



# 動的計画法

Dynamical Programming

計算機アルゴリズム特論：2015年度

只木進一

# Fibonacci数列

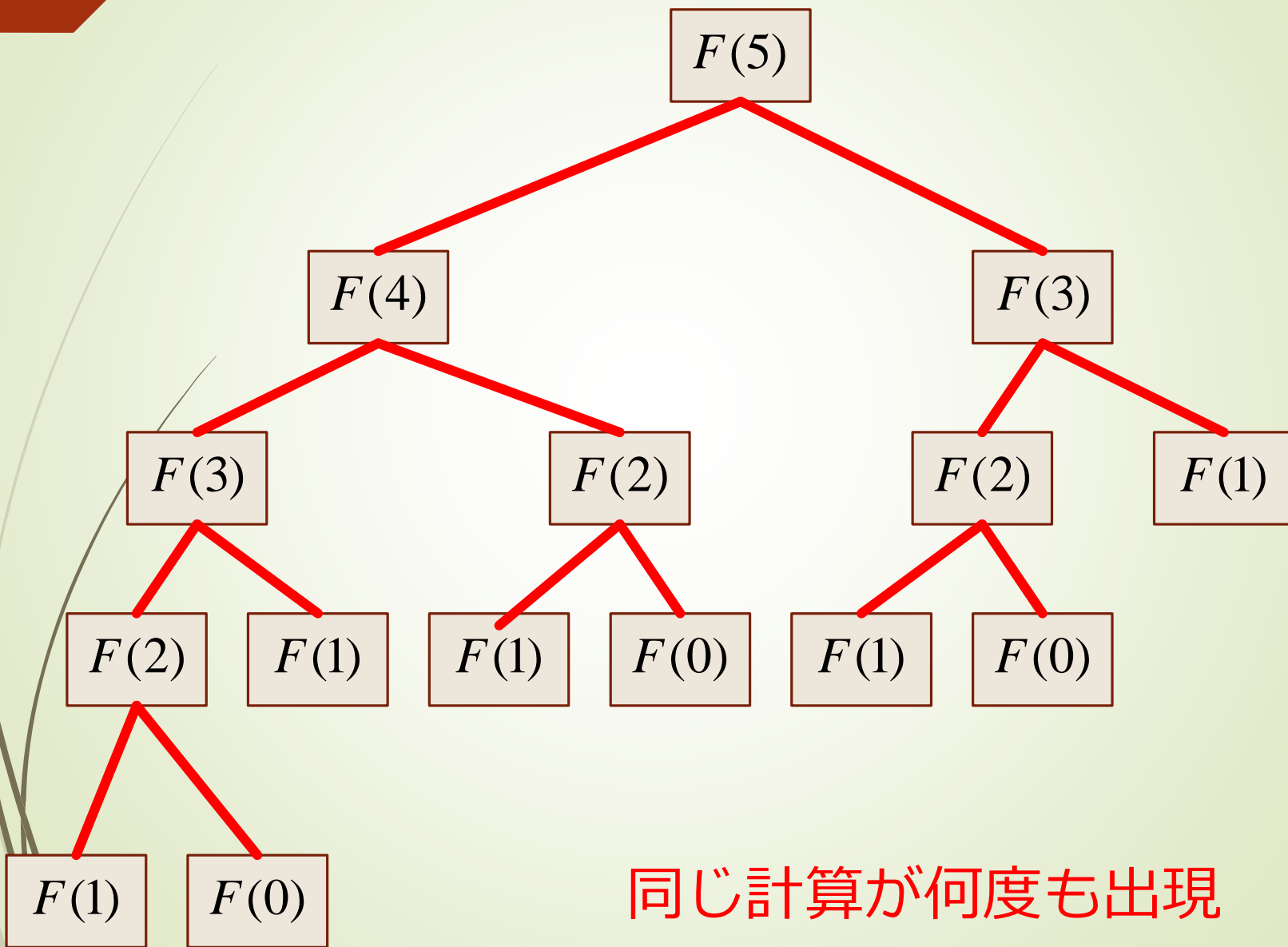
## ➡ 定義

$$f_0 = f_1 = 1$$

$$f_k = f_{k-1} + f_{k-2}$$

