

## I. 動的計画法 (Dynamic Programming) 基本編

---

動的計画法という競技プログラミングの最初の山となるアルゴリズムを紹介します。基本的な考え方はシンプルですが、さまざまな応用先がある重要なアルゴリズムです。動的計画法の原則は以下のようなものです。

- 与えられた問題を部分問題に分割する
- 部分問題の解を記録して再利用する

一度計算したものを記録して再び計算せずに利用することで効率的に計算を行うことができます。DP は慣れるまで難しいですが、慣れると非常に強力なアルゴリズムです。習うよりも慣れろということで、DP は具体的な問題を解きながら学ぶのが良いでしょう。

### i. フィボナッチ数列

最も簡単な DP を紹介します。フィボナッチ数列を求める問題です。フィボナッチ数列は以下のように定義されます。

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \quad (n \geq 2)$$

この定義に基づいて、フィボナッチ数列を求めるプログラムを再帰関数を用いて実装すると以下のようにになります。

コード 1 再帰関数を用いたフィボナッチ数列の実装

```
1 def fib(n):
2     if n == 0:
3         return 0
4     if n == 1:
5         return 1
6     return fib(n-1) + fib(n-2)
```

実装はシンプルですが、 $n = 40$  程度の大きさになると計算に時間がかかります。これは再帰関数を用いることで、同じ計算を何度も行っているためです。この問題を解決するために、DP を用いてフィボナッチ数列を求めるプログラムを実装します。

コード 2 DP を用いたフィボナッチ数列の実装

```
1 def fib(n):
2     dp = [0] * (n+1)
```

```

3 dp[0] = 0
4 dp[1] = 1
5 for i in range(2, n+1):
6     dp[i] = dp[i-1] + dp[i-2]
7 return dp[n]

```

フィボナッチ数列の例を通じて、漸化式の計算を効率的に行うために DP を用いることができることがわかりました。

## ii. 最大値問題

次の DP の例題として、最大値問題を紹介します。最大値問題は、与えられた数列の中からいくつかの数を選んで総和を最大化する問題です。例えば、数列  $[1, 2, 3, 4, 5]$  が与えられたとき、この数列の中からいくつかの数を選んで総和を最大化する問題を考えます。配列の中の正の数をすべて足せばいいですが、今回は DP を用いて解いてみます。 naïve な実装では配列の要素を選ぶか選ばないかの  $O(2^n)$  の計算量になります。

DP には 2 つの方針としてメモ化再帰と漸化式があります。メモ化再帰は再帰関数を用いて大きい問題を分割する方法で、漸化式は小さい問題から順に解いていき大きい問題を解くための漸化式を求める方法です。

### 1. メモ化再帰

メモ化再帰を用いて最大値問題を解くプログラムを実装します。メモ化再帰は再帰関数を用いて計算を行い、計算結果を記録して再利用する方法です。以下にメモ化再帰を用いた最大値問題のプログラムを示します。メモ化再帰の実装では、戻り値でメモした値を返すようにプログラムを書くことがポイントです。再帰を  $n$  から始めて徐々に小さい問題に分割していくことで、最終的に大きい問題を解くことができます。

コード 3 メモ化再帰を用いた最大値問題の実装

```

1 def max_sum(A: list[int], memo: dict[int, int], limit: int):
2     if limit < 0:
3         return 0
4
5     if limit in memo:
6         return memo[limit]
7
8     choose_current = A[limit] + max_sum(A, memo, limit - 1)
9     skip_current = max_sum(A, memo, limit - 1)
10

```

```

11 memo[limit] = max(choose_current, skip_current)
12 return memo[limit]

```

## 2. 漸化式

漸化式はメモ化再帰とは異なり、小さい問題から順に解いていき大きい問題を解くための漸化式を求める方法です。最大値問題では  $\text{index}(\leq n-1)$  は直前の値に足すか足さないかの2通りがあるため、漸化式は以下のようになります。

$$dp[i] = \max(dp[i-1] + A[i], dp[i-1])$$

この漸化式を用いて最大値問題を解くプログラムを実装します。

コード 4 漸化式を用いた最大値問題の実装

```

1 def max_sum(A: list[int]) -> int:
2     dp = [0] * len(A)
3     # 初期化
4     dp[0] = max(A[0], 0)
5
6     for index, a in enumerate(A, 1):
7         dp[index] = max(dp[index-1], dp[index-1] + A[index])
8
9     return dp[len(A) - 1]

```

これで基本の DP の考え方は理解できたと思います。問題 1 を解いてみましょう。EDPC の基本問題です。メモ化再帰と漸化式の両方で解いてみてください。

### iii. コイン問題

以下の問題を考えてみましょう。今までの DP の考え方に加えて少し応用を加えた問題です。後で扱うナップサック問題の基本となる問題です。

#### 問題

$m$  種類のコインがあり、それぞれのコインは無限にあるとする。コインの価値がそれぞれ  $c_1, c_2, \dots, c_m$  であるとき、合計金額が  $n$  円になるようにコインを選ぶ方法の最小枚数を求めよ。

