# 競プロDP発想ガイド

## どういうときにDPを使うのか

Reo Kambayashi — 2026-04-18

# Agenda

**前半: 発想編**

1. **全探索→DPへの道筋**
2. **DPが答える3種類の問い**
3. **他の手法との境界線**
4. **状態設計と同一視**
5. **テクニック集・遷移設計**
6. **制約から逆算する**
7. **シグナル集・体系マップ**

**後半: パターン詳解**

8. ナップサック・LIS/LCS・グリッド
9. 区間DP・木DP・DAG上DP
10. ビットDP・桁DP
11. 期待値DP・ゲームDP

> 本スライドの目的: DPを「知っている」から「見抜ける」へ。パターンの暗記ではなく、**発想の型**を身につける。

# 全探索からDPへ

DPは「賢い全探索」にすぎない

# 全探索 → メモ化 → DP の3ステップ

1. **再帰的な全探索を書く:** まず「正しいが遅い」解法を作る。書けないならDPではない
2. **重複を発見する:** 同じ引数で何度も呼ばれる関数があるか？ → あればメモ化で高速化
3. **ボトムアップに変換:** メモ化再帰をループに書き換える＝DPテーブルの完成

## 例: フィボナッチ数

```
# Step 1: 全探索（指数時間）
def fib(n):
    if n <= 1: return n
    return fib(n-1) + fib(n-2)

# Step 2: メモ化（O(N)）
memo = {}
def fib(n):
    if n in memo: return memo[n]
    ...
```

## 本質

- DPは「全探索＋重複除去」
- 全探索が書けない問題にDPは使えない
- 「再帰で書ける ＝ 部分問題に分解できる」

> **判定基準:** 再帰で書ける → 重複がある → DP

# なぜ全探索から始めるのか

### 全探索 = 正しさの保証

- 全探索は**必ず正しい解**を出す
- DPは全探索の「計算を省略する版」
- 省略の正当性は全探索の構造から導かれる

### いきなりDP式を考えない

- 「dp[i][j] = ...」をいきなり立てようとしない
- まず再帰関数を書き、引数が状態になる

### 再帰関数の引数 = DPの状態

```
# 再帰の引数が (i, w) → dp[i][w]
def solve(i, w):
    if i == n: return 0
    # 選ばない
    res = solve(i+1, w)
    # 選ぶ
    if w >= weight[i]:
        res = max(res, solve(i+1,
            w - weight[i]) + value[i])
    return res
```

> 再帰関数の引数をそのまま dp テーブルの添字にすればよい

# DPが答える3種類の問い

DPは万能ではない。DPが解けるのは以下の3タイプの問いだけ:

