**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.

Let $\delta = \min(w_j - y_j, w_k - y_k)$.

Observe $\delta > 0$.

Weight to add: $w_j y_j - x_j y_j$.

Weight to remove: $w_k y_k - x_k y_k$.
FEASIBILITY OF Y’
- Showing y’ ≥ 0
  - By definition, y’ ≥ y + δ ≥ 0 if δ ≥ yw
  - But δ is the minimum of w(y - x) ≤ w(y - x) and another expr.
  - So δ ≤ yw
  - Showing y’ ≤ 1
  - y’ ≥ y + δ ≥ 0 if δ ≤ 1 = y
  - So δ ≤ w(y - x)
  - δ ≤ w(y - x)
  - (definition of δ)
  - (by feasibility of Y)

PROFIT OF Y’
- profit(Y’) = profit(Y) + δ ≤ y = profit(Y) + w(1 - y)
- Since j is before k, and we consider items with more profit per unit weight first, we have y > y’.
- Since δ > 0 and y > y’, we have δ ≥ 0
- 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!
This is exponential in $n$.

This $O(1)$ would change in the bit complexity model.

This is doubly exponential in the input size $S$!

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

Subproblems have lots of overlap!

Every subtree on the right appears on the left.

...recursively...

Each subtree is computed exponentially often in its depth.

The overlap suggests dynamic programming may be able to help!

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."
SOLVING FIB USING DYNAMIC PROGRAMMING

- (Optimal) Recursion Structure
- Solution to $n$-th Fibonacci number ($f_n$) can be expressed as the sum of smaller Fibonacci numbers
- No notion of optimality for this particular problem
- Define Subproblems
  - Set of subproblems that will be combined to obtain $Fib(n)$ is: $\{Fib(0), Fib(1), \ldots, Fib(n)\}$
  - Recurrence Relation: $f(n) = f(n-1) + f(n-2)$

- Computing (Optimal) Solutions
  - Create table $[f_1 \ldots n]$ 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 $1 \ldots n$ 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
  - 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

MODEL OF COMPUTATION FOR RUNTIME

- Unit cost model is not realistic for this problem, because Fibonacci numbers grow quickly
  - $F[0]=0$
  - $F[1]=1$
  - $F[10]=55$
  - $F[100]=354224848179261915075$
  - $F[300]=222332442942044529739893461909967206$
  - $F[1000]=$ more than 200 digits

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

$$f_n = \phi^n - (-\phi)^{-n} = \frac{\phi^n - 1}{\sqrt{5}}.$$

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

- **Examples**
  - *Income $p_i$, Income_left $f[i-1] + f[i-2] + ... + f[n-1]$*
  - *Income $p_i$, Income_right $f[i]$*
  - *Income $p_i$, Income_total $f[i-1] + f[i-2] + ... + f[n-1] + f[i]$*

**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)**

**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 RiDP...

**Rod Cutting**

A “next” 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:**

<table>
<thead>
<tr>
<th>Length</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Optimal substructure: Max income from two rods sizes $i$ and $n - i$...
**RECURSION RELATION**
- Define \( M(k) = \text{maximum income for rod of length } k \)
- If we do not cut the rod, max income is \( p_k \)
- If we cut the rod at \( i \),
  \[
  M(k) = \max\{ p_k, \max_{1 \leq i < k} (M(i) + M(k-i)) \}
  \]

**COMPUTING SOLUTIONS BOTTOM-UP**
- Recurrence: \( M(k) = \max\{ p_k, \max_{1 \leq i < k} (M(i) + M(k-i)) \} \)
- Compute table of solutions: \( M[1..n] \)
  
  \[
  \begin{array}{c|c|c|c}
  k & 1 & 2 & 3 \\
  \hline
  M & p_i & p_i + p_{i+1} & p_i + p_{i+1} \end{array}
  \]
- Dependencies: \( \text{entry } k \) depends on
  - \( M(i) \) \rightarrow \( M[i..(k-1)] \)
  - \( M(k-1) \) \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 \( S \in \Theta(n) \)

**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