2010/03/19 代々木オリンピックセンター （情報オリンピック春合宿）

プログラミングコンテストでの動的計画法

秋葉 拓哉 / (iwi)

はじめに

- 「動的計画法」は英語で
  「Dynamic Programming」
  と言います.

- 略して「DP」とよく呼ばれます.

- スライドでも以後使います.

全探索と DP の関係

動的計画法のアイディア

ナップサック問題とは？

- n 個の品物がある
- 品物 i は重さ w_i, 価値 v_i
- 重さの合計が U を超えないように選ぶ
  - 1 つの品物は 1 つまで
- この時の価値の合計の最大値は？

ナップサック問題の例

U = 5

品物 1	品物 2	品物 3	品物 4
$w_1 = 2$	$w_2 = 1$	$w_3 = 3$	$w_4 = 2$
$v_1 = 3$	$v_2 = 2$	$v_3 = 4$	$v_4 = 2$

答え 7

品物 1	品物 2	品物 3	品物 4
$w_1 = 2$	$w_2 = 1$	$w_3 = 3$	$w_4 = 2$
$v_1 = 3$	$v_2 = 2$	$v_3 = 4$	$v_4 = 2$

全探索のアルゴリズム (1/3)

品物 1 から順に，使う場合・使わない場合の両方を試す

全探索のアルゴリズム (2/3)

```
// i 番目以降の品物で，重さの総和が u 以下となるように選ぶ
int search(int i, int u) {
        if (i == n) {           // もう品物は残っていない
                return 0;
        } else if (u < w[i]) {  // この品物は入らない
                return search(i + 1, u);
        } else {                // 入れない場合と入れる場合を両方試す
                int res1 = search(i + 1, u);
                int res2 = search(i + 1, u - w[i]) + v[i];
                return max(res1, res2);
        }
}
```

外からは search(0, U) を呼ぶ

全探索のアルゴリズム (3/3)

```
// i 番目以降の品物で，重さの総和が u 以下となるように選ぶ
int search(int i, int u) {
        …
}
```

このままでは，指数時間かかる

（U が大きければ，2^n 通り試してしまう）

少しでも n が大きくなると
まともに計算できない

改善のアイディア

```
// i 番目以降の品物で，重さの総和が u 以下となるように選ぶ
int search(int i, int u) {
        …
}
```

大事な考察 : search(i, u) は

i, u が同じなら常に同じ結果

同じ結果になる例

品物 3 以降の最適な選び方は同じ

- 2 + search(3, 10 – 2)
- 3 + search(3, 10 – 2)

} 2 度全く同じ探索をしてしまう

動的計画法のアイディア

- 同じ探索を 2 度しないようにする

- search(i, u) を
  - 各 (i, u) について一度だけ計算
  - その値を何度も使いまわす

- 名前はいかついが，非常に簡単な話
  - 簡単な話だが，効果は高い（次）

動的計画法の計算量

- 全探索では指数時間だった
  - $O(2^n)$，N が少し大きくなるだけで計算できない
- DP では，各 (i, u) について 1 度計算する
  - なので，約 N × U 回の計算で終了
  - O(NU) （擬多項式時間，厳密にはこれでも指数時間）
- N が大きくても大丈夫になった

メモ探索，漸化式

動的計画法の実装

2 つの方法

- ナップサック問題の続き
  - DP のプログラムを完成させましょう

- 以下の 2 つの方法について述べます
  1. 再帰関数のメモ化
  2. 漸化式＋ループ

方法 1 : メモ化 (1/2)

```
// i 番目以降の品物で，重さの総和が u 以下となるように選ぶ
int search(int i, int u) {
        …
}
```

- 全探索の関数 search を改造する
  - はじめての (i, u) なら
    - 探索をして，その値を配列に記録
  - はじめてでない (i, u) なら
    - 記録しておいた値を返す

方法 1 : メモ化 (2/2)

```
bool done[MAX_N][MAX_U + 1];        // すでに計算したかどうかの記録
int memo[MAX_N][MAX_U + 1];         // 計算した値の記録

// i 番目以降の品物で，重さの総和が u 以下となるように選ぶ
int search(int i, int u) {
        if (done[i][u]) return memo[i][u];        // もう計算してた？
        int res;

        ...                                       // 値を res に計算

        done[i][u] = true;                        // 計算したことを記憶
        memo[i][u] = res;                         // 値を記憶
        return res;
}
```

done を全て false に初期化し，search(0, U) を呼ぶ

方法 2 : 漸化式 (1/3)

