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。

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(n) &= F(n-1) + F(n-2) \quad (n \geq 2)
\end{aligned}
$$

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

```
def fib(n):
  if n == 0:
    return 0
  if n == 1:
    return 1
  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

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

```
  dp[0] = 0
  dp[1] = 1
  for i in range(2, n+1):
    dp[i] = dp[i-1] + dp[i-2]
  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。ナイーブ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

```
def max_sum(A: list[int], memo: dict[int, int],limit: int):
  if limit < 0:
      return 0

  if limit in memo:
      return memo[limit]

  choose_current = A[limit] + max_sum(A, memo, limit - 1)
  skip_current = max_sum(A, memo, limit - 1)

```

```
    memo[limit] = max(choose_current, skip_current)
    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 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

```
def max_sum(A: list[int]) -> int:
    dp = [0] * len(A)
    # 初期化
    dp[0] = max(A[0], 0)

    for index, a in enumerate(A, 1):
        dp[index] = max(dp[index - 1], dp[index - 1] + A[index])

    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 $c_1, c_2, \ldots, c_m$ 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

```python
def minimum_coin_counts(coins: list[int], memo: dict[int], limit: int) ->
    int:
  if limit in coins:
      memo[limit] = 1
      return 1

  if limit in memo:
      return memo[limit]
  if limit - coins[0] > 0:
      memo[limit] = minimum_coin_counts(coins, memo, limit - coins[0]) +
          1

  for coin in coins:
      if limit - coin > 0:
          memo[limit] = min(minimum_coin_counts(coins, memo, limit - coin
              ) + 1, memo[limit])

  return memo[limit]
```

コード 6　漸化式を用いたコイン問題の実装

```python
def minimum_coin_count(m: int, n: int, coins: list[int]) -> int:
  # dp table
  dp = [1 << 60] * (n + 1)
  # 初期化
  dp[0] = 0

  for i in range(1, n + 1):
      for coin in coins:
          if i - coin >= 0:
              dp[i] = min(dp[i], dp[i - coin] + 1)

  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 - 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 $w_1, w_2, \ldots, w_n$ で価値が $v_1, v_2, \ldots, v_n$ 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

```python
def recursive_knapsack(item_limit: int, weight_limit: int, memo: dict[
    tuple[int, int], int], items: list[tuple[int, int]]) -> int:
    if (item_limit, weight_limit) in memo:
        return memo[(item_limit, weight_limit)]

    if item_limit == 0 or weight_limit == 0:
        return 0

    value, weight = items[item_limit]

    not_taken = recursive_knapsack(item_limit - 1, weight_limit, memo,
        items)

    taken = 0
    if weight_limit - weight >= 0:
        taken = value + recursive_knapsack(item_limit - 1, weight_limit -
            weight, memo, items)

    memo[(item_limit, weight_limit)] = max(not_taken, taken)

    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] +
                       value)
9              else:
10                 dp[i][j] = dp[i - 1][j]
11
12     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

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

動的計画法は幅広い応用先があります。ここでは、動的計画法の応用例を紹介します。

i.　レーベンシュタイン距離

レーベンシュタイン距離とは、ある文字列 A と別の文字列 B の類似度を計算するための指標の 1 つです。レーベンシュタイン距離は、文字列 A から文字列 B に変換するために必要な操作回数の最小回数を表します。操作は以下の 3 つがあります。

- 挿入: 文字列 A に文字を挿入する
- 削除: 文字列 A から文字を削除する
- 置換: 文字列 A の文字を別の文字に置換する

例えば、kitten と setting のレーベンシュタイン距離は 3 です。以下のように計算できます。

- kitten → sitten （置換: $k \to s$）
- sitten → settin （置換: $i \to$e）
- settin → setting （挿入: g）

漸化式タイプの動的計画法を使ってレーベンシュタイン距離を考えてみましょう。簡単のために文字列 A、B はそれぞれ abc, adcd とします。dp テーブルの各セル (i, j) は文字列 A の i 文字目までと文字列 B の j 文字目までのレーベンシュタイン距離を表します。dp テーブルでの処理と文字列操作の対応を考えましょう。

