動的計画法

Dynamical Programming

計算機アルゴリズム特論：2017年度

只木進一

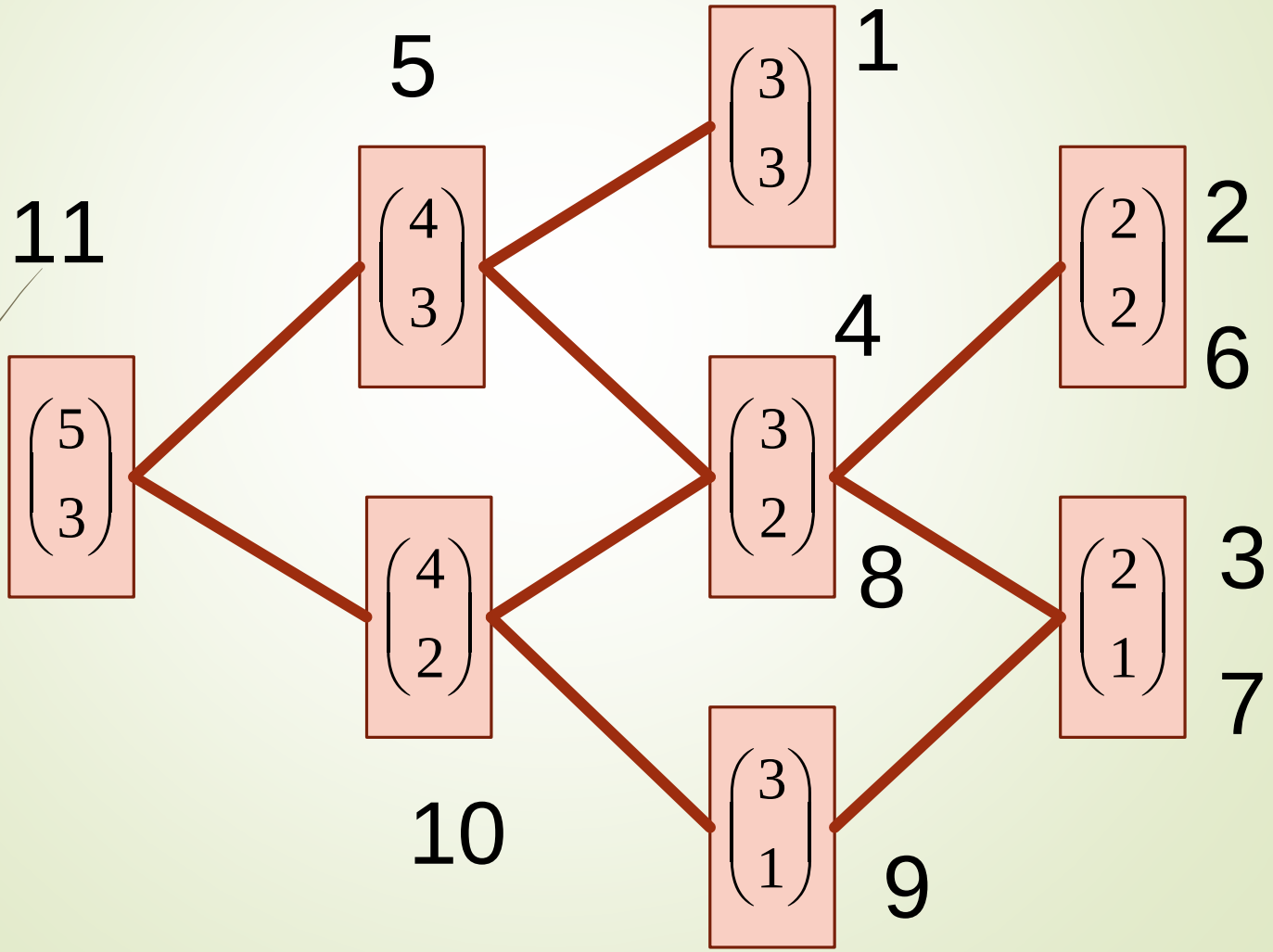
二項係数 (binomial coefficients)

$$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$$

$$\binom{n}{1} = 1$$

$$\binom{n}{n} = 1$$

再帰における値の確定する順序



- 再帰では、同じ二項係数を複数回計算することになる
- 一旦計算した係数を保存することで、複数回計算を回避

```
int binomial(int n, int r){
    if(b_nr > 0)return b_nr;
    int c = 1;
    if(r == 1){c = n;}
    else if(n > r){
        c = binomial(n − 1, r)+binomial(n − r);}
    b_nr = c;
    return c;
}
```

