Dynamic Programming
Reminder Overview
Algorithmic Paradigms

• **Greed.** Build up a solution incrementally, optimizing some local criterion.

• **Divide-and-conquer.** Break up a problem into two sub-problems, solve each sub-problem independently, and combine solutions of sub-problems to form a solution to the original problem.

• **Dynamic programming.** Break up a problem into a series of **overlapping** sub-problems, and build up solutions to larger and larger sub-problems.
Fibonacci Numbers

• $F_n = F_{n-1} + F_{n-2}$
• $F_0 = 0$, $F_1 = 1$
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

• Recursive procedure is straightforward but slow!

```java
static int F(int n) {
    if (n <= 1) return n;
    else return F(n-1) + F(n-2);
}
```

• Why? How slow?
• Let’s run a few examples: $n=10, 20, 30, 35, …$
• Now let’s draw the recursion tree
Fibonacci Numbers

\[ F(n) = F(n-1) + F(n-2) \ ; \ F(0) = 0, \ F(1) = 1 \]

\[ F(6) = 8 \]

- We keep calculating the same value over and over!
Fibonacci Numbers

- Run time: $T(n) = T(n-1) + T(n-2)$
- This function grows exponentially as $n$ grows
- The run time doubles as $n$ grows and is order $O(2^n)$.
- The recursive algorithm is inefficient as there are numerous repetitive calculations.
Fibonacci Numbers

• We can calculate $F_n$ in linear time by remembering solutions to the solved subproblems – dynamic programming

• Dynamic programming calculates from bottom to top.

• Values are stored for later use.

• This reduces repetitive calculation.

• Trade space for time!
Bottom-up computation

We can calculate $F(n)$ in linear time by storing small values.

\[
\begin{align*}
F[0] &= 0 \\
F[1] &= 1 \\
\text{for } i = 2 \text{ to } n \\
F[i] &= F[i-1] + F[i-2] \\
\text{return } F[n]
\end{align*}
\]

*Moral*: We can sometimes trade space for time.

*Do we really need $O(n)$ space?*
Bottom-up computation

We can calculate $F(n)$ in linear time by storing small values.

$F[0] = 0$
$F[1] = 1$

for $i = 2$ to $n$

$F[i] = F[i-1] + F[i-2]$

return $F[n]$

Can $F(n)$ be computed in less than $O(n)$ time?
Actually…

- We can compute Fibonacci numbers in logarithmic time:

\[
\begin{pmatrix}
F_k \\
F_{k+1}
\end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{k-1} \\
F_k
\end{pmatrix}
\]

\[
\begin{pmatrix}
F_n \\
F_{n+1}
\end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F_0 \\
F_1
\end{pmatrix}
\]
Dynamic Programming

• Dynamic Programming is an algorithm design technique, often used for optimization problems: often minimizing or maximizing.

• **Optimal Sub-Structure property:** Optimal solution to the problem can be obtained based on optimal solutions to sub-problems.

• Like divide and conquer, DP solves problems by combining solutions to subproblems.

• Unlike divide and conquer, subproblems are not independent.
  – Subproblems may share subsubproblems,
  – However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)

• DP reduces computation by
  – Solving subproblems in a bottom-up fashion.
  – Storing solution to a subproblem the first time it is solved (memoization).
  – Looking up the solution when subproblem is encountered again.

• Key: determine structure of optimal solutions
Steps in Dynamic Programming

2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching (memoization) or bottom-up in a table.
4. Construct an optimal solution from computed values.
Dynamic Programming History

• Bellman (RAND Inst). Pioneered the systematic study of dynamic programming in the 1950s.

• Etymology.
  – Dynamic programming = planning over time.
  – Secretary of Defense was hostile to mathematical research.
  – Bellman sought an impressive name to avoid confrontation.
    • "it's impossible to use dynamic in a pejorative sense"
    • "something not even a Congressman could object to"