いる.

動的計画法

- 解に対する再帰式(部分問題の解からもとの問題の解をつくる方法を再帰的に表す)
- 部分問題を非常に単純な場合から解いていき, それらの解を記憶していく.
- 上位の問題を解くとき, 必要に応じて, 既知の部分問題の解を用いる.
- 最後には全体の解が得られる.
- しばしば, 解を表現するある「量」について再帰式を作り, これを解き, 最後に解を構成することを行う.

0-1 ナップザック問題の例



0-1ナップザック問題

- 課題は、下記のものを見つけること

$$\max \sum_{i \in T} b_i \quad \text{ただし} \sum_{i \in T} w_i \le W$$

- この問題は *"0-1"* 問題と呼ばれる. というのも、どのアイテムも持っていくか持っていかないかの2値であるから.
- 実数ナップザック問題では、「実数個」持っていける、すなわち、任意の部分を持っていけるとする

0-1ナップザック問題: 力ずく

まずは、力ずくで

- n 個のアイテムがあるとすると, アイテムの組合せは 2^n 通りある.
- 全ての組合せをみて、重さの和が W より小さい組合せのなかで、価値の和が最大のものを見つける
- 実行時間は $O(2^n)$

0-1ナップザック問題: 力ずく

- もっと巧い手はないか?
- ある, 動的計画法を使えばよい
- そのためには、部分問題に分解する必要がある

試み:
各アイテムには 1..n のラベルが付いているとする
そうすると部分問題は
S_k = {アイテム 1, 2, .. k}
に対する最適解を見つけること

部分問題の定義

各アイテムには 1..n のラベルが付いているとすると
部分問題は S_k = {アイテム 1, 2, .. k} の最適解を見つけること

部分問題の定義はこれでよい.

- 課題は: 問題 (S_n) の解を部分問題 (S_k) の解で記述することができるか?
- 非常に残念なことに, これはできない.
  というのも….

部分問題の定義

最大積載量: W = 20

w_1=2	w_2=4	w_3=5	w_4=3
b_1=3	b_2=5	b_3=8	b_4=4

S_4に対して:
総重量: 14;
総価値: 20

w_1=2	w_2=4	w_3=5	w_5=9
b_1=3	b_2=5	b_3=8	b_5=10

S_5に対して:
総重量: 20
総価値: 26

荷物	重さ w_i	価値 b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4 S_5

S_4 に対する解は S_5 に対する解の部分になっていない!!

部分問題の定義

- 今の例では S_4 に対する解は S_5 に対する解の一部になっていない
- ということは部分問題の定義に欠陥があったことになる
- パラメータを増やそう: w, アイテム集合の正確な総重量
- 部分問題は $B[k,w]$ を計算すること

部分問題への再帰式

- 部分問題の再帰式は:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{otherwise} \end{cases}$$

- この意味: 総重量w である, S_k の最良部分集合は次の2つの場合の1つ:

1) 総重量wである, S_{k-1} の最良部分集合, または
2) 総重量$w-w_k$である, S_{k-1}の最良部分集合にアイテム k を付加えたもの

再帰式

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{otherwise} \end{cases}$$

- 総重量がw である, S_k の最良部分集合に, アイテム k が含まれるか否か.
- 場合1: w_k>w. アイテム k は解の一部にならない, もしそうであれば, 総重量> w となるがこれはありえないから.
- 場合2: w_k <=w. ことのきアイテム k は解に含まれうる, そこで, 価値の高い方を選ぶ

0-1 ナップザックアルゴリズム

```
for w = 0 to W       B[0,w] = 0
for i = 0 to n  {
   B[i,0] = 0
   for w = 0 to W
        if w_i <= w // アイテム i は解に含まれうる
              if b_i + B[i-1,w-w_i] > B[i-1,w]
                    B[i,w] = b_i + B[i-1,w- w_i]
              else
                    B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
}
```

実行時間

```
for w = 0 to W          O(W)
   B[0,w] = 0
for i = 0 to n          繰り返し n 回
   B[i,0] = 0
   for w = 0 to W          O(W)
        < 残り >
```

このアルゴリズムの実行時間は
O(n*W)

力ずくでは O(2^n) であった

例

次のデータで試してみよう:

n = 4 (要素数)
W = 5 (総重量)
要素 (重さ, 価値):
(2,3), (3,4), (4,5), (5,6)