## ➡ 再帰アルゴリズム

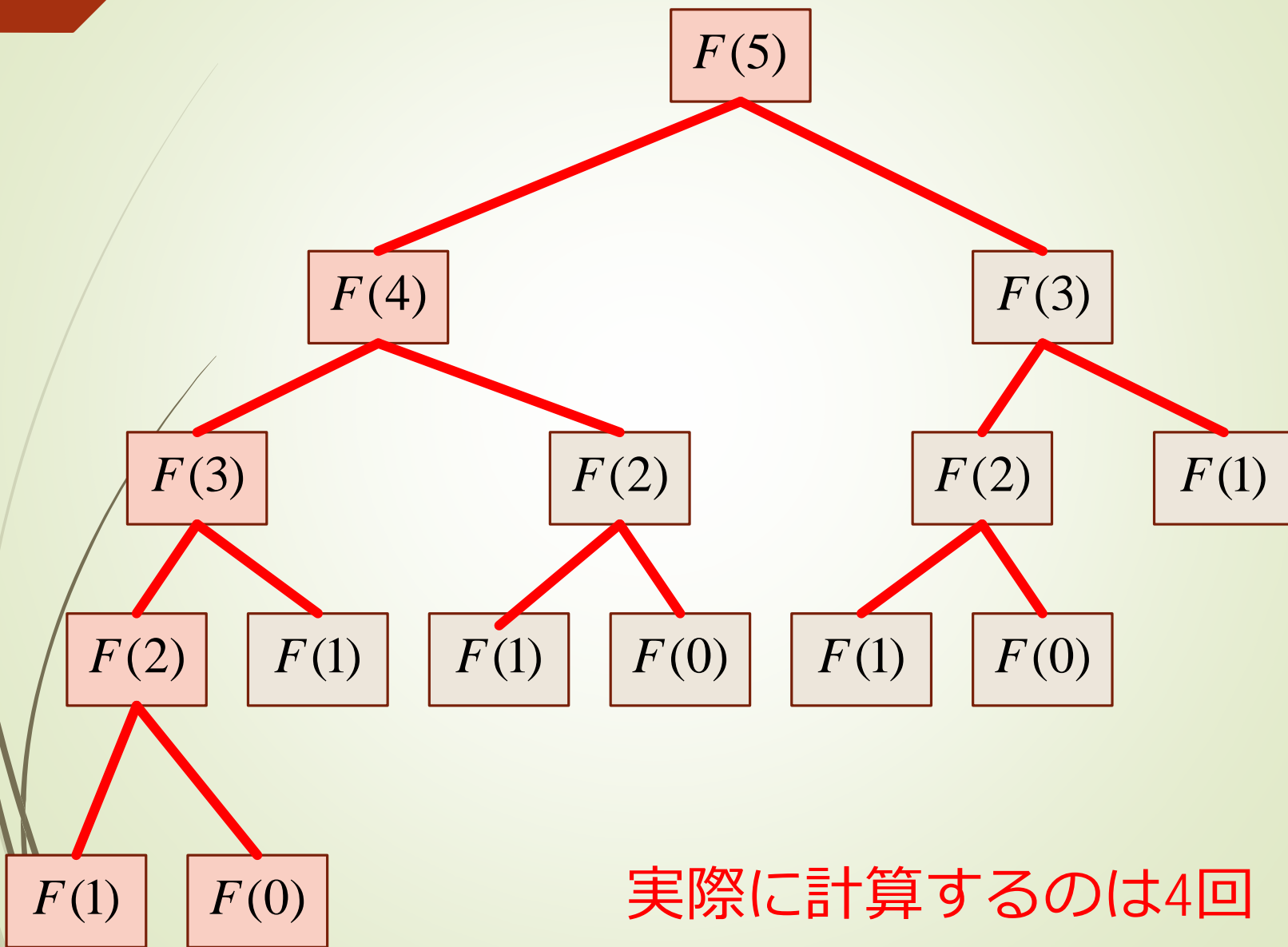
```
int Fibonacci (int n){  
    if(n == 0 || n == 1) return 1;  
    return Fibonacci(n-1)+Fibonacci(n-2);  
}
```



