How (not) to compute the Fibonacci numbers

$F_0 = F_1 = 1$
$F_n = F_{n-2} + F_{n-1}$ for $n \geq 2$

```python
def f(n):
    return 1 if n<2 else f(n-2) + f(n-1)
```

**QUESTION**
Why is this a daft implementation?

**Diagram:**
- Tree of function calls
- Dependency graph

We use $f(3)$ twice. We don't need to compute it twice.
How (not) to compute the Fibonacci numbers

\[ F_0 = F_1 = 1 \]
\[ F_n = F_{n-2} + F_{n-1} \text{ for } n \geq 2 \]

We can get \( \Theta(n) \) running time by leveraging duplication in the dependency graph.

```
def f(n):
    x = np.ones(n+1)
    for i in range(2,n+1):
        x[i] = x[i-2] + x[i-1]
    return x[n]
```

```
cache = {}
def f(n):
    if n in cache:
        return cache[n]
    else:
        res = 1 if n<2 else f(n-2)+f(n-1)
        cache[n] = res
    return res
```

```
def f(n):
    x,y = 1,1
    for _ in range(2,n+1):
        x,y = y, x+y
    return y
```
The naive recursive solution to the Bellman equation is often impractical, since the computation tree typically grows exponentially with the size of the problem.

In many interesting problems, there is substantial overlap in the subproblems, permitting polynomial-time solution, using ...

- **top-down memo-ization**
  Simply implement the recursion, and cache the results

- **or bottom-up iteration**
  Start from the leaves and work up
  (but we first need to figure out the dependency graph)
Example: rod cutting

A DIY supplier has a steel rod of length $n \in \mathbb{N}$, and a machine that can cut it into smaller pieces. Different lengths can be sold for different prices; a piece of length $\ell \in \mathbb{N}$ fetches $p_\ell$.

How should it be cut, to maximize profit?

**Bellman equation:** Let $v(n)$ be the maximum profit achievable from a rod of length $n$. Then

$$v(n) = \begin{cases} 
0 & \text{if } n = 0 \\
\max_{1 \leq i \leq n} \{p_i + v(n - i)\} & \text{if } n > 0
\end{cases}$$

The dependency graph is:

- $v(0) \rightarrow v(1) \rightarrow v(2) \rightarrow \ldots \rightarrow v(n)$

**Bottom-up strategy:**
1. Create an array of size $n+1$
2. Set $v(0) = 0$
3. Fill in $v(1), v(2), \ldots, v(n)$ in order.

**Top-down memoization strategy:**
- Nothing special to say here; it's a totally generic approach.
Example 3.1.1 Matrix chain multiplication

The cost of multiplying two matrices depends on their dimensions:

\[
\begin{bmatrix}
\vdots & \vdots & \vdots \\
\ell \times m
\end{bmatrix} \times \begin{bmatrix}
\vdots & \vdots & \vdots \\
m \times n
\end{bmatrix} = \begin{bmatrix}
\vdots & \vdots & \vdots \\
\ell \times n
\end{bmatrix}
\]

\(\ell mn\) multiplications + \(\ell (m - 1)n\) additions

Let’s take the total cost to be \(\ell mn\).

If we want to compute the product of several matrices, we have a choice about the order of multiplication (because matrix multiplication is associative). For example,

\[
ABCDE = (AB)((CD)E) = A \left(B((CD)E)\right)
\]

Find the least-cost way to compute the product \(A_0 \cdot A_1 \cdot \cdots \cdot A_{n-1}\)

\(d_0 \times d_1 \quad d_1 \times d_2 \quad \cdots \quad d_{n-1} \times d_n\)

**Bellman equation:** Let \(v(i, j)\) be the minimum cost for multiplying \(A_i A_{i+1} \cdots A_{j-1}\), for \(i < j\). Then

\[
v(i, j) = \begin{cases} 
0 & \text{if } j = i + 1 \\
\min_{i < k < j} \{d_i d_k d_j + v(i, k) + v(k, j)\} & \text{if } j > i + 1 
\end{cases}
\]
**Bellman equation:** Let \( v(i, j) \) be the minimum cost for multiplying \( A_iA_{i+1}\ldots A_{j-1} \), for \( i < j \). Then

\[
v(i, j) = \begin{cases} 
0 & \text{if } j = i + 1 \\
\min_{i < k < j} \{d_id_kd_j + v(i, k) + v(k, j)\} & \text{if } j > i + 1
\end{cases}
\]

*Dependency graph:*

Bottom-up strategy:

1. Create a \( n \times n \) matrix.
2. Fill in 0 on the diagonal \((j = i+1)\).
3. Fill in the \( j = i+2 \) diagonal then \( j = i+3 \ldots \) until we fill in \( i = 0, j = n \).