- 全探索のプログラムを式に

$$\mathrm{search}(i, u) = \max \begin{cases} \mathrm{search}(i+1, u) \\ \mathrm{search}(i+1, u - w_i) + v_i \end{cases}$$

（場合分けは省略）

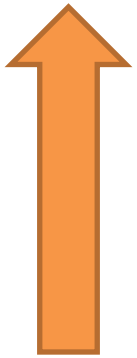
- このような式を，漸化式と呼ぶ
  - パラメータが異なる自分自身との間の式

方法 2 : 漸化式 (2/3)

$$\mathrm{search}(i, u) = \max \begin{cases} \mathrm{search}(i + 1, u) \\ \mathrm{search}(i + 1, u - w_i) + v_i \end{cases}$$

- i が大きい方から計算してゆけばよい
  – 下のような表を埋めてゆくイメージ

i ＼ u	0	1	2	3	…
1					
2					
3					
…					



i の大きい方から埋めてゆく

埋める際に，既に埋めた部分の値を利用する

方法 2 : 漸化式 (3/3)

```
int dp[MAX_N + 1][MAX_U + 1];          // 埋めてゆく「表」

int do_dp() {
    for (int u = 0; u <= U; u++) dp[N][u] = 0;       // 境界条件

    for (int i = N – 1; i >= 0; i--) {
        for (int u = 0; u <= U; u++) {
            if (u < w[i])                    // u が小さく，商品 i が使えない
                dp[i][u] = dp[i + 1][u];
            else                             // 商品 i が使える
                dp[i][u] = max( dp[i + 1][u] ,  dp[i + 1][u – w[i]] + v[i] );
        }
    }
    return dp[0][U];
}
```

どちらの方法を使うべきか

- 好きな方を使いましょう
  - どちらでも，多くの場合大丈夫です
- 後で，比較をします
  - どちらも使いこなせるようになるのがベスト

呼び方

- 方法 1 （再帰関数のメモ化）をメモ探索
- 方法 2 （漸化式＋ループ）を動的計画法 (DP)

と呼ぶことがあります.

– 「動的計画法」と言ったとき
  - メモ探索を含む場合と含まない場合がある
  - 2 つにあまり差がない状況では区別されない
    – アルゴリズムの話としては（ほぼ）差がない
    – 書き方としての違い

Tips

- 大きすぎる配列をローカル変数として確保しないこと
  - 大きな配列はグローバルに取る
    - スタック領域とヒープ領域
    - スタック領域の制限
  - 大丈夫な場合もあるが，多くはない（IOI 等）

知らない問題を動的計画法で解く

動的計画法の作り方

動的計画法の問題を解く

- おすすめの流れ（初心者）
  1. 全探索によるアルゴリズムを考える
  2. 動的計画法のアルゴリズムにする

- ここまでのナップサック問題の説明と同様の流れで解けばよい
  - パターンにしてしまおう

- 実際には，全探索を考えるのと，漸化式を考えるのは，ほぼ同じ行為

簡単な例で復習

- フィボナッチ数列

```
int fib(int n) {
        if (n == 0) return 0;
        if (n == 1) return 1;
        return fib(n – 1) + fib(n – 2);
}
```

- これを，先ほどと同様の流れで
  - メモ探索
  - 動的計画法（ループ）

に書き直してみてください.

非常に簡単ですが，やり方の確認なので，形式的に行うようにしましょう

メモ探索

```
bool done[MAX_N + 1];
int memo[MAX_N + 1];

int fib(int n) {
        if (n == 0) return 0;
        if (n == 1) return 1;

        if (done[n]) return memo[n];

        done[n] = true;
        return memo[n] = fib(n – 1) + fib(n- 2);
}
```

ループ

```
int dp[MAX_N + 1];

int fib(int n) {
	dp[0] = 0;
	dp[1] = 1;

	for (int i = 2; i <= n; i++) dp[i] = dp[i – 1] + dp[i – 2];

	return dp[n];
}
```

どんな全探索を考えれば良いのか

動的計画法にできる全探索

ナップサック問題

- ナップサック問題の際に使った全探索

```
// i 番目以降の品物で，重さの総和が u 以下となるように選ぶ
int search(int i, int u) {
        …
}
```