1 列目と 1 行目は空文字列です。空文字列から文字列 A の i 文字目までのレーベンシュタイン距離は i です。同様に空文字列から文字列 B の j 文字目までのレーベンシュタイン距離は j です。dp テーブルは以下のようになります。

	""	a	b	d	c
""	0	1	2	3	4
a	1				
b	2				
c	3				

1 列目を考えると、空文字列から文字列 B の i 文字目までのレーベンシュタイン距離を表していますが、右への移動は**挿入**の操作を表しています。同様に 1 行目を考えると、文字列 A の i 文字目から空文字列へのレーベンシュタイン距離を表しており、下への移動は**削除**の操作を表しています。テーブルで斜めに進むことは、文字列操作においては**置換**または**何もしない**の操作を表しています。テーブル上で削除してから追加する（下に行ってから右に行く）という操作は、置換と等価です。追加してから削除する（右に行ってから下に行く）という操作は、何もしないと等価です。よって、dp テーブルの更新は以下のようになります。

- 追加: dp[i][j] = dp[i][j-1] + 1
- 削除: dp[i][j] = dp[i-1][j] + 1
- 置換: dp[i][j] = dp[i-1][j-1] + 1
- 何もしない: dp[i][j] = dp[i-1][j-1]

完成した dp テーブルは以下のようになります。

	""	a	b	d	c
""	0	1	2	3	4
a	1	0	1	2	3
b	2	1	1	2	3
c	3	2	2	1	2

レーベンシュタイン距離の実装を漸化式を用いて実装してみましょう。

コード 9　レーベンシュタイン距離の実装

```
def levenshtein(str1: str, str2: str) -> int:
  dp = [[0] * (len(str2) + 1) for _ in range(len(str1) + 1)]
  # 初期化
  for i in range(1, len(str2) + 1):
      dp[0][i] = i

  for i in range(1, len(str1) + 1):
      dp[i][0] = i

  # dpテーブル更新
  for i in range(1, len(str1) + 1):
      for j in range(1, len(str2) + 1):
          if str1[i - 1] == str2[j - 1]:
              dp[i][j] = min(dp[i][j - 1] + 1, dp[i - 1][j] + 1, dp[i -
                  1][j - 1])
```

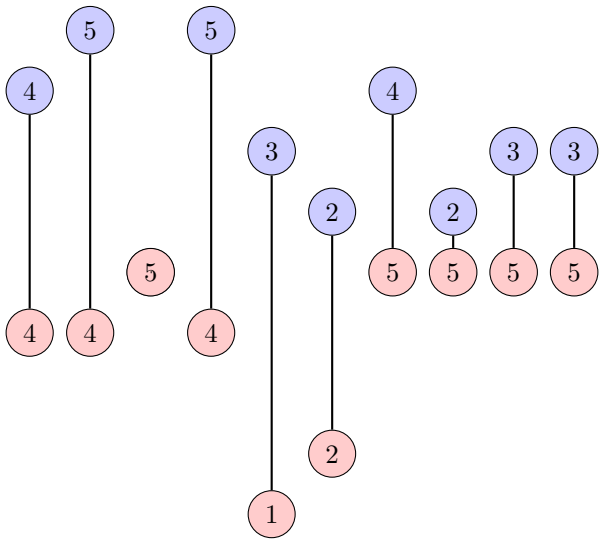
```
            else:
                dp[i][j] = min(dp[i][j - 1] + 1, dp[i - 1][j] + 1, dp[i -
                    1][j - 1] + 1)

    return dp[len(str1)][len(str2)]
```

ii.　最長共通部分列 (LCS)

iii.　動的時間伸縮法 (Ddynamic Time Warping, DTW)

2 つの時系列データの類似度を計算するための手法です。DTW は時系列データの長さが異なる場合でも、2 つの時系列データの類似度を計算することができ、弾性マッチング (Elastic Matching) とも呼ばれます。時系列データ間の類似度を計算するにはユークリッド距離を使うこともできますが、それでは時系列データの類似度をうまく測れなかったり、時系列データの長さが異なる場合に対応できません。



IV.　参考