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.

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.
Greedy solution X

Optimal solution Y

\[ d = \min(w_j(x_j - y_j), w_k(y_k - x_k)) \]

Modified optimal solution Y'

\[ y'_j = y_j + \delta \]

\[ y'_k = y_k - \delta \]

In the case pictured here we have \( \delta = w_j(x_j - y_j) \), so we end up with \( y'_j = x_j \).

Feasibility of Y'

- Showing \( y'_j \geq 0 \)
  - By definition, \( y'_j = y_j + \frac{\delta}{w_j} \geq 0 \) if \( \delta \leq y_j w_k \)
  - But \( \delta \) is the minimum of \( w_j(x_j - y_j) \leq w_k y_k \) and another expr.
  - So \( \delta \leq y_j w_k \)

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

Profit of Y'

- \( \text{profit}(Y') = \text{profit}(Y') + \frac{\delta}{w_j} y_j = \text{profit}(Y') + \left( \frac{y_j}{w_j} - 1 \frac{w_k}{w_j} \right) \)

- Since \( j \) is before \( k \), and we consider items with more profit per unit weight first, we have \( \frac{y_j}{w_j} \geq \frac{y_k}{w_k} \).

- Since \( \delta > 0 \) and \( \frac{y_k}{w_k} \geq 1 \), we have \( \frac{\delta}{w_j} \leq \frac{w_k}{w_j} \geq 0 \)

- Since \( Y \) is optimal, \( \delta \) 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 else, what came, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, in some 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 multivariate, this was time-varying—I thought, less kill two birds with one stone. IExports 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 opposite in the word, dynamical, in a predicate sense. Think thinking of some convention that will possibly give it a precise meaning, its impossible. Thus, I thought dynamic programming was a good sense. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

"Bottom-up recursion" might also a reasonable name, as we'll see...
This is exponential in $n$. Is it exponential in the input size?

This overlap suggests dynamic programming may be able to help!

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."
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 subproblems that will be combined to obtain \( Fib(n) \)
  - \( S() = \{Fib(0), Fib(1), \ldots, Fib(n)\} \)
- Recurrence Relation
  - \( \begin{align*}
  f(0) &= 0 \\
  f(1) &= 1 \\
  f(n) &= f(n-1) + f(n-2) \quad n \geq 2
  \end{align*} \)
- Computing (Optimal) Solutions
  - Create table \( \{f[i] \ldots n \} \) and compute its entries "bottom-up"

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

MODEL OF COMPUTATION FOR RUNTIME

- Unit cost model is not realistic for this problem, because Fibonacci numbers grow quickly
  - \( F[0] = 0 \)
  - \( F[1] = 1 \)
  - \( F[100] = 354224848179261915075 \)
  - \( F[1000] = 222323442942045529739893461909967206 \)
  - \( F[10000] \) = more than 200 digits
  - \( F[100000] \) = more than 2200 digits

FILLING THE TABLE "BOTTOM-UP"

- Key idea:
  - When computing a table entry, we 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\ldots n guarantees \( n-1 \) and \( n-2 \) are already computed when we compute \( n \)

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
- **Step 2** (similar to D&C)
  - Suppose subproblems are solved correctly (optimally)
  - Prove these optimal subsolutions are combined into an optimal solution
  - Suppose \( f[i-1] \) and \( f[i-2] \) are the \( i-1 \)th and \( i-2 \)th Fib #s
  - \( f[i] = f[i-1] + f[i-2] \)
  - \( f[i] = \text{new array of size } n \)
  - \( f[0] = 0 \)
  - \( f[1] = 1 \)
  - \( f[i] = f[i-1] + f[i-2] \)
  - \( f[n] \) = the \( n \)th Fib #

How quickly does \( f_n \) grow? Let \( \phi = (1 + \sqrt{5})/2 \), then

\[
 f_n = \phi^n - (\phi^{-1})^n \left[ \frac{\phi}{\sqrt{5}} \right]. 
\]

Therefore \( f_n \in \Theta(\phi^n) \) and hence we also have \( f_{n+1} \in \Theta(\phi^{n+1}) \).

The value \( \phi \approx 1.6 \) is the golden ratio.

- 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[0]=1, f[1]=1 have Θ(1) digits
• So f[n]=f[n-1]+f[n-2] takes Θ(n) time
• Θ(n) ∈ Σ_n=1 Θ(1) = Θ(n²)
• 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 RbDP...

ROD CUTTING
A “NEAT” DYNAMIC PROGRAMMING EXAMPLE
• Input:
  • n: length of rod
  • p₁, ..., pₙ: 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)

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
  • Where should we cut?

DYNAMIC PROGRAMMING APPROACH
• Try all ways of making that cut
  • i.e., try a cut at positions 1, 2, ..., n-1
  • In each case, recurse on two rods [0, i] and [i, n]
• Take the max income over all possibilities (each i / no cut)
  • Max income we can get from the rod size i →
  • Optimal substructure: Max income from two rods wizes i and n-i

Example output: 10

Income p₁ Income(Left) + Income(Right)

Income p₂

Income p₃

Income pₙ

Income pₙ−₁

Income pₙ−₂

Income p₁
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 $i$
    - $max(M(i) + M(k-i))$ for $0 < i < k$
    - $M(k) = max(p_k, max_{1 \leq i \leq k-1}(M(i) + M(k-i)))$

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: $M[1..n]$
  - Dependencies: entry $k$ depends on
    - $M[i] \rightarrow M[1..(k-1)]$
    - $M[i-1] \rightarrow M[1..(k-1)]$
  - 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 $S \in \Theta(n)$

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

Recall $M(k)$ = maximum income for rod of length $k$

Recurrence: $M(k) = max(p_k, max_{1 \leq i \leq k-1}(M(i) + M(k-i)))$

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 $S \in \Theta(n)$

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

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