配列b_{ij}の初期値はゼロ

動的計画法
Dynamical Programming

- 問題を小さなサイズに分けて解く
  - 再帰や分割統治法と同じ考え
  - 必ずしも、大きいサイズから小さいサイズへとは限らない
- 同じ問題を二度解かないように、結果を記憶しておく
- アルゴリズムを設計すること自体を「動的計画法」と呼ぶ

ナップザック問題 Knapsack Problems

- 品物の集合$G = \{g_i | 0 \leq i < n\}$
  - g_iの重量w_iと価値v_i
- 部分集合$S \subseteq G$に対して、制限$\sum_{i \in S} w_i \leq W$の下で、$V_S = \sum_{i \in S} v_i$を最大化する$S$を求める
- 重量w_iは自然数とする

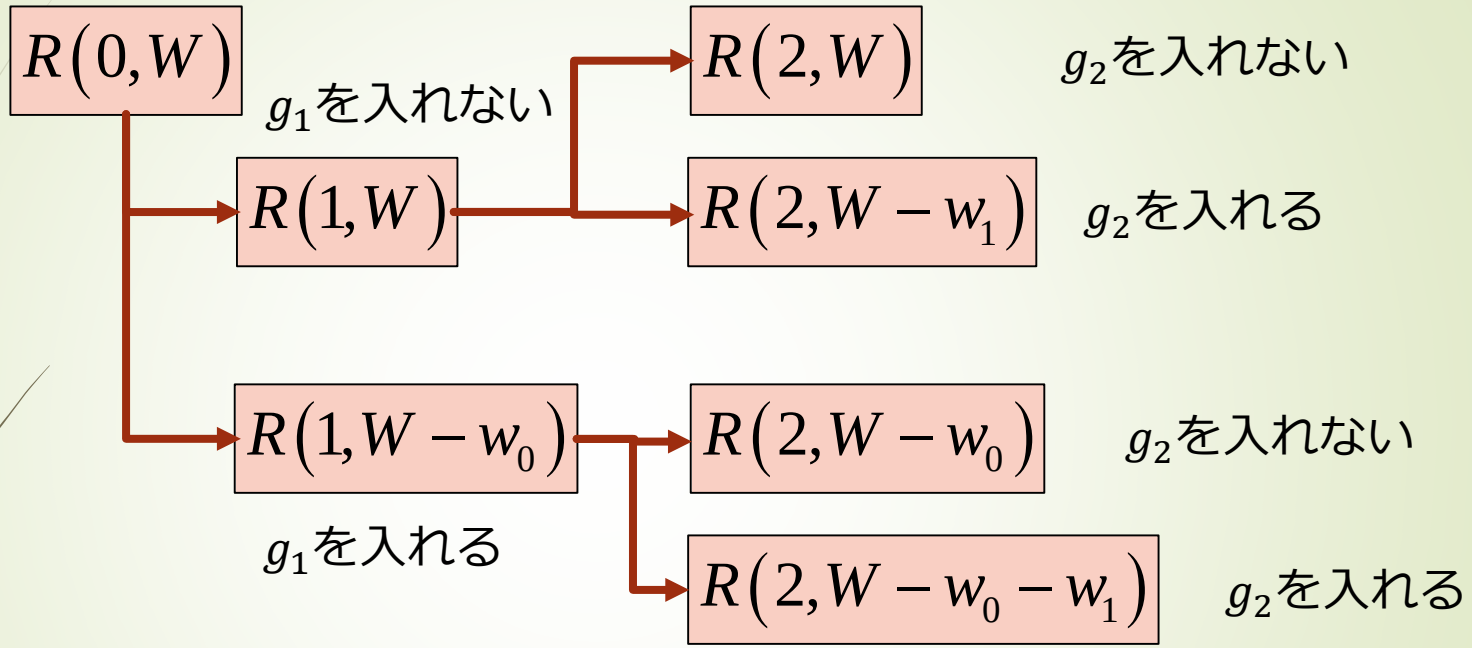
ナップザック問題：再帰的解法の概要

- 許される残りの重量w
- i番目の荷物(w_i, v_i)を入れるべきか
  - 入りきらない場合$w_i > w$、i番目の荷物は入れない
  - それ以外の場合、i番目の荷物を採用した場合と、採用しない場合の価値が大きな方を選ぶ
- 停止条件
  - $i = n$の場合、追加の価値はゼロ

