13.2
Dynamic programming

Removing the recursion by filling the table in the right order

"Dynamic programming"

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (M[n] ≠ −1)
        return M[n]
    M[n] ⇐ Fib(n − 1) + Fib(n − 2)
    return M[n]
```

```
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    F[0] = 0
    F[1] = 1
    for i = 2 to n do
        F[i] = F[i − 1] + F[i − 2]
    return F[n]
```

Dynamic programming: Saving space!

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

```
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    F[0] = 0
    F[1] = 1
    for i = 2 to n do
        F[i] = F[i − 1] + F[i − 2]
    return F[n]
```

```
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    prev2 = 0
    prev1 = 1
    for i = 2 to n do
        temp = prev1 + prev2
        prev2 = prev1
        prev1 = temp

    return prev1
```

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 $foo(x)$ that takes an input x.

- On input of size n the number of distinct sub-problems that $foo(x)$ generates is at most $A(n)$
- $foo(x)$ 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 $foo(x)$ if $|x| = n$? $O(A(n)B(n))$.

Analyzing memoized recursive function

Question: Suppose we have a recursive program $foo(x)$ that takes an input x.

- On input of size n the number of distinct sub-problems that $foo(x)$ generates is at most $A(n)$
- $foo(x)$ 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 $foo(x)$ if $|x| = n$? $O(A(n)B(n))$.

Analyzing memoized recursive function

Question: Suppose we have a recursive program $foo(x)$ that takes an input x.

- On input of size n the number of distinct sub-problems that $foo(x)$ generates is at most $A(n)$
- $foo(x)$ 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 $foo(x)$ if $|x| = n$? $O(A(n)B(n))$.

Analyzing memoized recursive function

Question: Suppose we have a recursive program $foo(x)$ that takes an input x.

- On input of size n the number of distinct sub-problems that $foo(x)$ generates is at most $A(n)$
- $foo(x)$ 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 $foo(x)$ if $|x| = n$? $O(A(n)B(n))$.

13.2.1
Fibonacci numbers are big – corrected running time analysis

Back to Fibonacci Numbers

```
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    prev2 = 0
    prev1 = 1
    for i = 2 to n do
        temp = prev1 + prev2
        prev2 = prev1
        prev1 = temp

    return prev1
```

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)