例 (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				

```
for w = 0 to W
      B[0,w] = 0
```

例 (3)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

```
for i = 0 to n
      B[i,0] = 0
```

例 (4)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0 →	0			
2	0				
3	0				
4	0				
5	0				

i=1
b_i=3
w_i=2
w=1
w-w_i =-1

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

例 (5)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	**3**			
3	0				
4	0				
5	0				

i=1
b_i=3
w_i=2
w=2
w-w_i =0

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

例 (6)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	**3**			
4	0				
5	0				

i=1
b_i=3
w_i=2
w=3
w-w_i=1

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

例 (7)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	**3**			
5	0				

i=1
b_i=3
w_i=2
w=4
w-w_i=2

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

例 (8)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	**3**			

i=1
b_i=3
w_i=2
w=5
w-w_i=2

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

例 (9)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0 →	**0**		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

i=2
b_i=4
w_i=3
w=1
w-w_i=-2

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

例 (10)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	**3**		
3	0	3			
4	0	3			
5	0	3			

i=2
b_i=4
w_i=3
w=2
w-w_i=-1

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

例 (11)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	**4**		
4	0	3			
5	0	3			

i=2
b_i=4
w_i=3
w=3
w-w_i=0

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

例 (12)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	**4**		
5	0	3			

i=2
b_i=4
w_i=3
w=4
w-w_i=1

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

例 (13)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	**7**		

i=2
b_i=4
w_i=3
w=5
w-w_i=2

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

例 (14)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**	**0**	
2	0	3	3	**3**	
3	0	3	4	**4**	
4	0	3	4		
5	0	3	7		

i=3
b_i=5
w_i=4
w=1..3

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

例 (15)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	**5**	
5	0	3	7		

i=3
b_i=5
w_i=4
w=4
w- w_i=0

if w_i <= w // アイテム i は解に含まれうる
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

例 (15)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	→ **7**	

$i=3$
$b_i=5$
$w_i=4$
$w=5$
$w- w_i=1$

if w_i <= w // アイテム i は解に含まれうる
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w- w_i]$
  else
    B[i,w] = B[i-1,w]
else $B[i,w] = B[i-1,w]$ // $w_i > w$

例 (16)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**	0 →	**0**
2	0	3	3	3 →	**3**
3	0	3	4	4 →	**4**
4	0	3	4	5 →	**5**
5	0	3	7	7	

$i=3$
$b_i=5$
$w_i=4$
$w=1..4$

if w_i <= w // アイテム i は解に含まれうる
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w- w_i]$
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // $w_i > w$

例 (17)

アイテム:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7 →	**7**

$i=3$
$b_i=5$
$w_i=4$
$w=5$

if w_i <= w / // アイテム i は解に含まれうる
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w- w_i]$
  else
    B[i,w] = B[i-1,w]
else $B[i,w] = B[i-1,w]$ // $w_i > w$

0-1 ナップザック問題の例題

荷物	重さ w_i	価値 b_i
	2	80
	3	150
	4	240
	5	300

ナップザック
最大積載量: W = 7

W = 7

注意

- 今述べたアルゴリズムは, ナップザックに入れうる最大値を与えるのみである
- この最大値に達成するに必要なアイテムを知るには, このアルゴリズムに付加えるべきことがある
- それは LCS アルゴリズムとまったく同様にできるので, 各自試みられたい

動的計画法が使える条件

- 部分問題への分解
  - もとの問題が, 同じ構造をもったより小さな部分問題に分解できる
- 部分構造の最適性
  - 問題に対する解(最適解)は, 部分問題の解(最適解)の合成である
- 部分問題の重複性
  - 部分問題(大量にできる)にかなり重複がある(同じ部分問題が繰り返し表れる)

動的計画法パラダイム

- 動的計画法から、「多段階決定」という問題設定をとりさり、方法の特徴である
  - 部分構造の最適性
  - 部分問題の重複性

  のみを取り出して再帰的プログラムの実行速度を向上させる方法を、動的計画法パラダイム、または単に動的計画法という

例1: フィボナッチ数 $f(n) = f(n–1) + f(n–2)$

単に再帰呼び出しを行うと $O(2^n)$. しかし、配列を利用したり、一度求めた値を記憶するようにすれば $O(n)$

例2: 組合せの数 $C(n, k) = C(n–1, k–1) + C(n–1, k)$

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. $\mathbf{O(2^n)}$