1. Nim

Justification for DP

Nim is a DP problem because it:

- is an optimization problem with constraints
    - The goal is to optimize your moves such that you win, and this optimization is constrained by the finite number of allowable moves (1 or 2 stones)
- has an optimal substructure
    - To solve Nim, you must solve whether the resulting game from either of your moves is winnable. When you move, you take away one or two stones, and you are left with another independent game of Nim.
- has discrete, overlapping sub-problems
    - Nim has discrete, overlapping sub-problems because to solve for your move with N spaces remaining, there are four options for you and your opponent's first two turns: $\{(1,1),(1,2),(2,1),(2,2)\}$. These four options leave you with four discrete and independent sub-problems of size $N-2$, $N-3$, $N-3$, and $N-4$. The two $N-3$ sub-problems are redundant and overlapping.

DP grid

Here is the grid for the DP solution:

N	winnability	next move
1	0	0
2	1	1
3	1	2
4	0	0
5	1	1
6	1	2
7	0	0
8	1	1
9	1	2

The dimensions are $N \times 2$ excluding the title column and title row. the winnability column represents whether a game of size N is winnable. The next move column represents the required next move: either 1 stone, 2 stones, or "haha, you can't win!" (represented by a 0)
To fill out the grid, you look at $N-1$ and $N-2$. If they are both 1, then you must lose a game with this length, because whether you choose to remove 1 or 2 stones, your opponent will always be able to win the resulting sub-game. Otherwise, you can choose the move that results in your opponent being handed the unwinnable sub-game.
The memoization is done through the Next Move column. It is always the previous row's value plus one, all mod 3. This is because every time, N increases by one, so the number of stones that you need to remove to give your opponent the unwinnable game increases by one as well. It must be %3 because there are only three possible outcomes: you are a loser, you take one, or you take two.
Looking at the grid for $N = 9$, we see that it is possible for Alex to force a win, and his first move should be to take 2 stones.

2. Fibonacci

Here is the DP table for FibDP:

N	$\mathrm{Fib}(N)$	$\mathrm{Fib}(N-1)$
0	0	-
1	1	0
2	1	1
3	2	1
4	3	2
5	5	3
6	8	5
7	13	8
8	21	13

The dimensions are $N \times 2$, excluding the title column and title row.
To fill it out, you enter the sum of $\mathrm{Fib}(N)$ and $\mathrm{Fib}(N-1)$ from the previous row as your $\mathrm{Fib}(N)$, and enter $\mathrm{Fib}(N)$ from the previous row as $\mathrm{Fib}(N-1)$.
Then you realize that this could be written as a matrix multiplication, since these are all linear combinations! Then you diagonalize the matrix and get an explicit formula, which is so much faster than this.

Anyway, this DP solution is the same as the accumulation based algorithm from Lab03. That one just didn't store the entire grid. It only stored the current row, which is all that you need.

3. Exact Change

Here is my DP grid:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0
5	1	1	1	0	0	1	1	1	0	0	0	0	0
10	1	1	1	0	0	1	1	1	0	0	1	1	1
25	1	1	1	0	0	1	1	1	0	0	1	1	1

The dimensions are $(M+1) \times (N+1)$, where M is the number of coins and N is the target change.
Each row corresponds to adding one of the coins to the set we're considering, and each column corresponds to a change value.
The actual values are just booleans representing whether or not you can make exact change for that setup.
The grid is filled out row-by-row. You start with 1 in the top left, because you can always make exact change for 0. Then you insert 1's into the current row at every index k for which either index k or $k-c$ where c is the value of your current row's coin. After repeating this for every row, you check whether the column corresponding to your desired value is true or false.
Essentially, each row the union of the previous row and the previous row shifted right by the coin value, with each of those two corresponding to taking or not taking the current row's coin.
This works because, at every step, you are copying the change numbers that worked before, and adding in the numbers where adding the new coin is neccessary.

The implementation is in the `Lab06Functions.py` file, but here it is for reference (with comments and stuff removed for brevity):

```
def exactChange(target, coins):
        row = 1<<target
        for coin in coins: row |= row >> coin
        return row&1
```

The algorithm's runtime is $O(n)$, where n is the length of `coins`, because bitshift and bitwise operations are generally close enough to constant time (shifting is only constant time if your computer has a barrel shifter, which is common). Anyway, in this case, the numbers are limited to having at most `target` bits, so this algorithm is $O(n)$ in the same way radix sort is.

This is much much better than the $O(2^n)$ naive recursive implementation.