# CS173 Lecture B, October 13, 2015

Tandy Warnow

October 13, 2015

CS 173, Lecture B
October 13, 2015
Tandy Warnow

# Dynamic Programming

Dynamic programming is an algorithmic design technique that can make it easy to solve problems efficiently.
Dynamic programming is similar to recursion – but it is bottom-up, instead of top-down.

# Fibonacci numbers

Consider how to calculate Fibonacci numbers, $F(n)$, defined by

- $F(1) = F(2) = 1$
- $F(n) = F(n-1) + F(n-2)$ if $n > 2$

Let's do this calculation recursively.
Input: $n \in Z^+$
Algorithm:

- If $[n = 1 \text{ or } n = 2]$ then Return(1)
- Else
  - Recursively compute $F(n-1)$ and store in $X$
  - Recursively compute $F(n-2)$ and store in $Y$
  - Return$(X + Y)$

# Running time of recursive algorithm for F(n)

The running time $t_1(n)$ of this algorithm satisfies:

- $t_1(1) = C$
- $t_1(2) = C$
- $t_1(n) = t_1(n-1) + t_1(n-2) + C'$

for some positive integers $C, C'$.

It's immediately obvious that $t_1(n) \geq F(n)$ for all $n \in Z^+$ (compare the recurrence relations).

This is a problem, because $F(n)$ grows exponentially (look at http://mathworld.wolfram.com/FibonacciNumber.html), and so $t_1(n)$ grows at least exponentially!

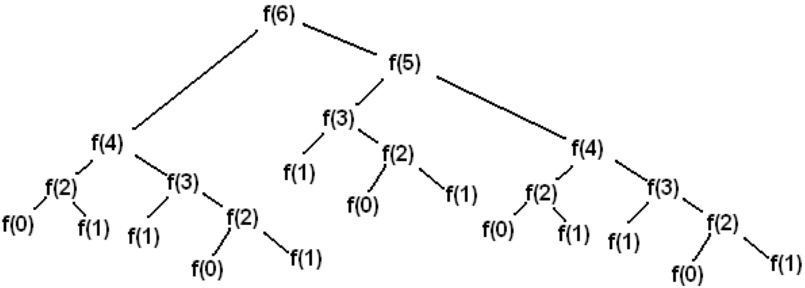
# Recursive computation of the Fibonacci numbers

When we compute the Fibonacci numbers recursively, we compute $F(n)$ by independently computing $F(n-1)$ and $F(n-2)$.

Note that $F(n-1)$ also requires that we compute $F(n-2)$, and so $F(n-2)$ is computed twice, rather than once and then re-used.

It would be much better if we had stored the computations for $F(i)$ for smaller values of $i$ (in $\{1, 2, \ldots, n-1\}$) so that they could be re-used.

# The recursion tree for the Fibonacci numbers



https://en.wikibooks.org/wiki/Algorithms/Dynamic_Programming

# A better way of computing $F(n)$

The simple recursive way of computing $F(n)$ is exponential, but there is a very simple **dynamic programming** approach that runs in linear time!

Input: $n \in Z^+$

```
If $n \leq 2$ return 1. Else:
    $F[1] := 1$
    $F[2] := 1$
    For $i = 3$ upto $n$, DO
        $F[i] := F[i-1] + F[i-2]$
    Endfor
    Return($F[n]$)
```

The running time $t_2(n)$ for this algorithm satisfies

- $t_2(1) \leq C'$
- $t_2(2) \leq C'$
- $t_2(n) = t_2(n-1) + C'$ if $n > 2$

for some positive constant $C'$. Note the difference in the recursive definition for $t_1(n)$ and $t_2(n)$.

# Bounding this recurrence relation

The running time $t_2(n)$ for this algorithm satisfies

- $t_2(1) \leq C_0$
- $t_2(2) \leq C_1$
- $t_2(n) \leq t_2(n-1) + C_2$

for some positive constants $C_0, C_1$, and $C_2$.
Let $C' = \max\{C_0, C_1, C_2\}$. It is easy to see that $C' > 0$.

We will prove that $t_2(n) \leq C'n$ for all $n \geq 1$, by induction on $n$.