- これ以外にも全探索は考えられる

良くない全探索 (1/3)

```
// i 番目以降の品物で，重さの総和が u 以下となるように選ぶ
// そこまでに選んだ品物の価値の和が vsum
int search(int i, int u, int vsum) {
        if (i == n) {           // もう品物は残っていない
                return vsum;
        } else if (u < w[i]) {  // この品物は入らない
                return search(i + 1, u, vsum);
        } else {                // 入れない場合と入れる場合を両方試す
                int res1 = search(i + 1, u, vsum);
                int res2 = search(i + 1, u - w[i], vsum + v[i]);
                return max(res1, res2);
        }
}
```

外からは search(0, U, 0) を呼ぶ

良くない全探索 (2/3)

```
// i 番目以降の品物で，重さの総和が u 以下となるように選ぶ
// そこまでに選んだ品物の価値の和が vsum
int search(int i, int u, int vsum) {
        ...
}
```

- 引数が増え，動的計画法に出来ない
  - 各 (i, u, vsum) で一度ずつ計算すれば出来るが，効率が悪い
- このような方針で書くと枝刈りができたりするので，不自然な書き方ではないが…

同じ結果になる例 (再)

品物 3 以降の最適な選び方は同じ

- 2 + search(3, 10 – 2)
- 3 + search(3, 10 – 2)

} 2 度全く同じ探索をしてしまう

良くない全探索 (3/3)

- 引数が増え，動的計画法に出来ない
  - 各 (i, u, vsum) で一度ずつ計算すれば出来るが，効率が悪い
- このようなことを防ぐため，

全探索の関数の引数は，
その後の探索に影響のある
最低限のものにする

- 選び方は，残る容量にはよるが，そこまでの価値によらない．なので vsum は引数にしない

その他の注意

- グローバル変数を変化させるようなことはしてはいけない

- 状態をできるだけシンプルにする
  - × 「どの商品を使ったか」・・・ 2^n
  - ○ 「今までに使った商品の重さの和」 ・・・ U

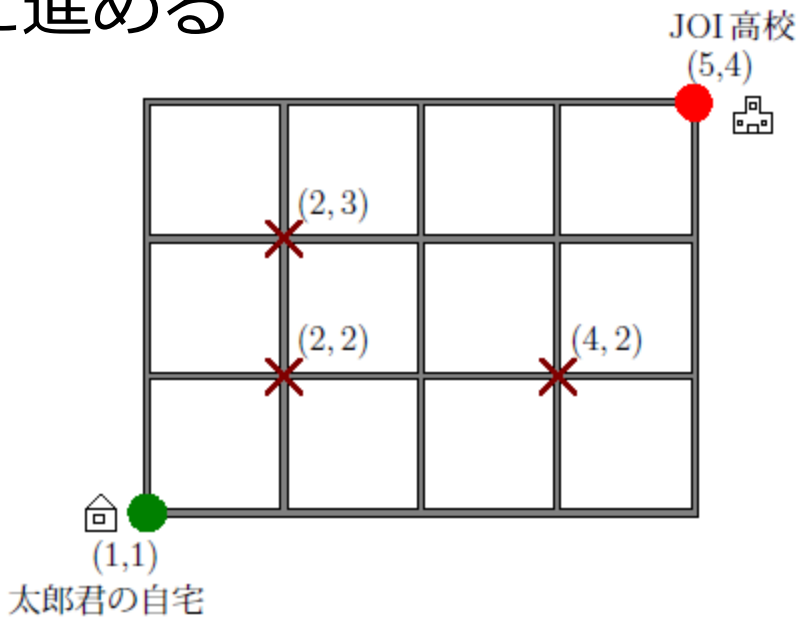
- ここで扱った事項は，アルゴリズムを考える際だけでなく，動的計画法のアルゴリズムをより効率的にしようとする際にも重要です．

DP の問題をいくつか簡単に

典型的問題

経路の数 (1/2)

