13.2
Dynamic programming
Removing the recursion by filling the table in the right order

“Dynamic programming”

\[ \text{Fib}(n): \]
\[
\begin{align*}
\text{if } (n = 0) & \quad \text{return } 0 \\
\text{if } (n = 1) & \quad \text{return } 1 \\
\text{if } (M[n] \neq -1) & \quad \text{return } M[n] \\
M[n] & \leftarrow \text{Fib}(n - 1) + \text{Fib}(n - 2) \\
\text{return } M[n]
\end{align*}
\]

\[ \text{FibIter}(n): \]
\[
\begin{align*}
&\text{if } (n = 0) \quad \text{then} \\
&\text{return } 0 \\
&\text{if } (n = 1) \quad \text{then} \\
&\text{return } 1 \\
F[0] & = 0 \\
F[1] & = 1 \\
\text{for } i & = 2 \text{ to } n \text{ do} \\
\quad F[i] & = F[i - 1] + F[i - 2] \\
\text{return } F[n]
\end{align*}
\]
Dynamic programming: Saving space!

Saving space. Do we need an array of $n$ numbers? Not really.

\begin{algorithm}
\textbf{FibIter}(n):
\begin{algorithmic}
\State if ($n = 0$) then
\State \quad return 0
\State if ($n = 1$) then
\State \quad return 1
\State $F[0] = 0$
\State $F[1] = 1$
\For {i = 2 to $n$}
\State \quad $F[i] = F[i - 1] + F[i - 2]$
\EndFor
\State return $F[n]$
\end{algorithmic}
\end{algorithm}
Dynamic programming – quick review

Dynamic Programming is **smart recursion**
+ explicit memoization
+ filling the table in right order
+ removing recursion.
Dynamic programming – quick review

Dynamic Programming is **smart recursion**
+ **explicit memoization**
+ filling the table in right order
+ removing recursion.
Dynamic programming – quick review

Dynamic Programming is **smart recursion**
+ **explicit memoization**
+ filling the table in right order
+ removing recursion.
Analyzing memoized recursive function

**Question:** Suppose we have a recursive program $\text{foo}(x)$ that takes an input $x$.
- On input of size $n$ the number of **distinct** sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time **not counting** the time for its recursive calls.

Suppose we memoize the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

**Q:** What is an upper bound on the running time of **memoized** version of $\text{foo}(x)$ if $|x| = n$? $O(A(n)B(n))$. 

Analyzing memoized recursive function

**Question:** Suppose we have a recursive program \( \text{foo}(x) \) that takes an input \( x \).

- On input of size \( n \) the number of distinct sub-problems that \( \text{foo}(x) \) generates is at most \( A(n) \).
- \( \text{foo}(x) \) spends at most \( B(n) \) time not counting the time for its recursive calls.

Suppose we memoize the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes \( O(1) \) time.

**Q:** What is an upper bound on the running time of memoized version of \( \text{foo}(x) \) if \( |x| = n? \ O(A(n)B(n)). \)
Analyzing memoized recursive function

**Question:** Suppose we have a recursive program \( \text{foo}(x) \) that takes an input \( x \).

- On input of size \( n \) the number of distinct sub-problems that \( \text{foo}(x) \) generates is at most \( A(n) \)
- \( \text{foo}(x) \) spends at most \( B(n) \) time not counting the time for its recursive calls.

Suppose we memoize the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes \( O(1) \) time.

**Q:** What is an upper bound on the running time of memoized version of \( \text{foo}(x) \) if \( |x| = n \)? \( O(A(n)B(n)) \).
Analyzing memoized recursive function

**Question:** Suppose we have a recursive program $\text{foo}(x)$ that takes an input $x$.

- On input of size $n$ the number of distinct sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time not counting the time for its recursive calls.

Suppose we memoize the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

**Q:** What is an upper bound on the running time of memoized version of $\text{foo}(x)$ if $|x| = n$? $O(A(n)B(n))$. 
13.2.1
Fibonacci numbers are big – corrected running time analysis
Is the iterative algorithm a polynomial time algorithm? Does it take $O(n)$ time?

1. input is $n$ and hence input size is $\Theta(\log n)$
2. output is $F(n)$ and output size is $\Theta(n)$. Why?
3. Hence output size is exponential in input size so no polynomial time algorithm possible!
4. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?
THE END

... (for now)