### 最適化

「最小コスト」「最大利益」

```
dp[i] = min(dp[j] + cost)
dp[i] = max(dp[j] + value)
```

遷移で `min` / `max` を取る

### 数え上げ

「何通りあるか」「場合の数」

```
dp[i] += dp[j]
```

遷移で**加算**する

### 判定（Yes/No）

「可能か不可能か」

```
dp[i] |= dp[j]
```

遷移で **OR** を取る（部分和問題）

> **まずこのフィルターを通す:** 問題の問いかけがこの3タイプに当てはまるか確認する。当てはまらなければDPではない。

# DPが使えない問題 — 2条件の確認

DPには**2つの前提条件**がある。どちらか欠ければDPは適用できない。

### ① 最適部分構造

全体の最適解が部分問題の最適解から**構成できる**

✓ 最短経路: A→B→C が最短なら A→B も最短

✗ 最長単純パス: 部分の最長が全体の最長に使えない

### ② 重複部分問題

同じ部分問題が**複数回**現れる

✓ フィボナッチ: fib(3) が何度も呼ばれる

✗ マージソート: 各部分配列は1回しか処理しない

**片方だけ成り立つ場合:** 最適部分構造あり＋重複なし → 分割統治で十分（DPは不要）。最適部分構造なし → DPでは原理的に解けない（貪欲・全探索・専用アルゴリズムを使う）。

# 例1: 最長単純パス — 最適部分構造が壊れる

### 問題

一般グラフで、同じ頂点を2度通らずに最も長いパスを求めよ（NP困難）

### なぜDPが使えないか

```
グラフ:
  q --- r --- t
  |   / |
  s --  |
   \---/

全体の最長 q→t: q→s→r→t（長さ3）
```

部分問題 q→r の最長は q→s→t→r

→ **t を先に使ってしまい**、全体のパスに組み込めない

### 核心:「使用済み頂点」が問題間で干渉

- 最短経路では「中間の頂点を2回通っても損するだけ」→ 部分問題が独立
- 最長パスでは「中間の頂点を使うと他のパスで使えなくなる」→ **部分問題が依存**

### DAGなら解ける理由

- 閉路がないので、一方向にしか進めない
- 「使用済み頂点」の干渉が起こらない
- → トポロジカル順序でDP可能

> **教訓:** 「同じリソースの取り合い」がある問題はDPの最適部分構造が壊れやすい

# 例2: 貪欲が最適 — DPは過剰

DPが「使えない」のではなく「不要」なケース。貪欲法で最適解が保証される。

### 区間スケジューリング

N個の仕事 [開始, 終了] から最大個数を選ぶ

```
貪欲: 終了時刻が早い順に選ぶ → 最適
DP:   解けるが O(N²) で遅い
```

**なぜ貪欲が最適か:** 終了が早い仕事を選ぶと残り時間が最大化される（交換論法で証明可能）

### 最小全域木

Kruskalは辺をコスト昇順に追加するだけで最適。

DPでは状態の定義自体が困難。

### 見分けるポイント

| 特徴 | 貪欲 | DP |
|---|---|---|
| 選択の影響 | 後に残らない | 後に残る |
| 最適性の証明 | 交換論法 | 漸化式 |
| 一度の選択で | 確定できる | 確定できない |

### 貪欲が正しいか確認する方法

1. 貪欲で解を作る
2. **反例を探す**: 貪欲で最適にならない入力はあるか？
3. 反例があれば → DPを検討

# 例3: 専用アルゴリズムが必要な問題

DPの枠組みでは効率的に解けず、問題固有のアルゴリズムが必要。

| 問題 | なぜDPが不向きか | 正しい手法 | 計算量 |
|---|---|---|---|
| 最大フロー | 増加路の選び方で解が変わり、部分構造が複雑 | Ford-Fulkerson / Dinic | $O(V^2E)$ |
| 二部マッチング | 局所的に最適なペアリング ≠ 全体最適 | Hopcroft-Karp | $O(E\sqrt{V})$ |
| 最短経路（非負辺） | DPでも解けるがグラフが密なとき非効率 | Dijkstra | $O(E \log V)$ |
| 強連結成分分解 | 構造発見問題でありDP向きでない | Tarjan / Kosaraju | $O(V+E)$ |
| 文字列検索 | パターンマッチは状態機械が効率的 | KMP / Aho-Corasick | $O(N+M)$ |

**注意:** 上の一部はDPで解くことも理論上は可能だが、専用アルゴリズムの方が計算量・実装の両面で優れる。「DPで解けるか」ではなく「DPで解くべきか」を考える。

# 他の手法との境界線

貪欲法・分割統治との使い分け

# 貪欲法 vs DP — どちらを使うか

### 貪欲法が使える条件

- **局所最適 = 全体最適**が成り立つ
- 一度決めた選択を後から覆す必要がない
- 交換論法やマトロイドで正当性が証明できる

### 貪欲が使える例

- 区間スケジューリング（終了時刻でソート）
- ダイクストラ法（非負辺の最短経路）
- ハフマン符号

### DPに切り替えるサイン

- **反例が作れる**: 貪欲で最適にならない入力がある
- **選択の影響が後に残る**: 今の選択が将来の選択肢を変える
- **最適解の構造が複雑**: 単純な「一番良いものから取る」では不十分

