CS173
Dynamic Programming

Tandy Warnow
Today’s material

- Dynamic Programming (DP) essentials (review)
- DP algorithm for computing the Fibonacci numbers (review)
- DP algorithm for finding a longest increasing subsequence
- DP algorithms for two-person games
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.
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:** $n \in \mathbb{Z}^+$

**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 $n \in \mathbb{Z}^+$ (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!
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: $n \in \mathbb{Z}^+$

- 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($F[n]$)

Furthermore, this way of computing the Fibonacci numbers has linear running time!
The difference is whether it is top-down (i.e., recursive) or bottom-up (i.e., dynamic programming).

For the example of Fibonacci numbers, the difference in running times is huge: exponential for the recursive algorithm and linear for the dynamic programming algorithm.

But it’s better to use recursion, so you need to check.
Finding a Longest Increasing Subsequence (LIS)

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, the subsequence must 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 $Smaller[i] = \{ j \in \{ 1, 2, \ldots, i-1 \} : x_i > x_j \}$.

- **Case:** $Smaller[i] = \emptyset$. Then $n[i] = 1$, because the only increasing subsequence of $x_1x_2\ldots x_i$ that includes $x_i$ is $x_i$ itself.

- **Case:** $Smaller[i] \neq \emptyset$. Then, for every $j \in Smaller[i]$, there is an increasing subsequence of $x_1x_2\ldots x_i$ of length $n[j] + 1$. Hence, $n[i] = \max\{ n[j] + 1 : j \in Smaller[i] \}$.
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.
Class Exercise

Apply the dynamic programming algorithm for Longest Increasing Subsequence to the following input:

8, 2, 3, −3, 1, 5, 2, 0, 3, 4, 6
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.

Class assignment: in pairs, play the game with 5 rocks on pile 1 and 4 rocks on pile 2.
Hint: the second player has a winning strategy!
Coming up with a Dynamic Programming Algorithm

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

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