同じ計算が何度も出現

## 必要最小限の値を記憶しておく

```
int Fibonacci (int n){  
    if (n ==0 || n ==1) return 1;  
    int f0=1;    int f1=1;  
    int f2;  
    for(int i=2;i<=n;i++){  
        f2 = f1 + f0;  
        f1 = f2;    f0 = f1;  
    }  
    return f2;  
}
```



実際に計算するのは4回

## 必要な値を記憶しておく

```
int Fibonacci (int n){  
    if (n ==0 || n ==1) return 1;  
    int f[] = new int[n];  
    f[0]=1; f[1]=1;  
    return fSub(n,f);  
}  
  
int fSub(int n,int f[]){  
    if(f[n]>0) return f[n];  
    return fSub(n-1,f)+fSub(n-2,f);  
}
```

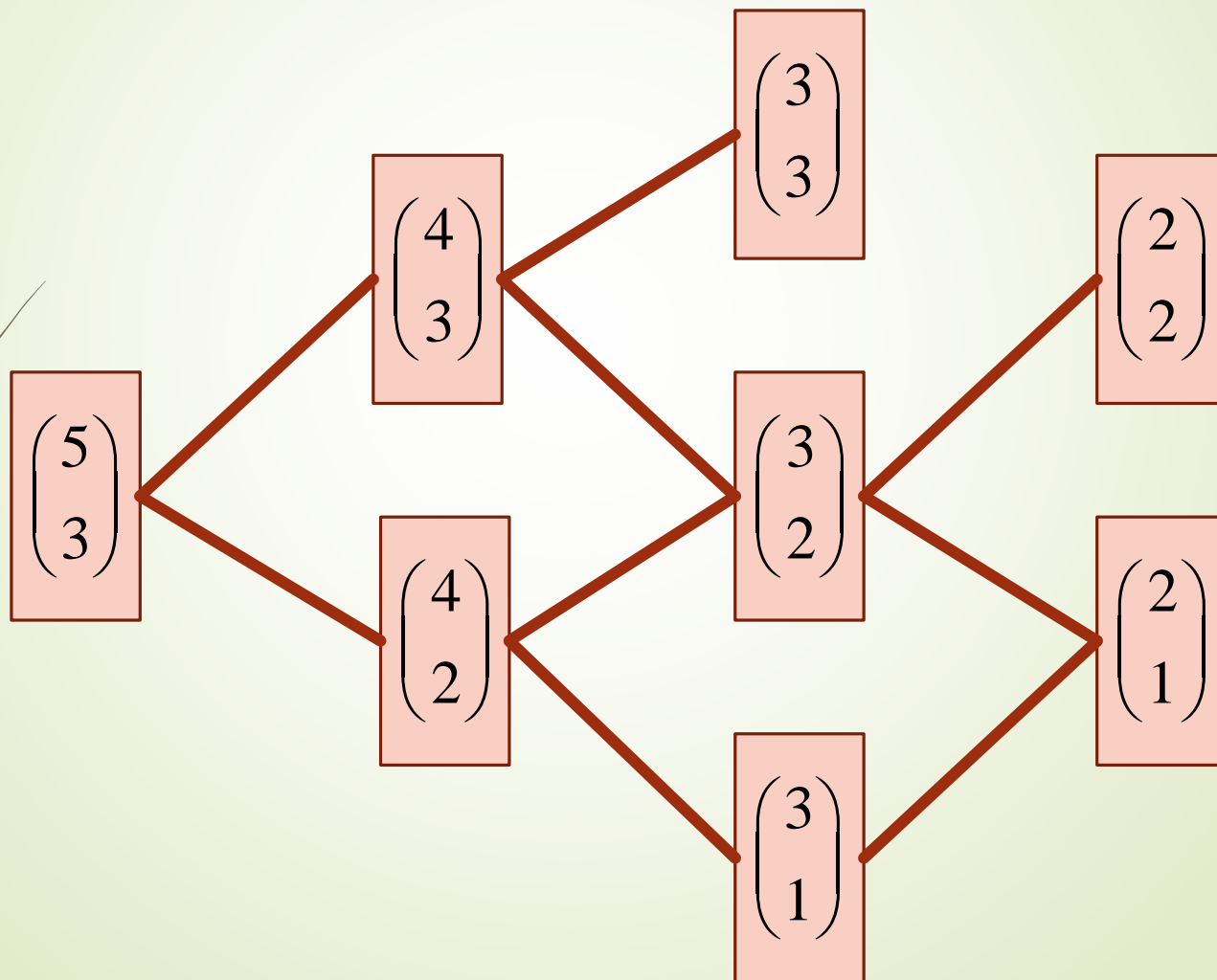


## 二項係数 (binomial coefficients)


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

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

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







```
int binomial(int n, int r){  
    if(b[n][r]>0)return b[n][r];  
    int bb=1;  
    if(r==1){bb=n;}  
    else if(n>r){  
        bb = binomial(n-1,r)+binomial(n-r);}  
    bb[n][r]=bb;  
    return bb;  
}
```