# Running time analysis of the DP algorithm for $F(n)$

Note that

- $t_2(n) \leq C'$ if $n \leq 2$ and
- $t_2(n) \leq t_2(n-1) + C'$ for $n > 2$

We prove by induction that $t_2(n) \leq C'n$ for all $n \in Z^+$.
**Proof:** The base case is $n = 1, 2$, and is verified.
The inductive hypothesis is that there is some $K \in Z^+$ with $K \geq 2$, such that for all $n \in \{1, 2, \ldots, K\}$, $t_2(n) \leq C'n$.
We want to show that $t_2(K+1) \leq C'(K+1)$.
Since $K \geq 2$, $K + 1 > 2$. Hence, $t_2(K+1) = t_2(K) + C'$.
By the inductive hypothesis, $t_2(K) \leq C'K$. Hence,

$$t_2(K+1) \leq t_2(K) + C'$$
$$\leq C'K + C' = C'(K+1)$$

which is what we wanted to prove.
Since $K$ was arbitrary, it follows that $t_2(n) \leq C'n$ for all $n \in Z^+$.

# Finding a longest increasing subsequence

Input: string $X = x_1 x_2 \ldots x_n$ over alphabet $\Sigma$
Output: increasing subsequence of $X$ that is as long as possible.
Note, we require that the subsequence be strictly increasing!

Example: $X = 1, 3, 1, 8, 2, 4, 9, 2, 10, 3$
What are some increasing subsequences?
$1, 3, 8, 9, 10$ is an increasing subsequence. Is it the longest one?
Is $1, 1$ an increasing subsequence? (Answer: NO!)

# Dynamic programming algorithm for LIS

Let $n[i]$ be the length of the longest increasing subsequence of the first $i$ letters of $X$, and that includes $x_i$.
For example, when $X = 1, 3, 1, 8, 2, 4, 9, 2, 10, 3$, then

- $n[1] = 1$
- $n[2] = 2$
- $n[3] = 1$
- $n[4] = 3$
- $n[5] = 2$

After we compute $n[i]$ for $i = 1, 2, \ldots, n$, then we set $LIS = \max\{n[i] : i = 1, 2, \ldots, n\}$.
How can we compute $n[i]$ efficiently?

# Dynamic programming algorithm for LIS

To compute $n[i], i = 1, 2, \ldots, n$, we could use recursion or dynamic programming. Let's do it using dynamic programming.

- For $i = 1$ upto $n$, DO
    - Compute $n[i]$ somehow

  Endfor

Can we use the values for $n[1], n[2], \ldots, n[i-1]$ to compute $n[i]$?

# Computing $n[i]$

Can we use the values for $n[1], n[2], \ldots, n[i-1]$ to compute $n[i]$?
Let's define $\mathit{Smaller}[i] = \{j \in \{1, 2, \ldots, i-1\} : x_i > x_j\}$.

- Case: $\mathit{Smaller}[i] = \emptyset$. Then $n[i] = 1$, because the only increasing subsequence of $x_1 x_2 \ldots x_i$ that includes $x_i$ is $x_i$ itself.
- Case: $\mathit{Smaller}[i] \neq \emptyset$. Then, for every $j \in \mathit{Smaller}[i]$, there is an increasing subsequence of $x_1 x_2 \ldots x_i$ of length $n[j] + 1$. Hence, $n[i] = \max\{n[j] + 1 : j \in \mathit{Smaller}[i]\}$.

# DP algorithm

```
▶ For $i = 1$ upto $n$, DO
    ▸ Comment: Compute $Smaller[i]$
      If $Smaller[i] = \emptyset$, then $n[i] := 1$
            else $n[i] := \max\{n[j] + 1 : j \in Smaller[i]\}$
  Endfor
```

Running time analysis:

- Computing each $Smaller[i] : i = 1, 2, \ldots, n$ takes $O(n)$ time, and so computing them all takes $O(n^2)$ time.
- Computing each $n[i]$ after all the previous $n[j]$ are computed takes $O(i)$ time. Since $i \leq n$, this is $O(n)$ for each $i$. Hence, these calculations take $O(n^2)$ overall.

Altogether, the running time is $O(n^2)$ time.

# Two-person games

Remember the original two person game?

- There are $n$ rocks on pile 1, and $m$ rocks on pile 2.
- Each player can take one rock off of one pile, or one rock off of each pile.
- The person to take the last rock off wins.

Let's write a dynamic programming algorithm to determine who has a winning strategy.

# DP algorithm

Input: $n$ and $m$.
Output: TRUE if player 1 has a winning strategy for starting condition (i,j), and FALSE otherwise.

