Intro to DP. The Fibonacci sequence $\{f_i\}_{i=0,1,\ldots}$ is defined as:

$$f_i = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ f_{i-1} + f_{i-2} & i > 1 \end{cases}$$

Give an efficient algorithm `fib` taking in an integer input i such that `fib`$(i) = f_i$.

Solution.

1. **(Top Down Solution).** There are three or two parts to a top down solution:

   (a) **Recursive spec**. Write what this algorithm takes as input and what it outputs. It should be written with enough specificity that it looks like a proposition you would prove correct by induction. In this case, the recursive spec is basically already given to you:

   `fib`(n) : Given $n \in \mathbb{N}$, outputs f_n, the n'th fibonacci number defined above.

   (b) **The Algorithm.** Implement the recursive spec. Importantly, mention that you *save* (or 'cache') previously computed `fib` values. This is how you prevent your algorithm from doing exponential, repeated work.

   `fib`(n)

   i. If $n = 0$: output 0. If $n = 1$: output 1.

   ii. Let `add1` $\leftarrow$ `fib`$(n-1)$ and `add2` $\leftarrow$ `fib`$(n-2)$. //compute these values if not already cached

   iii. Cache and output `add1` + `add2`.

   (c) **The Driver Function (Sometimes).** Often times, your algorithm will need additional arguments. If this is the case, then you need to call your DP algorithm as a helper function. I will provide a simple example at the end of this worksheet.

   For the analysis, you prove correctness using induction on $n \in \mathbb{N}$. For runtime, an upperbound on the running time is

   $$\text{runtime} \leq (\#\text{ subproblems}) \times (\text{work per subproblem}) = n \times O(1) = O(n).$$

   Outside of some other tricks, this is most often how you analyze DP algorithm runtimes.

2. **(Bottom Up Solution).** You can think of this as the 'non-recursive' approach, where you build up the values of `fib` from 0 to 1 to ... to n. As before, you'd want to write the recursive spec and its implementation:

   `fib`(n) : Given $n \in \mathbb{N}$, outputs f_n, the n'th fibonacci number defined above.

`fib`(n)

(a) Initialize array $f[0, \ldots, n]$ to be all $0's$.

(b) Set $f[0] = 0$ and $f[1] = 1$.

(c) For $i = 2, \ldots, n$: set $f[i] = f[i-1] + f[i-2]$.

(d) output $f[n]$.

Observe that the 'cache'ing' is done explicitly here. The runtime is $O(n)$ since we have n iterations and do $O(1)$ work per iteration. Correctness follows by a similar induction on n as in the top-down solution, except you reason about the array f and its prior computed $f[i-1]$ and $f[i-2]$ values.

Tips Appreciated. You are a third year Purdue CS PhD grad student trying to buy a \$3.81 black coffee at Mochaland. Your entire life savings is in your left pocket in the form of a few pennies (1 cent), dimes (10 cent), and nickels (5 cent) coins.

One of your remaining joys in life is putting your hand in your pocket and fidgeting with the weight of the coins – you'd like to imagine that having more coins means that you are doing better financially. Hence, you would like to purchase this black coffee with as few coins as possible. More generally, assume you have an infinite number of each type of coin and suppose a coffee costs n cents. Give an efficient algorithm `change` taking in coffee cost n as input and outputs the minimum number of coins needed to reach cost *exactly* n.[1]

Solution. Justin says: Here is another way to write a top down solution, in a less pseudocode, more mathematical formuation.

Recursive Spec:

$$\texttt{change}(n) : \text{Given } n \in \mathbb{N}, \text{ outputs the minimum number of pennies, dimes, and nickels needed to reach exactly cost } n.$$

We implement `change` as:

$$\begin{aligned} \texttt{change}(n < 0) &= \infty \\ \texttt{change}(0) &= 1 \\ \texttt{change}(n \geq 0) &= 1 + \min(\underbrace{\texttt{change}(n-1)}_{\text{penny}}, \underbrace{\texttt{change}(n-5)}_{\text{nickel}}, \underbrace{\texttt{change}(n-10)}_{\text{dime}}) \end{aligned}$$

There are n subproblems and $O(1)$ work per subproblem (to determine the minimum and add 1), so the total work is $O(n)$. Correctness follows from an induction proof on n.

Followup hints: (1) A driver function is useful here. (2) Augment `change` to take additional arguments. For the more complicated $O(3n)$ solution, google 'dynamic programming monotone queue optimization.'

[1] Followups: (1) Counterintuitively, it may take less coins to buy a large cup of coffee instead of a small cup. If a large cup costs m cents, and a small cup costs n cents, determine if it takes less coins to reach the exact price of a small cup or a large cup; (2) Find the minimum number of coins needed to exactly buy a n cent cup of coffee when you have at most p pennies, k nickels, and d dimes for some integers $p, k, d \geq 0$ in $O(npkd)$ time. Can you do this in $O(3n)$ time?