→ Repeating Subproblems means it needs DATE Dynamic Programming

+ Overlapping problems.

+ problems have optimal substructure property,- Overall soln can be constructed from optimal solutions of its subproblems.

> Dynamic Programming solve every subproblem just once and saves the answer in a lookup table, thereby avoiding recalculating the answer everytime the subproblem is encountered.

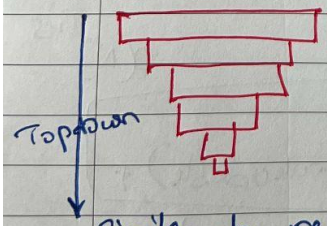
2 ways / patterns of DP: 1) memoization 2) tabulation.

How to recognize DP:-

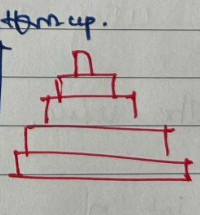
2 subproblems share the problem but subproblems are not independent.

① Overlapping subproblems:
② Optimal substructure property: optimal soln can be constructed from optimal solns of subproblems.

① Memoization



② Tabulation.



Top-down approach avoids recursion.

Typically done by filling up a lookup table, and computing the soln based on results of the table.

Similar to recursive - except for it looks for the answer in a lookup before computing the answer.

+ Lookup Table is used to store and search values.

+ Stores values in an "array" returned from a computation. Fast or linear associative.

+ Lookup-tables stores (key-values) pairs.

Fibonacci using Memoization :-

Top down

Builds as you need

```
fib( int n, int lookupTable[]) {
    if (lookupTable[n] == -1) { // if not found
        if (n <= 1) lookupTable[n] = n;
        else lookupTable[n] = fib(n-1, lookupTable)
                            + fib(n-2, lookupTable);
    }
    returns lookupTable[n]
```

O(N) T×S

Fibonacci using Tabulation :-

Bottom up

Build the whole table first & return the value.

Build everything first & lookup.

```
fib( int n, int[] lookupTable ) {
    lookupTable[0] = 0; lookupTable[1] = 1;
    for( i=2; i<=n; i++)
        lookupTable[i] = fib(n-i, lookupTable) +
                         fib(n-2, lookupTable);
```

O(N) T×S

Another Optimal:

DATE

fibonacci requires only last 2 places.

```
lastSecond = 0
last = 1
current = 1          | Initialization

for(i=2; i<=n; i++){
    current = last + lastSecond;
    lastSecond = last;
    last = current;
}
return current;
```

reducing Space.
O(N) = 1
O(1) = 5.

0/1 KNAPSACK PROBLEM.

Q. Given two integer Arrays of weights and profits, Implement a knapsack() func, where max wt. cannot exceed "capacity_weight" but it should have max. profit.

Each item can be selected only once. either take it / skip it.

A: Example:-