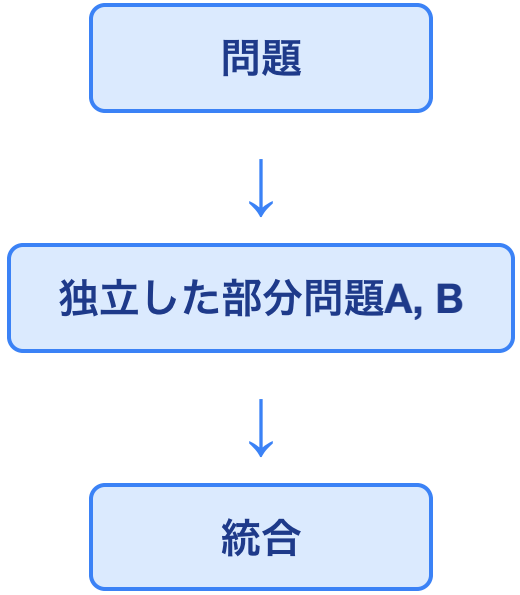
### 典型的な切り替え例

| 貪欲 ✗ → DP ✓ |
| --- |
| 0/1ナップサック（重さと価値が独立） |
| コイン問題（任意額面） |
| 最長増加部分列 |

# 分割統治 vs DP — 重複の有無が分岐点

### 分割統治

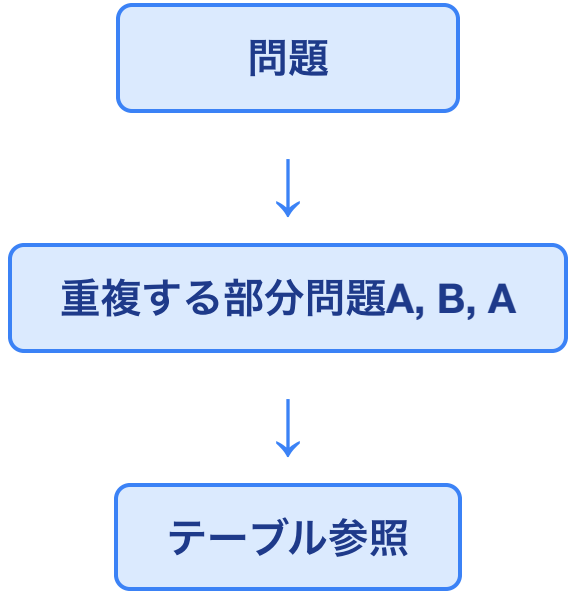
- 問題を**独立な**部分問題に分割
- 部分問題間に**重複がない**
- 各部分問題を1回だけ解く




例: マージソート、二分探索

### DP

- 部分問題が**重複する**
- 同じ部分問題を何度も使い回す
- テーブルに保存して再計算を防ぐ




例: フィボナッチ、経路数え上げ

# DP設計の核心

状態の定義と遷移の設計

# 状態設計 — 何を添字にするか

DPの難しさの80%は**状態の定義**にある。

1. **「何を決めたか」を考える:** i番目まで見た、集合Sを訪問済み、区間[l,r]を処理済み
2. **「あと何が必要か」を考える:** 残り容量w、現在のmod値、tightフラグ
3. **不要な情報を削る:** 状態に含めなくても遷移が書けるなら、それは不要

## 状態が多すぎる → 削る

✗ `dp[i][j][k][l]` → 状態爆発
✓ `dp[i][j]` で十分なら `k,l` は不要

**削れるかの判定:** k, l を固定しなくても
遷移式が書けるか？ → 書ければ不要

## 状態が足りない → 足す

✗ `dp[i]` だけでは遷移が書けない
✓ `dp[i][j]` にして情報を追加

**足すかの判定:** 「次の選択」を決めるのに
追加情報が必要か？ → 必要なら状態に入れる

**目安:** 状態数 × 遷移コスト ≤ 10^8 に収まるか？ 超えたら状態を減らす工夫が必要。

# 状態の同一視 — 全探索を圧縮する発想

DPの本質は「**同じ未来を持つ状態をまとめる**」こと。

### 全探索では区別される

コイン問題: 合計7円を作る方法

- 5+1+1 と 1+1+5 と 1+5+1 は**異なる経路**
- 全探索はすべて別々に探索する

### DPでは同一視される

- どの経路で来ても「残り金額が同じ」なら**未来は同じ**
- → `dp[残り金額]` だけで状態を表現できる

### 同一視のチェックリスト

- **過去の経路が未来に影響しないか？**
  → 影響しなければ、経路の違いを無視できる
- **どの情報だけあれば次の選択ができるか？**
  → その情報だけが状態になる
- **状態をまとめすぎていないか？**
  → まとめすぎると正しい遷移が書けなくなる