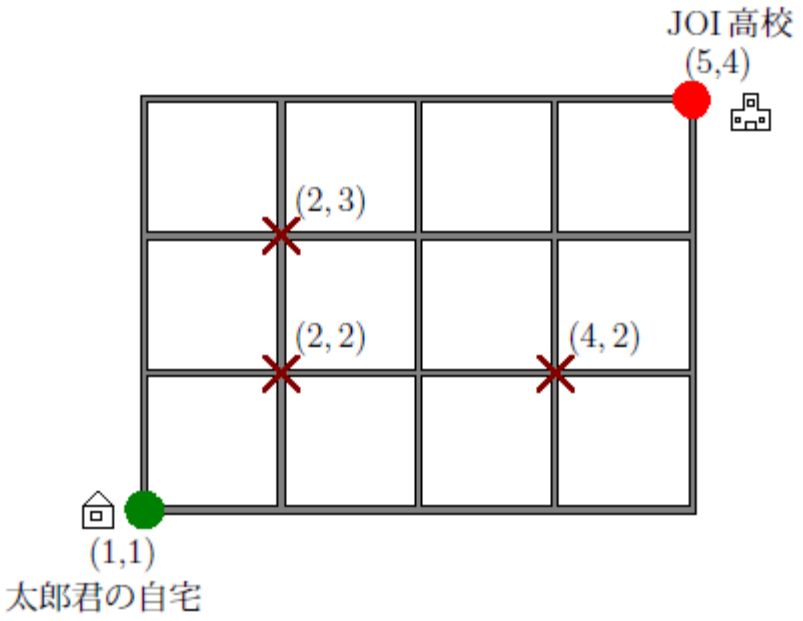
- ● から ● へ移動する経路の数
  - 右か上に進める



- ある地点からの経路の数は，そこまでの経路によらない

経路の数 (2/2)

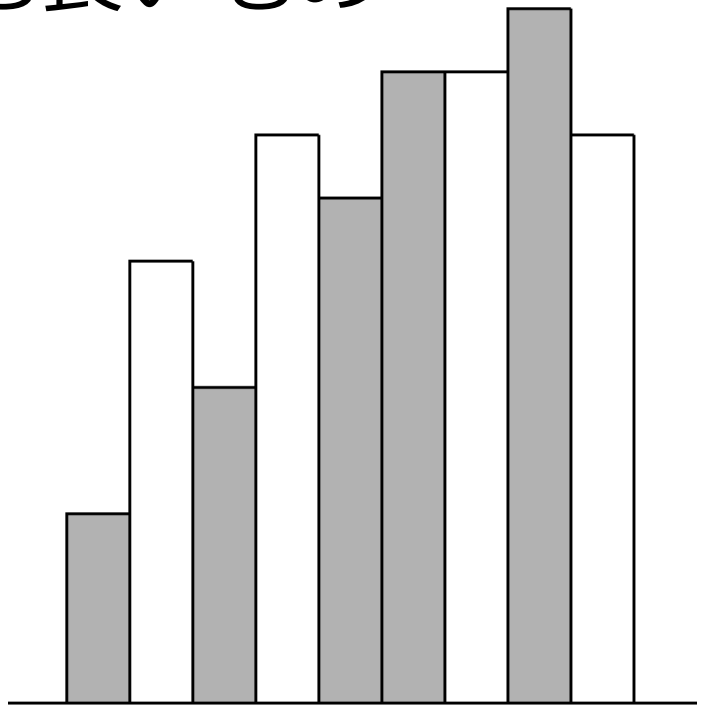
- 通れるなら
  - route(x, y) = route(x-1, y) + route(x, y-1)
- ×なら
  - route(x, y) = 0



最長増加部分列(LIS) (1/2)