価値が大きい方から選ぶという貪欲法では最適解が求められないため、DP を用いて解く必要があります。こちらの問題もメモ化再帰と漸化式の両方で解いてみます。

コード 5 メモ化再帰を用いたコイン問題の実装

```
1 def minimum_coin_counts(coins: list[int], memo: dict[int], limit: int) ->
  int:
2     if limit in coins:
3         memo[limit] = 1
4         return 1
5
6     if limit in memo:
7         return memo[limit]
8     if limit - coins[0] > 0:
9         memo[limit] = minimum_coin_counts(coins, memo, limit - coins[0]) +
            1
10
11    for coin in coins:
12        if limit - coin > 0:
13            memo[limit] = min(minimum_coin_counts(coins, memo, limit - coin
                ) + 1, memo[limit])
14
15    return memo[limit]
```

コード 6 漸化式を用いたコイン問題の実装

```
1 def minimum_coin_count(m: int, n: int, coins: list[int]) -> int:
2     # dp table
3     dp = [1 << 60] * (n + 1)
4     # 初期化
5     dp[0] = 0
6
7     for i in range(1, n + 1):
8         for coin in coins:
9             if i - coin >= 0:
10                dp[i] = min(dp[i], dp[i - coin] + 1)
11
12    return dp[n]
```

メモ化再帰と漸化式の両方で、「ずらした値 + コスト」( $dp[i - \text{coin}] + 1$ ) という形で解いていることがわかります。動的計画法において前のどの結果を使うのかを考えることが重要です。続くナップサック問題でも同様の考え方が使われます。

メモ化再帰と漸化式どちらが考えやすいかは人それぞれですが、漸化式の方が実装が簡単であることが多いです。

#### iv. ナップサック問題

ナップサック問題は DP の中でも有名な問題です。ナップサック問題は以下のような設定の問題です。

##### 問題

重さが  $w_1, w_2, \dots, w_n$  で価値が  $v_1, v_2, \dots, v_n$  である  $n$  個の品物があり、重さの総和が  $W$  を超えないようにナップサックに詰め込むとき、価値の総和の最大値を求めよ。

各荷物について品物を入れるか入れないかの 2 通りがあるため、0/1 ナップサック問題と呼ばれます。先ほどまでの問題とは違って価値と重さという 2 つの基準がある問題です。dp テーブルが 2 次元の配列になるため、実装が少し複雑になります。ナップサック問題はメモ化再帰と漸化式の両方で解いてみましょう。

##### コード 7 メモ化再帰を用いたナップサック問題の実装

```
1 def recursive_knapsack(item_limit: int, weight_limit: int, memo: dict[
2     tuple[int, int], int], items: list[tuple[int, int]]) -> int:
3     if (item_limit, weight_limit) in memo:
4         return memo[(item_limit, weight_limit)]
5
6     if item_limit == 0 or weight_limit == 0:
7         return 0
8
9     value, weight = items[item_limit]
10
11     not_taken = recursive_knapsack(item_limit - 1, weight_limit, memo,
12         items)
13
14     taken = 0
15     if weight_limit - weight >= 0:
16         taken = value + recursive_knapsack(item_limit - 1, weight_limit -
17             weight, memo, items)
18
19     memo[(item_limit, weight_limit)] = max(not_taken, taken)
20
21     return memo[(item_limit, weight_limit)]
```

```
19
20 def main():
21     n, w = map(int, input().split())
22     items = [(0, 0)]
23     for i in range(n):
24         value, weight = map(int, input().split())
25         items.append((value, weight))
26
27     memo = dict()
28
29     ans = recursive_knapsack(n, w, memo, items)
30
31     print(ans)
32
33 if __name__ == "__main__":
34     main()
```

コード 8 漸化式を用いたナップサック問題の実装

```
1 def knapsack(n: int, w: int, items: list[tuple[int, int]]) -> int:
2     dp = [[0] * (w + 1) for _ in range(n + 1)]
3
4     for i in range(1, n + 1):
5         value, weight = items[i]
6         for j in range(1, w + 1):
7             if j - weight >= 0:
8                 dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight] +
9                                 value)
10            else:
11                dp[i][j] = dp[i - 1][j]
12
13     return dp[n][w]
```

## v. 問題

問題 1 EDPC A - Frog 1

問題 2 AtCoder Beginner Contest 032 ナップサック問題

### 3 動的計画法 応用編

---

上で扱ったナップサック問題とは少しだけ設定が異なる問題です。

## II. 参考

---

- <https://dai1741.github.io/maximum-algo-2012/docs/dynamic-programming/>
- 

## III. 動的計画法 応用編

---

- i. レーベンシュタイン距離
- ii. 最長共通部分列 (LCS)
- iii. 動的時間伸縮法