RETURNING TO RATIONAL KNAPSACK

- What if we cannot assume distinctness for profit/weight ratios in the proof?
- There is no longer a unique optimal solution
- So cannot prove optimal Y must be identical to greedy X
- Swapping might not improve the solution

OPTIMALITY PROOF WITHOUT DISTINCTNESS

There may be many optimal solutions
Key idea: Let Y be an optimal solution that matches X on a maximal number of indices
Observe: if X is really optimal, then Y = X
  - Suppose not for contra
  - We will modify Y to make it match X on one more index (a contradiction!)

As before, let j be the first index where X and Y differ
FEASIBILITY OF \( Y' \)

Showing \( y'_j \geq 0 \)
- By definition, \( y'_j = \gamma_j - \delta \) if \( \delta \leq \gamma_j \gamma_k \)
- But \( \delta \) is the minimum of \( \gamma_j (y_j - x_j) \leq \gamma_j \) and another expr.
- So \( \delta \leq \gamma_j \gamma_k \)

Showing \( y'_j \leq 1 \)
- \( y'_j = y_j + \frac{\delta}{\gamma_j} \leq 1 \) if \( \frac{\delta}{\gamma_j} \leq 1 - y_j \) \( \frac{\delta}{\gamma_j} \leq w_j (1 - y_j) \) (rearranging)
- \( \delta \leq w_j (y_j - y_k) \) (definition of \( \delta \))
- \( \delta \leq w_j (y_j - y_k) \leq w_j (1 - y_j) \) (by feasibility of \( X \))

PROFIT OF \( Y' \)

\[ \text{profit}(Y') = \text{profit}(Y) + \frac{\delta}{\gamma_j} p_j - \frac{\gamma_j}{\gamma_k} p_k = \text{profit}(Y) + \delta \left( \frac{p_j}{\gamma_j} - \frac{p_k}{\gamma_k} \right) \]

Since \( j \) is before \( k \), and we consider items with more profit per unit weight first, we have \( \frac{p_j}{\gamma_j} \geq \frac{p_k}{\gamma_k} \).
- Since \( \delta \geq 0 \) and \( \frac{p_j}{\gamma_j} \geq \frac{p_k}{\gamma_k} \), we have \( \delta \left( \frac{p_j}{\gamma_j} - \frac{p_k}{\gamma_k} \right) \geq 0 \)

Since \( Y \) is optimal, this cannot be positive
- So \( Y' \) is a new optimal solution that matches \( X \) on one more index than \( Y \)

Contradiction: \( Y \) matched \( X \) on a maximal number of indices.

Richard Bellman, the inventor of dynamic programming in 1950, related the following in his autobiography:
“What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, ‘programming.’ I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let’s kill two birds with one stone. Let’s take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is its impossibly to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning, its impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.”
Computing Fibonacci Numbers Inefficiently: A Toy Example to Compare D&C to Dynamic Programming

**Runtime**

- In unit cost model (UNREALISTIC!)
  - \( T(n) = T(n-1) + T(n-2) + O(1) \)
  - \( T(n) \geq 2T(n-2) + O(1) \)
  - \( T(n) \leq 2T(n-1) + O(1) \)

- \( n/2 \) levels of recursion for the first expression
- \( n \) levels for the second expression
- Work doubles at each level
- \( T(n) \) is certainly \( \Omega(2^n) \) and \( O(2^n) \)

**What Is the Input Size?**

Input: \( n \)
- Bits to store \( n \)? \( \log n \)
- For simplicity say input size is \( S = \log n \)
- \( 2^S = 2^{\log n} = n \)
- \( 2^S \) is doubly exponential in the input size \( S \)

Therefore, \( T(n) \in \Omega(2^{n/2}) \) \( \Rightarrow T(n) \in \Omega(2^{S/2}) \)

Recall \( T(n) \in O(2^n) = O(2^{2^S}) \)

In most of our analyses before this point, \( n \) has coincidentally been the size of the input.