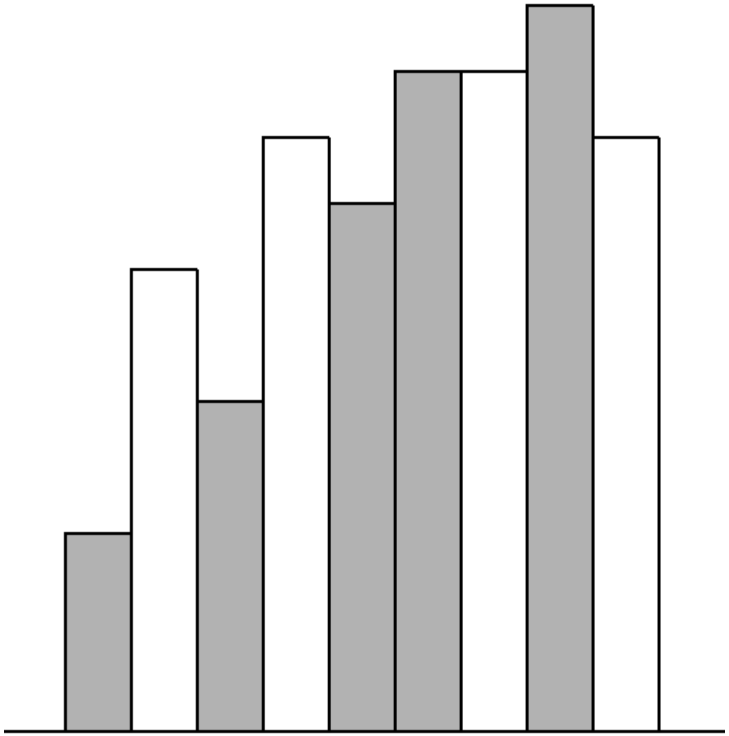
- 数字の列が与えられる
- 増加する部分列で，最も長いもの

- 後ろの増加部分列は，
  一番最後に使った数
  のみよる



最長増加部分列(LIS) (2/2)

- lis(i) := i を最後に使った LIS の長さ
- $lis(i) = \max_j \{ lis(j) + 1 \}$
  - ただし，$j < i$ かつ $a_j < a_i$



i	1	2	3	4	5	6	7	8	9
lis(i)	1	2	2	3	3	4	4	5	4

最長共通部分列 (LCS)

- 2 つの文字列が与えられ，その両方に含まれる最も長い文字列
  - X = “ABCBDAB”，Y = “BDCABA” ⇒ “BCBA”

- （X の i 文字目以降, Y の j 文字目以降）で作れる共通部分列は同じ

最後に

DP に強くなるために

- 慣れるために，自分で何回かやってみましょう

- 典型的な問題を知りましょう
  - ナップサック問題
    - 0-1，個数無制限
  - 最長増加部分列 (LIS)
  - 最長共通部分列 (LCS)
  - 連鎖行列積

ナップサック問題＝DP？ (1/2)

- ナップサック問題，ただし：
  - 商品の個数 $n \leqq 20$,
  - 価値 $v_i \leqq 10^9$, 重さ $w_i \leqq 10^9$
  - 重さの和の上限 $U \leqq 10^9$
- ナップサック問題だから DP だ！

とは言わないように！

ナップサック問題＝DP？ (2/2)

- DP　・・・　O(nU) ⇒ 20×10^9
- 全探索・・・$O(2^n)$ ⇒ $2^{20} \fallingdotseq 10^6$

- 全探索の方が高速な場合もある

- DP に限らず，アルゴリズムは手段であり目的ではありません
  - 無理に使わない

おしまい

お疲れ様でした

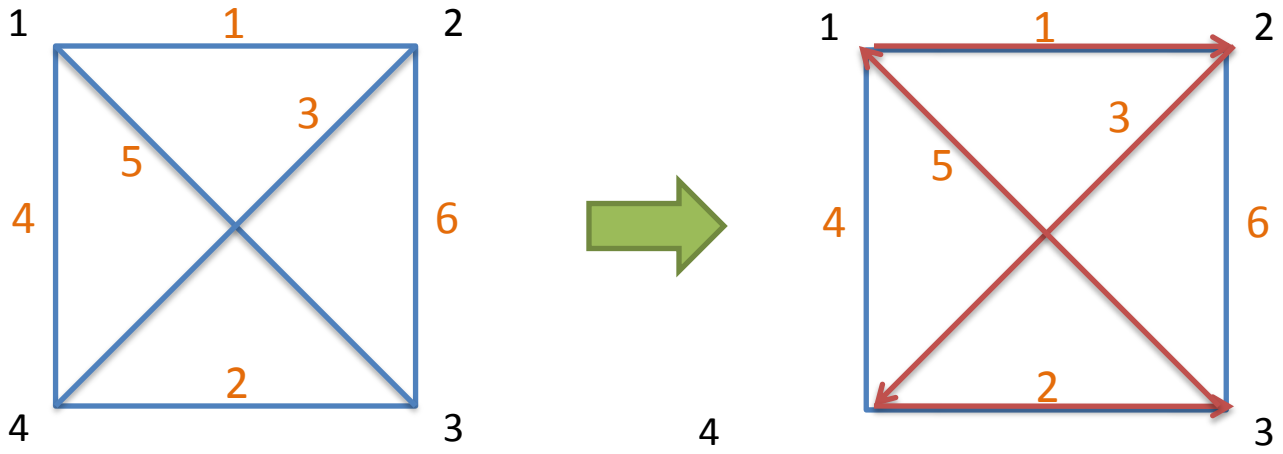
補足スライド

集合を状態にする DP

ビット DP

TSP (1/2)