> **核心:** 「未来に影響する情報だけ」を状態にする

# 困ったときのテクニック集

### 添字と値の入れ替え（双対）

状態爆発するとき、添字と値を交換する

```
通常: dp[i][w] = 最大価値
双対: dp[i][v] = 最小重さ
```

W が巨大で V が小さいナップサックで有効

### 次元圧縮（配列使いまわし）

`dp[i]` が `dp[i-1]` だけに依存するなら、1次元配列で十分

```
// 2次元 → 1次元に圧縮
for w in W..0:  // 逆順ループが鍵
    dp[w] = max(dp[w], dp[w-w_i]+v_i)
```

### 寄与で考える（配るDPの発展）

「この要素は答えにどれだけ寄与するか」で考える

- 数え上げ問題で「各要素が何回使われるか」を数える
- 期待値問題で「各状態の寄与」を足し合わせる

### 上界・下界を利用する

- 「ありえない状態」をINFや-INFで初期化
- 不可能な遷移を自然に除外できる

```
// 部分和: 初期値は False
dp[0] = True
dp[i] = dp[i] or dp[i - a[j]]
```

# 遷移の考え方 — もらうDP vs 配るDP

### もらうDP（Pull型）

`dp[i]` を、**過去の状態から集めて**計算する

```
// dp[i] を確定させる
dp[i] = min(dp[j] + cost(j, i))
        for all valid j
```

- 直感的で書きやすい
- 多くの問題でこちらが自然

### 配るDP（Push型）

`dp[i]` から、**未来の状態に配る**

```
// dp[i] が確定済みとして配る
for each next state j:
    dp[j] = min(dp[j], dp[i] + cost(i, j))
```

- 遷移先が明確な場合に便利
- 期待値DP・確率DPで頻出

> **どちらを選ぶか:** 「あるマスの値を決めるのに、どこから来たか考えやすい」→ もらうDP。「あるマスから、どこに行けるか考えやすい」→ 配るDP。結果は同じ。

# 制約からパターンを逆算する

制約の数値は**出題者からのヒント**。計算量の上限からDPの型を逆算できる。

| 制約 | 許容計算量 | 第一候補のDP |
|---|---|---|
| N ≤ 15〜20 | O(2^N · N) | ビットDP |
| N ≤ 300〜500 | $O(N^3)$ | 区間DP |
| N ≤ 3000〜5000 | $O(N^2)$ | LCS・2次元DP |
| N ≤ 10^5 | O(N log N) | LIS（二分探索） |
| N ≤ 10^5, W ≤ 10^5 | O(NW) | ナップサック |
| 10^18 以下の整数 | O(桁数 × 状態) | 桁DP |

> **実践テクニック:** 制約を見て「$O(N^2)$ が通る → 2次元DPか」「N ≤ 18 → ビットDPか」と仮説を立ててから問題を読むと、解法に辿り着くのが速い。

# 問題文のシグナル集

| キーワード | DPの種類 | 典型例 |
| --- | --- | --- |
| 「最小コスト」「最大利益」 | 最適化DP | 最小コスト経路 |
| 「〜の場合の数」「通り数」 | 数え上げDP | 経路の数え上げ |
| 「〜以下で条件を満たす数の個数」 | 桁DP | N以下で各桁の和がK |
| 「全部訪問する最短」「順列の最適化」 | ビットDP (TSP) | 巡回セールスマン |
| 「部分列」「連続でなくてよい」 | LIS / LCS | 最長増加部分列 |
| 「区間をまとめる」「分割コスト」 | 区間DP | 行列連鎖積 |
| 「木の各頂点で選ぶ/選ばない」 | 木DP | 最大独立集合 |
| 「確率」「期待値」「期待回数」 | 期待値DP | すごろくの期待値 |
| 「二人交互に」「先手必勝か」 | ゲームDP | Nim / Grundy数 |

