Goal: Pick maximum size compatible set

Greedy Alg: 1) Pick T w/earliest finish time
            - Remove intervals incompatible with T
            2) Repeat

Claim: Alg picks optimal size set. Proof?

Let \( O \) be optimal size set, \( A \) be set alg chooses. Will show \( |O| = |A| \)

\[
A: \quad \begin{array}{c}
A_1 \\
A_2 \\
\vdots
\end{array}
\]

\[
O: \quad \begin{array}{c}
O_1 \\
O_2 \\
\vdots
\end{array}
\]

Note \( f(A_1) \leq f(O_1) \), because of how \( A \) is chosen.

What about \( f(A_2) \) vs \( f(O_2) \)?

\( O_2 \) is compatible with \( A_1 \), \( f(O_2) \geq f(A_2) \), otherwise greedy alg would have chosen \( O_2 \)

In general, suppose \( f(A_K) \leq f(O_K) \). Then

\[
\Rightarrow O_{K+1} \text{ compatible with } A_1, A_2, \ldots, A_K \Rightarrow f(O_{K+1}) \geq f(A_{K+1})
\]

By induction, \( A \) is "at least as good" as Optimal \( \checkmark \)

Dynamic Programming (review?)

When to use it? When answers depend on multiple subanswers, want to avoid asking same questions twice.

\[ f(n) = f(n-1) + f(n-2) \]

General Recipe for DP problems:

1. Express solution as recursive equation e.g. \( s(m,n) = \max(s(m-1,n), s(m,n-1)) \)
2. Determine base cases.
3. Evaluate bottom up

Hard Part
Warm-up: let's "solve" an NP-complete problem

**Subset-sum:** \( S = \{ s_1, s_2, \ldots, s_n \} \), target \( t \)

**Goal:** output \( \text{True} \) iff \( \exists I \subseteq [n] \) \( \sum_{i \in I} s_i = t \) "naive" \( 2^n \) solution

DP solves in time \( O(nt) \)

1. Express solution recursively. Let \( A(i, t) = \text{True} \) iff can form \( t \) using \( s_i \) to \( s_n \). (False otherwise)

\[
A(n, t) = A(n-1, t) \text{ OR } A(n-1, t - s_n)
\]

2. Base Case:
\[
A(0, 0) = \text{True} \quad A(0, e) = \text{False}
\]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Q: Why isn't this a poly-time alg for subset-sum?

A: \( t \) only takes \( \log t \) bits to represent, so alg is exponential in input size

...still, practical if \( t \ll 2^n \)

---

**Harder Problem**

**Matrix-Chain-Multiplication:**

\[
\begin{bmatrix}
100 & 3 \\
4 & 100
\end{bmatrix}
\begin{bmatrix}
3 & 100 \\
100 & 3
\end{bmatrix}
\]

Multiplication order matters:

\[
( A_1 \cdot A_2 ) \cdot A_3 = 3 \cdot 100 \cdot 100 + 100 \cdot 100 \cdot 100 = 4,000,000 \text{ ops}
\]

\[
A_1 \cdot ( A_2 \cdot A_3 ) = 300 + 300 = 600 \text{ ops}
\]

Big difference!!
Given: \( A_1, A_2, \ldots, A_n \)

Goal: what order to multiply them in? (minimize total ops)

Naively, \((n-1)!\) different orders (actually some are equivalent, but number of parentheses is still \(\Omega(2^n)\), see CLR)

DP can do better.

Trick is to express solution recursively, but how?

Multiplying \( A_1 \cdot A_n \) eventually comes down to multiplying two matrices:

\[ A_1 \cdot A_n = (A_1, A_2, \ldots, A_K) \cdot (A_{K+1}, \ldots, A_n) \]

for some choice of \( K \)

Let \( \text{opt}(i,j) = \text{optimal # of operations to multiply } A_i \cdot A_j \) (we want \( \text{opt}(1,n) \))

Then \( \text{opt}(i,j) = \min_{1 \leq K < n} \left( \text{opt}(i,K) + \text{opt}(K+1,j) + \text{cost}(K) \right) \)

"Base case": \( \text{opt}(i,i) = 0 \)

Filling the table:

```
  i   1   2   3   4   5   6   7   8   9
  1   0   0   0   0   0   0   0   0   0
  2   0   0   0   0   0   0   0   0   0
  3   0   0   0   0   0   0   0   0   0
  4   0   0   0   0   0   0   0   0   0
  5   0   0   0   0   0   0   0   0   0
  6   0   0   0   0   0   0   0   0   0
  7   0   0   0   0   0   0   0   0   0
  8   0   0   0   0   0   0   0   0   0
  9   0   0   0   0   0   0   0   0   0
```

Fill along diagonals

Running time: \( O(n^2) \) entries, each takes \( O(1) \) time

\( O(n^3) \) total

New topic: Sketch of an approximation alg

Non-identical Machine Interval Scheduling

Given: \( m, n \) machines, list of which jobs can be processed on which machines

Goal: maximize # of processed intervals

NP-hard, but \((1-\frac{1}{e})\) approximation alg exists

Sketch: Define variables \( x_{ij} \) to indicate whether interval \( i \) is scheduled on machine \( j \)

constrain \( x_{ij} \in \{0,1\} \)
Goal: maximize $\sum_i \sum_j x_{ij}$

subject to $\sum_j x_{ij} \leq 1 \ \forall i$ (each interval on at most one machine)

$\sum_{i \in \text{range}(t)} x_{ij} \leq 1 \ \forall \text{start times } t$ (no overlapping intervals)

$x_{ij} \in \{0,1\} \ \forall i,j$

This is an integer linear program

Solving ILPs is still NP-hard.

so Relax integer constraint, instead $0 \leq x_{ij} \leq 1$

This LP can be solved in poly time (will talk about this) later in class

Solution might involve fractional $x_{ij}$'s (e.g. $x_{ij} = 0.2$)

can sometimes round these to 0 or 1 intelligently, to get a sensible result.