- 巡回セールスマン問題 (TSP)
  - n 個の都市があって，それぞれの間の距離がわかっています
  - 都市 1 から全部の都市に訪れ，都市 1 に戻る
  - 最短の移動距離は？



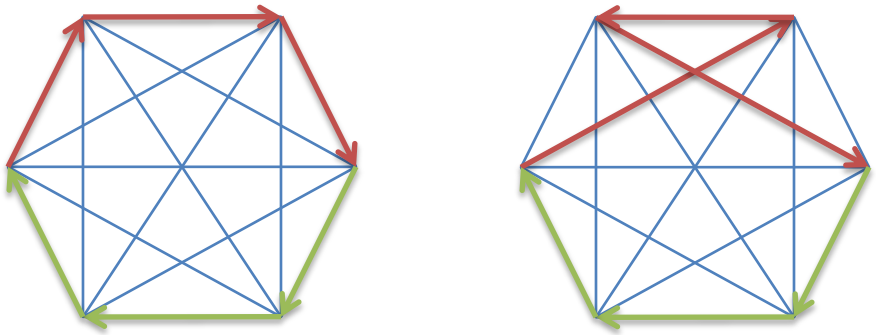
TSP (2/2)

- NP 困難問題なので，効率的な解法は期待できない
- 全探索は $O(n!)$ 時間
  - $n \leqq 10$ 程度まで
- DP を用いると $O(n^2 2^n)$ 時間にできる
  - $n \leqq 16$ 程度まで

TSP の DP の状態 (1/2)

- 状態は，下の 2 つの組
  - 既に訪れている街はどれか・・・2^n 通り
  - 今いる街はどれか・・・n 通り

- そこまでの順番に関わらず，その後の最適な道のりは等しい



赤色：そこまでの道のり，緑色：そこからの最適な道のり

TSP の DP の状態 (2/2)

- 「既に訪れている街はどれか」 (2^n 通り)

- これを，ビットマスクで表現
  - i ビット目が 1 なら訪れている
  - i ビット目が 0 なら訪れていない

ビットマスクの利点

- 状態が 0 から 2^n-1 にエンコードされ，そのまま配列を利用できる

- 集合の包含関係が数値の大小関係
  - 集合 A, B について $A \subset B$ ならば
  - ビット表現で $a \leqq b$

ビットマスクを用いた全探索

```
int N, D[30][30];     // 頂点数，距離行列

// 今頂点 v に居て，既に訪れた頂点のビットマスクが b
int tsp(int v, int b) {
        if (b == (1 << N) - 1) return D[v][0];              // 全部訪れた

        int res = INT_MAX;
        for (int w = 0; w < N; w++) {
                if (b & (1 << w)) continue;                 // もう訪れた
                res = min(res, D[v][w] + tsp(w, b | (1 << w)));
        }
        return res;
}
```

外からは tsp(0, 1) を呼ぶ

動的計画法にする

```
int tsp(int v, int b) {
        …
}
```

- 配列 memo[MAX_N][1 << MAX_N] 等を作って，tsp(v, b) の結果を記録すれば良い

- ループで記述するのも簡単
  - b は降順に回せばよい

2 つの実装方法の違い

メモ探索 vs 動的計画法

メモ探索の利点

- ループの順番を考えなくて良い
  - ツリー，DAG の問題
- 状態生成が楽になる場合がある
  - 状態が複雑な問題
- 起こり得ない状態を計算しなくてよい

動的計画法 (ループ) の利点

- メモリを節約できる場合がある
  - 今後の計算で必要になる部分のみ覚えていれば良い
- 実装が短い
- 再帰呼び出しのオーバーヘッドがない

配る DP と貰う DP

- 配る DP
  - 各場所の値を計算してゆくのではなく，各場所の値を他のやつに加えてゆくような DP
  - メモ探索ではできない
- 確率，期待値の問題で有用なことがある

おしまい