Approach: Let $M[i,j]$ be a matrix with values **T** and **F**, where $M[i,j] = T$ iff player 1 has a winning strategy when the starting condition has $i$ rocks on pile 1 and $j$ rocks on pile 2.
We let $i = 0, 1, \ldots, n$ and $j = 0, 1, \ldots, m$.
Boundary conditions:

- $M[0,0] = F$ (player 1 does not have a winning strategy, and no one wins; however, we don't even need to set this)
- $M[1,1] = M[1,0] = M[0,1] = T$ (player 1 does have a winning strategy)

# DP algorithm for Two-person game

Player 1 has a winning strategy if he/she can create a starting condition where player 2 (who will then be the "new" player) does not have a winning strategy.
Remember that the starting position $(i, j)$ always has at least one rock on at least one pile; hence at least one of $i, j$ is positive.
There are three cases:

- $i = 0$: the only legal move from $(i, j)$ is to $(i, j - 1)$
- $j = 0$: the only legal move from $(i, j)$ is to $(i - 1, j)$
- $i > 0, j > 0$: there are three legal moves from $(i, j)$ – to $(i - 1, j), (i, j - 1)$, and $(i - 1, j - 1)$.

# DP algorithm, both $i, j$ positive

On starting position (i,j), player 1 can move to each of the following "new positions":

- $(i-1, j-1)$ by taking one rock off of each pile
- $(i, j-1)$ by taking one rock off of the first pile
- $(i-1, j)$ by taking one rock off of the second pile

There are two possibilities:

- $M[x,y] = T$ for all new positions $[x,y]$. What this means is that the new player 1 has a winning strategy on the new position. Hence, the original player 1 *cannot win*. In this case, we set $M[i,j] = F$.
- $M[x,y] = F$ for at least one new position $[x,y]$. In this case, for that specific new position, player 1 has a winning strategy – as long as he/she moves to $[x,y]$. We set $M[i,j] = T$.

# The DP algorithm: Phase 1

Input: non-negative integers $m$ and $n$, with at least one of them positive

Phase 1: Compute top row and left-most column:

```
$M[1,0] = M[1,1] = M[0,1] = T$
For $i = 1$ up to $m$ DO:
    $M[i,0] = \neg M[i-1,0]$
Endfor
For $j = 1$ up to $n$ DO
    $M[0,j] = \neg M[0,j-1]$
Endfor
```

# DP algorithm, Phase 2

Comment: Now we have the top row and leftmost column filled in.

```
For $i = 1$ up to $m$ DO
    For $j = 1$ up to $n$ DO
        $M[i,j] = \neg M[i-1,j] \vee \neg M[i,j-1] \vee \neg M[i-1,j-1]$
    Endfor
Endfor
```

# Generalizing this observation

Take an arbitrary two-person game where people take rocks off of $k$ piles, under some rules, and the last person to remove a rock wins.

Let $M[i, j]$ be TRUE if and only if the first player has a winning strategy.

How would you use DP to solve two-person games?

# Class Exercise

Consider the two-person game as follows:

- You begin with $k$ rocks on one pile and $m$ rocks on the other pile, and the players take turns taking rocks off the piles.
- The player to remove the last rock wins.
- At each turn, each player can do one of the following:
  - Take all the rocks off of one pile if it has between 1 and 3 rocks, and not remove rocks from the other pile
  - Take a total of one to three rocks off (so 3 rocks from one pile, 2 from one pile, 1 rock from one pile, 2 rocks from one pile and 1 rock from the other, or 1 rock off from each pile)

Questions:

1. Who wins when you start with $k = 0, m = 2$?
2. Who wins when you start with $k = 0, m = 3$?
3. Who wins when you start with $k = 1, m = 2$?
4. Who wins when you start with $k = 1, m = 3$?
5. Who wins when you start with $k = 0, m = 4$?

Write the DP algorithm to figure out who wins on input $k, n$.

# DP vs. Recursive Algorithms

In these examples, the DP approach has been more efficient than recursion. But this is not always the case!

Sometimes the recursive approach is faster.

It depends on whether you really need to compute *all* the subproblems. If you do, then DP is at least as efficient, and often faster.

# Writing DP algorithms

Please observe the following guidelines for writing a dynamic programming algorithm:

- Explain your variables using English, showing what they are supposed to mean
- Show how to compute the values for the boundary conditions
- Specify the order in which you compute the values
- Show how to compute each value based on the earlier computations
- Show where the final answer is stored

# Summary

- Dynamic programming and recursive algorithms are two ways of dealing with algorithm design.
- One is top down (recursion) and the other is bottom-up (dynamic programming).
- You can **prove** your algorithm is correct using induction, when the algorithm uses recursion or dynamic programming.

In both cases, you identify subproblems and show how solving subproblems lets you solve big problems.