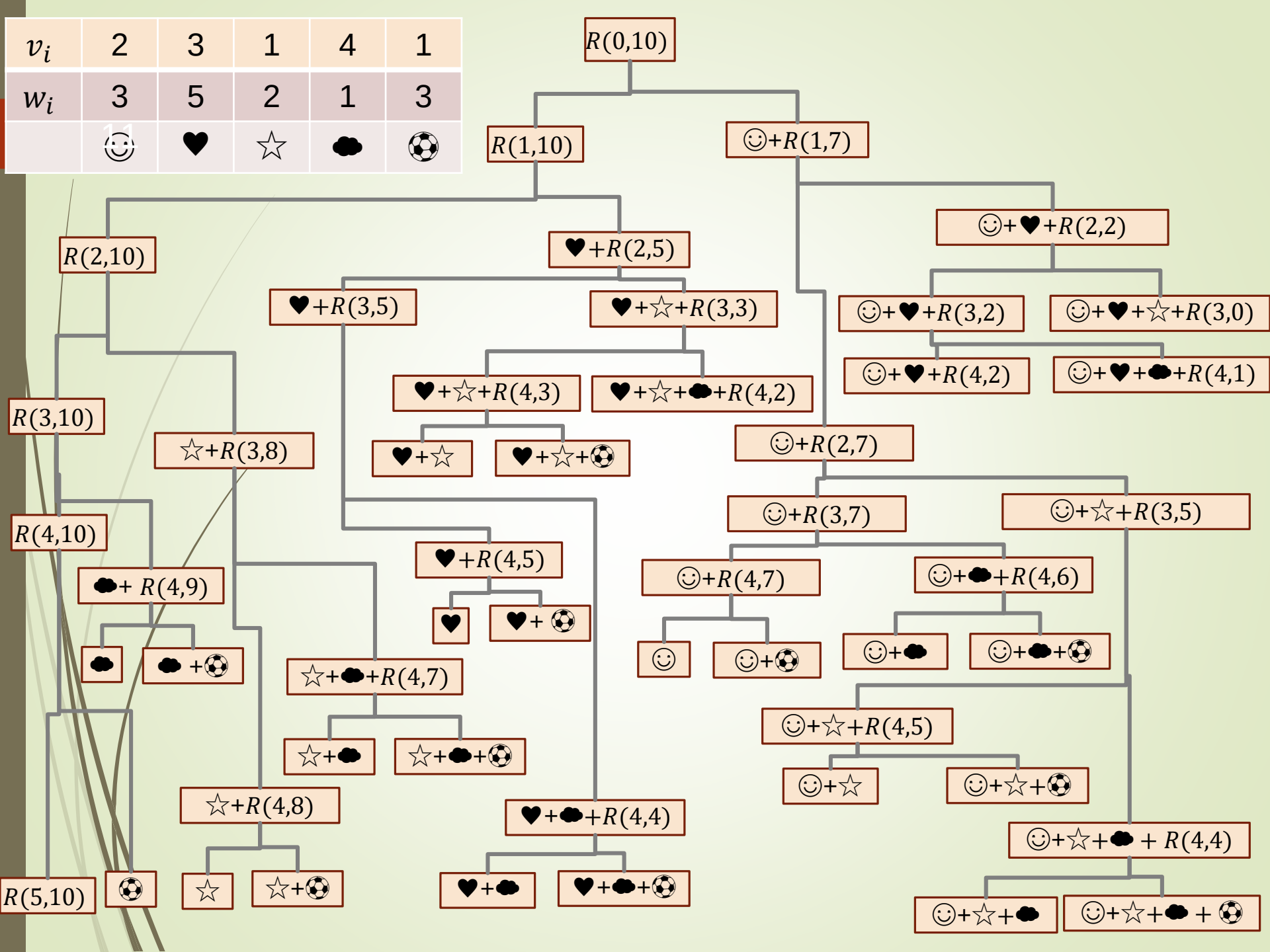
ナップザック問題：再帰的解法

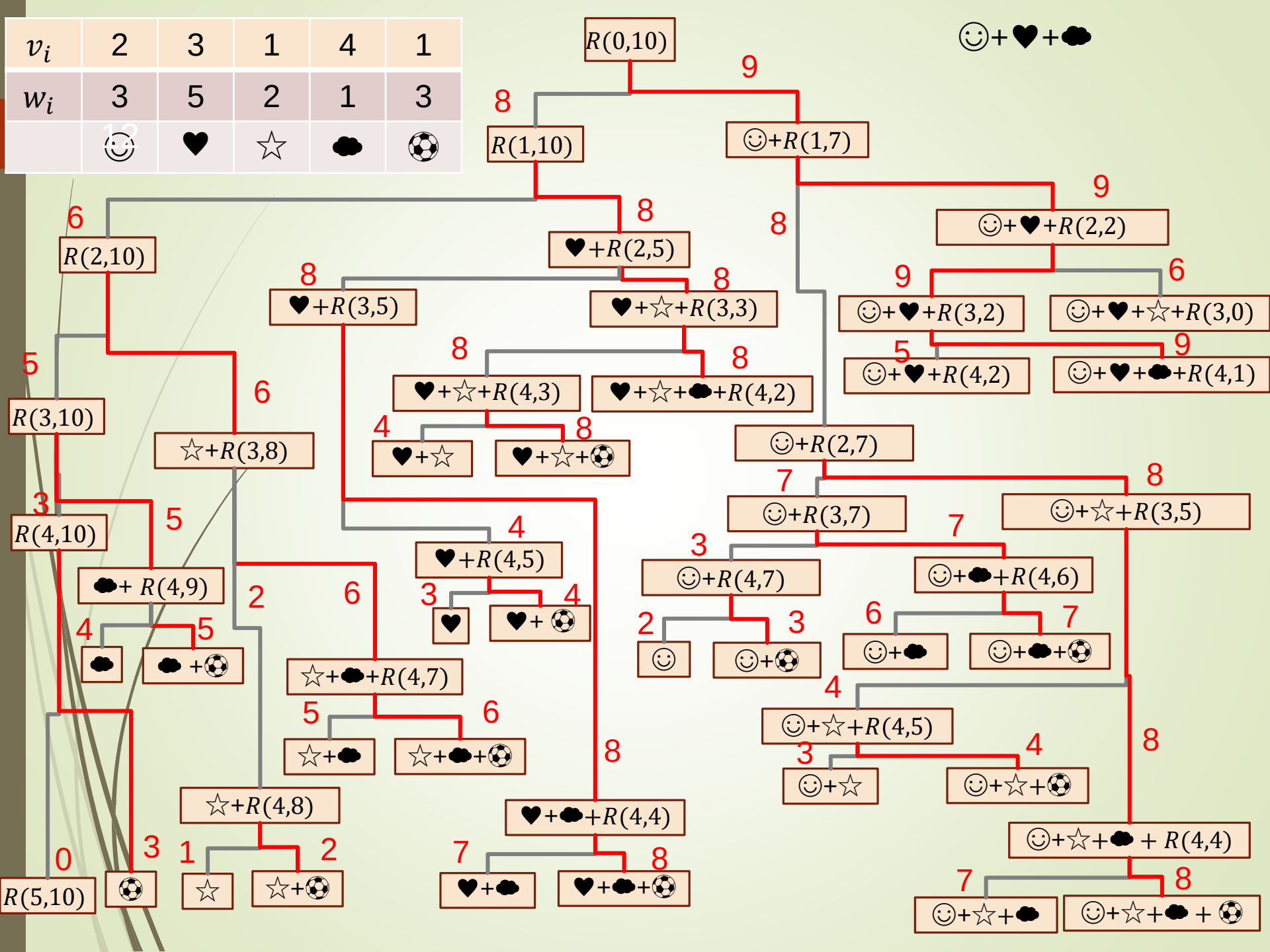
- $\mathrm{R}(i, w)$をi番以上の品物で重量がw以下で、価値の最大値
- 最大で2^n回の操作が必要

```
int R(int i, int w){
  if(i == n) return 0;//残りの荷物なし
  if(w < w_i) return R(i + 1, w);//g_iは使わない
  return max(R(i + 1, w),//g_iを使わない
                  R(i + 1, w − w_i)+v_i);//g_iを使う
}
```


