動的計画法 (Dynamic Programming)

動的計画法は、問題を部分問題に分割し、それらを解くことで全体の問題を効率的に解決するアルゴリズム設計手法です。以下にその基本的な概念と手順をまとめます。

基本概念

1. **部分問題の分割**
   問題を小さな部分問題に分割し、それらを再帰的に解きます。

2. **重複計算の回避**
   同じ部分問題を何度も計算しないように、結果を保存して再利用します（メモ化またはテーブル化）。

3. **最適部分構造**
   問題の最適解が部分問題の最適解から構築できる性質を利用します。

手順

1. 問題を部分問題に分割する。
2. 部分問題の解を保存するデータ構造を用意する（例: 配列やハッシュテーブル）。
3. 再帰または反復を用いて部分問題を解く。
4. 保存した部分問題の解を利用して全体の問題を解く。

実装方法

- **トップダウンアプローチ (メモ化)**
  再帰を使用し、計算結果をメモ化して再利用します。

- **ボトムアップアプローチ (タブレーション)**
  小さな部分問題から順に解き、結果をテーブルに保存します。

使用例

動的計画法は以下のような問題でよく使用されます。

- フィボナッチ数列
- ナップサック問題
- 最長共通部分列（LCS）
- 最長増加部分列（LIS）
- 最短経路問題 (例: Floyd-Warshall)
- 部分和問題
- 編集距離（Edit Distance）

例

フィボナッチ数列

```java
// ボトムアップアプローチ
public class Fibonacci {
	public static int fibonacci(int n) {
		if (n <= 1) return n;
		int[] dp = new int[n + 1];
		dp[0] = 0;
		dp[1] = 1;
		for (int i = 2; i <= n; i++) {
			dp[i] = dp[i - 1] + dp[i - 2];
		}
		return dp[n];
	}

	public static void main(String[] args) {
		System.out.println(fibonacci(10)); // 出力: 55
	}
}
```

ナップサック問題

```java
// ボトムアップアプローチ
public class Knapsack {
	public static int knapsack(int[] weights, int[] values, int capacity) {
		int n = weights.length;
		int[][] dp = new int[n + 1][capacity + 1];

		for (int i = 1; i <= n; i++) {
			for (int w = 0; w <= capacity; w++) {
				if (weights[i - 1] <= w) {
					dp[i][w] = Math.max(dp[i - 1][w], dp[i - 1][w -
weights[i - 1]] + values[i - 1]);
				} else {
					dp[i][w] = dp[i - 1][w];
				}
			}
		}
		return dp[n][capacity];
	}

	public static void main(String[] args) {
		int[] weights = {2, 3, 4, 5};
		int[] values = {3, 4, 5, 6};
		int capacity = 5;
		System.out.println(knapsack(weights, values, capacity)); // 出力: 7
	}
}
```

メリットとデメリット

メリット

- 再帰的な問題を効率的に解ける。
- 計算量を大幅に削減できる。

デメリット

- メモリ使用量が多くなる場合がある。
- 問題の分割や状態の定義が難しい場合がある。

動的計画法は、効率的なアルゴリズム設計において非常に重要な手法です。問題の性質を理解し、適切に適用することが成功の鍵です。