# 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、 $\sum_{i \in S} v_i$  を最大化する  $S$  を求める
- 重量  $w_i$  は自然数とする

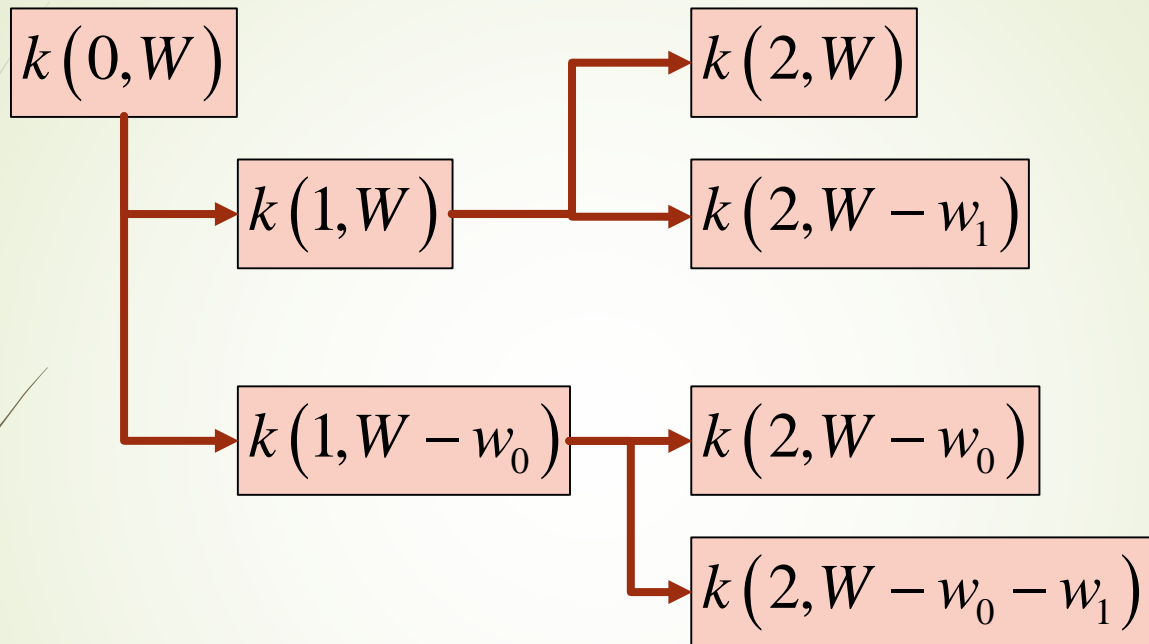
# ナップザック問題： 再帰的解法の概要


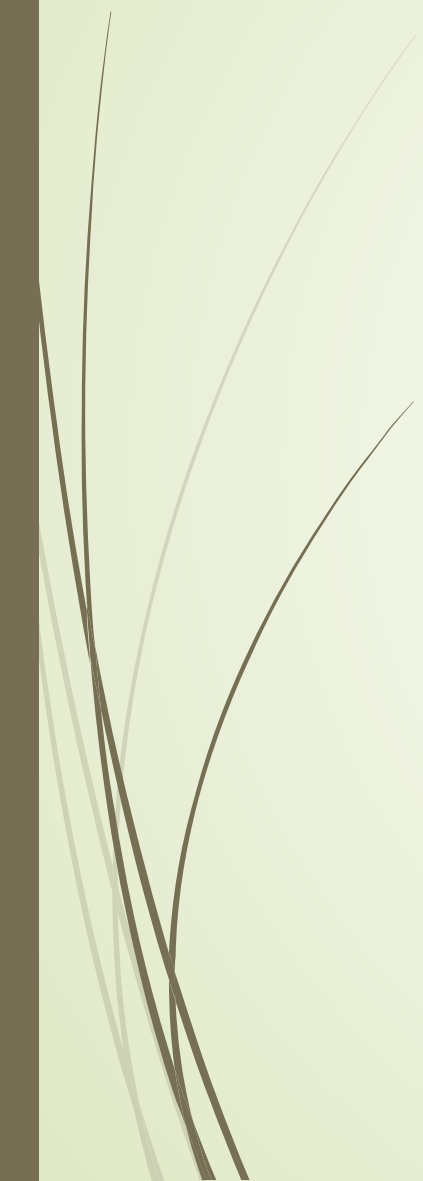
- 許される残りの重量  $w$
- $i$  番目の荷物  $(w_i, v_i)$  を入れるべきか
  - 入りきらない場合  $w_i > w$ 、 $i$  番目の荷物は入れない
  - それ以外の場合、 $i$  番目の荷物を採用した場合と、採用しない場合の大きな方を選ぶ
- 停止条件
  - $i = n$  の場合、追加の価値はゼロ

# ナップザック問題： 再帰的解法

- $kRec(i, w)$  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 kRec(int i, int w){  
    if(i==n) return 0;  
    if(w<w_i) return kRec(i+1,w); //g_iは使わない  
    return max(kRec(i+1,w),  
               kRec(i+1,w-w_i)+v_i);  
}
```




- 
- 
- 同じ値の組 $(i, w)$ に対して、 $k\text{Rec}(i, w)$ 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 kRec(int l, int w){
    if (q[l][w]!=-1)return q[l][w];
    int res;
    if(i==n) {res=0;}
    else {
        if(w<w_i) res=kRec(i+1,w);//g_iは使わない
        else res=max(kRec(i+1,w),
                     kRec(i+1,w-w_i)+v_i);
    }
    q[l][w]=res;
    return res;
}
```

最大 $nW$ 回の操作



- 
- 配列 $q[i][j]$ 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 $q[n][j] = 0$

## ➡ 漸化式として再整理


$$q[n][j] = 0, \forall j$$

$$q[i][j] = \begin{cases} q[i+1][j] & \text{if } j < w_j \\ \max(q[i+1][j], q[i+1][j - w_i] + v_i) & \text{otherwise} \end{cases}$$

## ナップザック問題： 再帰的解法の改良 2


```
for(int j=0;w+1;j++)q[n][j]=0;
for(i=n-1;i>=0;i--){
  for(j=0;j<=w;j++){
    if(j < w_i)q[i][j] = q[i+1][j];
    else
      q[i][j] = max( q[i+1][j],
                    q[i+1][j-w_i] + 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
  - 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



# 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