**Why Is This So Slow?**

Subproblems have LOTS of overlap!
- Every subtree on the right appears on the left
- … recursively …
- Each subtree is computed exponentially often in its depth

This overlap suggests dynamic programming may be able to help!

Designing Dynamic Programming Algorithms for Optimization Problems

**(Optimal) Recursive Structure**

Examine the structure of an optimal solution to a problem instance \( I \), and determine if an optimal solution for \( I \) can be expressed in terms of optimal solutions to certain subproblems of \( I \).

**Define Subproblems**

- Define a set of subproblems \( S(I) \) of the instance \( I \), the solution of which enables the optimal solution of \( I \) to be computed. \( I \) will be the last or largest instance in the set \( S(I) \).

Or, if it's not an optimization problem, simply "determine if a solution for \( I \) can be expressed in terms of solutions to certain subproblems of \( I \)."

"... which enables the solution of \( I \) to be computed"

"Optimal" Recursive Structure

Designing Dynamic Programming Algorithms (cont.)

**Recurrence Relation**

Derive a recurrence relation on the optimal solutions to the instances in \( S(I) \). This recurrence relation should be completely specified in terms of optimal solutions to (smaller) instances in \( S(I) \) and/or base cases.

**Compute Optimal Solutions**

Compute the optimal solutions to all the instances in \( S(I) \). Compute these solutions using the recurrence relation in a bottom-up fashion, filling in a table of values containing these optimal solutions. Whenever a particular table entry is filled in using the recurrence relation, the optimal solutions of relevant subproblems can be looked up in the table (they have been computed already). The final table entry is the solution to \( I \).
SOLVING FIB USING DYNAMIC PROGRAMMING

- (Optimal) Recursive Structure
  - Solution to \( n \)-th Fibonacci number \( f(n) \) can be expressed as the addition of smaller Fibonacci numbers
  - No notion of optimality for this particular problem
- Define Subproblems
  - The set of subproblems that will be combined to obtain \( f(n) \) is \( S(n) = \{ f(0), f(1), ..., f(n) \} \)
- Recurrence Relation
  \[
  f(n) = f(n-1) + f(n-2) \quad (n \geq 2) \\
  f(1) = 1 \quad (n = 1) \\
  f(0) = 0 \quad (n = 0)
  \]
- Computing (Optimal) Solutions
  - Create table \( f[1..n] \) and compute its entries "bottom-up"

FILLING THE TABLE "BOTTOM-UP"

- Key idea:
  - When computing a table entry, must have already computed the entries it depends on!
- Dependencies
  - Extract directly from recurrence
  - Entry \( n \) depends on \( n-1 \) and \( n-2 \)
  - Computing entries in order 1..n guarantees \( n-1 \) and \( n-2 \) are already computed when we compute \( n \)

DP SOLUTION

- Space saving optimization:
  - We never look at \( f[i-3] \) or earlier
  - Can make do with a few variables instead of a table

CORRECTNESS

- Step 1
  - Order \( 0, n \) means \( i-1 \) and \( i-2 \) are already computed
  - Prove that when computing a table entry, dependent entries are already computed

  \[
  \text{Suppose } f[i-1] \text{ and } f[i-2] \text{ are the } (i-1)\text{-th and } (i-2)\text{-th Fibonacci numbers. Then } f[i] = f[i-1] + f[i-2] \]

  \[
  \text{This is still considered to be dynamic programming...} \\
  \text{We've just optimized out the table.}
  \]

MODEL OF COMPUTATION FOR RUNTIME

- Unit cost model is not realistic for this problem, because Fibonacci numbers grow quickly
  - \( F[0] = 0 \)
  - \( F[1] = 1 \)
  - \( F[100] = 354224848179261915075 \)
  - \( F[300] = 22223224462942044552979893461909967206656939649764990979600 \)
  - \( F[1000] \) is more than 200 digits

- Value of \( f[n] \) is exponential in \( n \)
- So number of digits of \( f[n] \) is linear in \( n \)
- Big numbers suggest using bit-complexity model
RUNNING TIME