# DPパターン体系マップ

### 基本パターン

- **ナップサック系**
  選択 × 容量制約
- **LIS / LCS 系**
  部分列の最適化
- **グリッドDP**
  2D経路・移動
- **線形DP**
  配列を左から順に

### 構造パターン

- **区間DP**
  区間の分割・統合
- **木DP**
  部分木の再帰構造
- **DAG上DP**
  トポロジカル順序

### 特殊パターン

- **ビットDP**
  集合を状態に圧縮
- **桁DP**
  各桁を上から決定
- **期待値 / ゲームDP**
  確率的遷移・対戦

> **発想の核心:** すべてのDPは「状態をどう定義するか」「遷移をどう設計するか」に帰着する。パターンごとに状態空間と遷移の型が異なる。

# パターン選択チートシート

| 問題の特徴 | 第一候補 | 状態の形 |
| --- | --- | --- |
| 選ぶ/選ばない × 容量制約 | ナップサック | `dp[i][w]` |
| 部分列（順序保持、不連続OK） | LIS / LCS | `dp[i]` or `dp[i][j]` |
| グリッド上の移動・経路 | グリッドDP | `dp[i][j]` |
| 配列を左から順に処理 | 線形DP | `dp[i]` |
| 区間の分割・統合コスト | 区間DP | `dp[l][r]` |
| 木・部分木の集約 | 木DP | `dp[v]` |
| N ≤ 20 で全訪問・順列 | ビットDP | `dp[S][v]` |
| N以下で桁ごとの条件 | 桁DP | `dp[i][tight][...]` |
| 確率的遷移・期待回数 | 期待値DP | `dp[state]` |
| 二人ゲーム・勝敗判定 | ゲームDP | `dp[state]` |

# パターン詳解

各パターンの状態設計・遷移式・典型問題

# ナップサック系 — 選択 × 制約

**発想トリガー:** 「N個の中から選ぶ」「容量/重さの上限がある」

## 状態設計

`dp[i][w]` = i番目まで見て、容量wのときの最大値

## 遷移

```
dp[i][w] = max(
  dp[i-1][w],          // 選ばない
  dp[i-1][w-w_i] + v_i  // 選ぶ
)
```

## 派生パターン

| 種類 | 特徴 |
| --- | --- |
| 0/1 | 各品物1個まで |
| 個数制限付き | 各k個まで |
| 無制限 | 何個でもOK |
| 部分和 | 合計ちょうどW作れるか |

> **見抜くコツ:** 「制約付き最適選択」はほぼナップサック。制約が2軸（重さ+体積）なら3次元DPに拡張。

# LIS / LCS 系 — 部分列の最適化

**発想トリガー:** 「順序を保ったまま」「連続でなくてよい」「共通部分」

### LIS（最長増加部分列）

`dp[i]` = a[i] を末尾とするLIS長

```
dp[i] = max(dp[j] + 1)
        (j < i, a[j] < a[i])
```

二分探索で **O(N log N)** に高速化可能

### LCS（最長共通部分列）

`dp[i][j]` = S[1..i] と T[1..j] のLCS長

```
dp[i][j] =
  dp[i-1][j-1] + 1      (S[i]=T[j])
  max(dp[i-1][j],
      dp[i][j-1])        (S[i]≠T[j])
```

> **LIS ≠ 連続部分列:** 「連続部分配列の最大和」は Kadane のアルゴリズム（貪欲的DP）。不連続ならLIS/LCS系を疑う。

# グリッドDP / 線形DP

**発想トリガー:** 「盤面を移動」「左上→右下」「配列を左から順に処理」

## グリッドDP

`dp[i][j]` = (i,j) に到達するときの最適値

```
dp[i][j] = min(
  dp[i-1][j],   // 上から
  dp[i][j-1]    // 左から
) + cost[i][j]
```