$R(0,W)$
g_1を入れない
g_1を入れる
$R(1,W)$
$R(1,W-w_0)$
$R(2,W)$
g_2を入れない
$R(2,W-w_1)$
g_2を入れる
$R(2,W-w_0)$
g_2を入れない
$R(2,W-w_0-w_1)$
g_2を入れる

v_i	2	3	1	4	1
w_i	3	5	2	1	3
	☺	♥	☆	●	⚽




v_i 2 3 1 4 1
w_i 3 5 2 1 3
12
R(0,10)
R(1,10)
☺+R(1,7)
R(2,10)
♥+R(2,5)
♥+R(3,5)
♥+☆+R(3,3)
♥+☆+R(4,3)
♥+☆+●+R(4,2)
♥+☆
♥+☆+⚽
R(3,10)
☆+R(3,8)
R(4,10)
●+ R(4,9)
●
● +⚽
♥+R(4,5)
♥
♥+ ⚽
☆+●+R(4,7)
☆+●
☆+●+⚽
☆+R(4,8)
R(5,10)
⚽
☆
☆+⚽
♥+●+R(4,4)
♥+●
♥+●+⚽
☺+♥+R(2,2)
☺+♥+R(3,2)
☺+♥+☆+R(3,0)
☺+♥+R(4,2)
☺+♥+●+R(4,1)
☺+R(2,7)
☺+R(3,7)
☺+☆+R(3,5)
☺+R(4,7)
☺+●+R(4,6)
☺
☺+⚽
☺+●
☺+●+⚽
☺+☆+R(4,5)
☺+☆
☺+☆+⚽
☺+☆+●+R(4,4)
☺+☆+●
☺+☆+●+⚽

同じ値の評価が複数回

- 例えば
  - $R(4,4), R(4,5), R(4,7)$
  - $R(3,5),$

同じ値を複数回計算しないための工夫

- 同じ値の組(i, w)に対して、R(i, w)を複数回計算する可能性→配列に記憶
- 配列$q[n+1][W+1]$
  - 初期値を-1に設定

ナップザック問題：再帰的解法の改良

```
int R(int i, int w){
  if (q_iw ≠ −1)return q_iw;
  int r;
  if(i == n) {r = 0;}
  else {
    if(w < w_i) r = R(i + 1, w);//g_iは使わない
    else r = max(R(i + 1, w),
                 R(i + 1, w − w_i) + v_i);
  }
  q_iw = r;
  return r;
}
```

最大nW回の操作

q_{ij}を端から計算していく工夫

- 配列q_{ij}では、iが大きく、jが小さいところから、値が定まっている。
- 停止条件$q_{nj} = 0$

- 漸化式として再整理

$$q_{nj} = 0, \forall j$$

$$q_{ij} = \begin{cases} q_{(i+1)\,j} & \text{if } \; j < w_j \\ \max\left(q_{(i+1)\,j}, q_{(i+1)(\,j-w_i\,)} + v_i\right) & \text{otherwise} \end{cases}$$

ナップザック問題：再帰的解法の改良２

```
for(int j = 0;j < w + 1;j + +)q_nj = 0;
for(int i = n − 1;i >= 0;i − −){
  for(int j = 0;j <= w;j + +){
    if(j < w_i)q_ij = q_(i+1)j;
    else
      q_ij = max( q_(i+1)j, q_(i+1)(j−w_j) + v_i);
  }
}
```

最大nW回の操作

Javaでの実装
クラス設計

- 品物のクラスGood
  - 重量と価値を保持
- ナップザックのクラス
  - 入っている品物
  - 総重量と総価値を保持
  - コピーできるように
    - Clonableインターフェイス

再帰的手法と動的計画法での注意

- メソッドの戻り値をナップザッククラスのインスタンスとする
- トラックバック時にインスタンスの上書きを防ぐ
  - 戻り値はインスタンスのコピー

クラス構成

- AbstractKnapsack
  - Knapsack問題解法の抽象クラス
- Recursive
  - 単純な再帰手法
- DynamicalProgramming
  - 動的計画法による
- Sequential
  - 配列を下から埋めていく