BIT-COMPLEXITY MODEL

- $f[2], f[1]$ have $\Theta(i)$ digits
- So $f[1]+f[2]$ takes $\Theta(i)$ time
- $T(n) \in \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$
- Is this quadratic runtime?
  - NO! This is “quadratic in n”
  - When we say “quadratic runtime” we mean “quadratic in the input size.”
  - What’s the input size $S$?
    - The input is the number $n$, so $S = \log n$ bits

TIPS FOR ANALYSIS OF DP ALGORITHMS

- Think carefully about which model of computation (unit cost / bit complexity) is appropriate
  - If you can’t decide which is appropriate, you can try both and see if it changes the answer
- Think carefully about the input size $S$
  - Try to express runtime in terms of $S$
  - If that’s too hard, try to find an elegant/natural expression (see future lectures)
  - An algorithm is “linear time” only if it’s “linear in $S$”

OTHER MISCELLANEOUS TIPS

Building a table of results bottom-up is what makes an algorithm DP
- There is a similar concept called memoization
  - But, for the purposes of this course, we want to see bottom-up table filling!
- Base cases are critical
  - They often completely determine the answer
  - Try setting $f[1]=0$ in FibDP...

DYNAMIC PROGRAMMING APPROACH

- High level idea (can just think recursively to start)
  - Given a rod of length $n$
  - Either make no cuts, or make a cut and recurse on the remaining parts
    - Income $p_i$
    - Income[Left] + Income[Right]
  - Where should we cut?

ROD CUTTING

A “REAL” DYNAMIC PROGRAMMING EXAMPLE

Input:
- $n$: length of rod
- $p_1, ..., p_n$: price of a rod of length $i$

Output:
- Max income possible by cutting the rod of length $n$ into any number of integer pieces (maybe no cuts)

Example output: 10
**Recurrence Relation**

- Define $M(k)$ = maximum income for rod of length $k$
  - If we do not cut the rod, max income is $p_k$
  - If we do cut a rod at $i$
    - max income is $M(i) + M(k - i)$
  - Want to maximize this over all $i$
    - $M(k) = \max \{ p_k, \max_{1 \leq i \leq k - 1} (M(i) + M(k - i)) \}$

**Computing Solutions Bottom-Up**

- Recurrence: $M(k) = \max \{ p_k, \max_{1 \leq i \leq k - 1} (M(i) + M(k - i)) \}$
  - Compute table of solutions: $M[1..n]$
  
    | k | M[k] |
    |---|------|
    | 1 | 0    |
    | 2 | $p_2$ |
    | 3 | $p_3$ |
    | 4 | $p_4$ |
    | 5 | $p_5$ |
    | 6 | $p_6$ |
    | 7 | $p_7$ |
    | 8 | $p_8$ |
    | 9 | $p_9$ |
    | 10 | $p_{10}$ |
    | 11 | $p_{11}$ |
    | 12 | $p_{12}$ |

- Dependencies: entry $k$ depends on
  - $M[i] \rightarrow M[i+1..k-1]$  
  - $M[k-i] \rightarrow M[1..(k-1)]$
  - All of these dependencies are $< k$
  - So we can fill in the table entries in order $1..n$

**Input Size**

- Unit cost model is appropriate here
  - Each element of $p$ takes one word
    - $\Theta(1)$ bits each
  - So $\Theta(n)$ words
  - Input size $S \in \Theta(n)$

**Next Up...**

- DP 0-1 Knapsack and Coin Changing (for all currencies)
  - Tables will feature multiple dimensions
    - (Not just a 1D array)
    - Bottom-up filling orders become non-trivial
  - We often want to solve optimization problems
    - Arguing that an optimal solution is build from optimal sub-solutions becomes more significant
  - Input size calculations become more complex, and runtimes often include multiple variables

- Time complexity? $O(n)$?
  - Is this a quadratic time algorithm?

- So with runtime $\Theta(n^2) = \Theta(S^2)$, this is a quadratic time algorithm.