移動方向が限定されている＝DAG構造

## 線形DP

`dp[i]` = i番目まで処理した最適値

```
// 例: 連続部分和の最大
dp[i] = max(a[i], dp[i-1] + a[i])
```

- 最も基本的なDP形式
- 添字 = 位置、値 = 最適値 or 場合の数

# 区間DP / 木DP / DAG上DP

## 区間DP

**トリガー:** 「分割」「括弧」「まとめるコスト」

`dp[l][r]` = 区間 [l, r] の最適値

```
dp[l][r] = min over k in [l, r-1]:
  dp[l][k] + dp[k+1][r] + cost(l,k,r)
```

区間の長さが短い方からループ。O(N³)

## 木DP

**トリガー:** 「部分木」「各頂点で選ぶ/選ばない」

`dp[v]` = 頂点 v を根とする部分木の最適値

```
dp[v][0] = Σ max(dp[c][0], dp[c][1])
dp[v][1] = val[v] + Σ dp[c][0]
```

DFS帰りがけで計算。全方位木DPで根の付替えも可

> **DAG上DP:** トポロジカル順序 = DPの計算順序。グリッドDPも木DPも、本質的にはDAG上のDP。

# ビットDP / 桁DP

## ビットDP

**トリガー:** 「N個すべて訪問」「N ≤ 20」

`dp[S][v]` = 集合 S 訪問済み、最後が v

```
dp[S|(1<<u)][u] =
  min(dp[S][v] + cost[v][u])
```

O(2^N · N²)。TSP・ハミルトンパス・割当問題

## 桁DP

**トリガー:** 「N以下で条件を満たす数」

`dp[i][tight][...]` = i桁目まで決めた状態

```
for d in 0..limit:
  new_tight = tight & (d == N[i])
  dp[i+1][new_tight][...]
    += dp[i][tight][...]
```

tight フラグで上限制約を管理

**共通点:** どちらも「状態を圧縮して表現する」パターン。ビットDPは集合を整数に、桁DPは数値を桁列に変換する。

# 期待値DP / ゲームDP

## 期待値 / 確率DP

**トリガー:** 「期待回数」「確率」「ランダム遷移」

`dp[i]` = 状態 i からゴールまでの期待回数

```
// すごろく型: ゴールから逆算
dp[i] = 1 + (1/6)Σ dp[i+k]
                k=1..6
```

- **逆方向DP**: ゴール→スタートに向かって計算
- 期待値の線形性を活用する

## ゲームDP

**トリガー:** 「二人で交互に」「勝敗」「最適戦略」

`dp[state]` = 手番プレイヤーが勝つか

```
// Grundy数 / minimax
dp[state] = 遷移先すべてが
  負け → 勝ち
  1つでも負けなし → 負け
```

- Sprague-Grundy 定理
- 先手/後手の区別 → 状態に含める

**共通点:** どちらも「状態遷移グラフ上のDP」。遷移が確率的か選択的かが違うだけ。

# まとめ

- まず**全探索を書く** → 重複があれば DP（いきなりDP式を立てない）
- 貪欲で**反例が作れる**なら DP に切り替える
- **再帰関数の引数** = DPの状態。不要な情報は削り、足りなければ足す
- **制約の数値**から計算量を逆算してパターンを絞り込む
- **問題文のキーワード**で候補を特定 → チートシートで確認

# 参考文献


- Educational DP Contest — AtCoder — 26問で全パターンを網羅
- 典型的な DP のパターンを整理 — drken (Qiita) — ナップサック系の体系的解説
- 動的計画法超入門！ Educational DP Contest A〜E — drken (Qiita) — 入門〜中級の解説と類題集
- 20 Patterns to Master Dynamic Programming — AlgoMaster — 英語圏の体系的パターン集
- DP Tutorial and Problem List — Codeforces — 大量の問題リスト付き
- Introduction to Dynamic Programming — cp-algorithms — 理論と実装の両面をカバー