\( v(G, j) \) depends on \( v(i, k) \) for \( k \) larger than \( i \) and \( v(k, j) \) for \( k \) smaller than \( j \).

(Note: Ignore the \( i > j \) part, (the part under the diagonal), since \( v(i, j) \) only makes sense for \( i < j \).)
Example 3.1.2  Longest common subsequence

A subsequence of a string $s$ is any string obtained by dropping zero or more characters from $s$. Given two strings $s$ and $t$, what’s the longest subsequence they have in common?

Bellman equation: Let $v_{i,j}$ be the length of the LCS between $s[0:i]$ and $t[0:j]$. Then

$$v_{i,j} = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
 v_{i-1,j} \lor v_{i,j-1} \lor (1 + v_{i-1,j-1}) & \text{if } i > 0 \text{ and } j > 0 \text{ and } s[i-1]=t[j-1] \\
 v_{i-1,j} \lor v_{i,j-1} & \text{if } i > 0 \text{ and } j > 0 \text{ and } s[i-1] \neq t[j-1]
\end{cases}$$
Bellman equation: Let $v_{i,j}$ be the length of the LCS between $s[0:i]$ and $t[0:j]$. Then

$$v_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ v_{i-1,j} \lor v_{i,j-1} \lor (1 + v_{i-1,j-1}) & \text{if } i > 0 \text{ and } j > 0 \text{ and } s[i-1]=t[j-1] \\ v_{i-1,j} \lor v_{i,j-1} & \text{if } i > 0 \text{ and } j > 0 \text{ and } s[i-1] \neq t[j-1] \end{cases}$$

Bottom-up strategy:
1. Create a $(\text{len}(s)+1) \times (\text{len}(t)+1)$ matrix
2. Fill in 0 on $i=0$ and on $j=0$
3. Fill in the rest, e.g. or any way that's consistent with the dependencies.
How to extract the programme

When we compute the maximum

$$v(x) = \max_a \{\text{reward}_{x,a} + v(\text{next}_{x,a})\}$$

let’s also store which $a$ achieved the maximum.

To find an optimal path, just start at the top and repeatedly pick the best action.

This works whether we’re computing the values bottom-up, or top-down with memo-ization.

**QUESTION**

What would you do if there are two equally-good actions?
Example 3.1.2  Longest common subsequence

We produce a table of $v_{i,j} = \text{length of longest common subsequence between } s[0:i] \text{ and } t[0:j]$

At the same time, we store the optimal action at each state $(i, j)$

To extract the match, start at the initial state $(i, j) = (\text{len}(s), \text{len}(t))$, then follow the optimal actions.

A longest common substring of ALGORITHM and LOGARITHM is LGRITHM
The art of dynamic programming is to formulate the problem so that we maximize overlap between subproblems.

**Example.** Find the least-cost way to compute the matrix product \(A_0A_1 \cdots A_{n-1}\)

Recall that matrix multiplication is associative: \(ABCDE = (AB)((CD)E) = A(B((CD)E))\)

Let’s think of the problem as “repeatedly, choose a pair of adjacent matrices to multiply”.

Let \(v(e)\) be the minimum cost of multiplying matrices with dimension-sequence \(e = [e_0, e_1, \ldots, e_n]\). Then

\[
  v(e) = \begin{cases} 
  0 & \text{if } n=1 \\
  \min_{0 \leq k < n} \left\{ d_{k-1}d_kd_{k+1} + v(e \text{ with } e_k \text{ dropped}) \right\} & \text{otherwise}
  \end{cases}
\]

This is yucky because the dependency graph has lots of nodes (one node for every possible \(e\) for a given list of matrices). For our other approach, #nodes is quadratic in \(n\).
Example 3.2.1 Resource allocation

Several different university societies have all requested to book the sports hall, request $k$ having start time $u_k \in \mathbb{R}$ and end time $v_k \in \mathbb{R}$. The hall can only fit one activity at a time. What is the maximum number of requests that can be satisfied without overlap?

**EXERCISE**
Find a different formulation, not based on sets, so that the subproblems overlap better.