Dynamic Programming

Intuition: Divide + Conquer +

Divide problems into subproblems,
solve subproblems,
combine subproblem solutions into full solution

- What if subproblems overlap?
- What if recursion too slow?
Today's Example: Weighted Interval Scheduling

Input:
- n requests \( \{1, 2, \ldots, n\} \)
- For each request \( i \):
  - start time \( s_i \)
  - finish time \( f_i \)
  - value \( v_i \)
- Assume sorted by finish time: \( f_1 \leq f_2 \leq \ldots \leq f_n \)

Goal: Find \( S \subseteq \{1, 2, \ldots, n\} \) s.t. no two intervals in \( S \) overlap, \( \max \sum_{i \in S} v_i \)

Ex:

(values not shown)

\[
\begin{aligned}
& p(1) = 0 \\
& p(2) = 0 \\
& p(3) = 0 \\
& p(4) = 1 \\
& p(5) = 0 \\
& p(6) = 2 \\
& p(7) = 3 \\
& p(8) = 5 \\
\end{aligned}
\]
Def: Let $p(i)$ be largest $j < i$ s.t. intervals $i, j$ are disjoint ($f_j \leq s_i$)

($i$ finishes before $j$ even starts)
start reasoning about optimal solution $S^*$

**Fact:** Either $n \in S^*$ or $n \not\in S^*$

If $n \not\in S^*$: $S^*$ is optimal sol for $\{1, \ldots, n-1\}$

If $n \in S^*$: nothing b/w $(p(n), n-1]$ in $S^* \Rightarrow S^*$ is $\{n\} \cup \text{opt sol for } \{1, \ldots, p(n)\}$

---

Formalize this:

**Def:** Let $OPT(i)$ be value of optimal solution $S^*$ for $\{1, 2, \ldots, i\}$

**Note:** $S^*$ not necessarily $S^* \setminus \{1, 2, \ldots, i\}$

$OPT(0) = 0$ by convention

If $n \not\in S^*$: $OPT(n) = OPT(n-1)$

If $n \in S^*$: $OPT(n) = u_n + OPT(p(n))$

$OPT(n) = \max(OPT(n-1), u_n + OPT(p(n)))$
\textbf{Thm}: \text{OPT}(i) = \max \left( v_j + \text{OPT}(p(j)) , \text{OPT}(i-1) \right) \\
\forall 1 \leq j \leq n

\textbf{PE}: By def, \exists feasible solutions of value
\text{OPT}(i-1) \quad \text{(don’t include } j \text{)}
\quad v_j + \text{OPT}(p(j)) \quad \text{(include } j \text{)}
\quad \Rightarrow \text{OPT}(i) \geq \max \left( v_j + \text{OPT}(p(j)) , \text{OPT}(i-1) \right)

\Rightarrow \text{OPT}(i) \geq \max \left( v_j + \text{OPT}(p(j)) , \text{OPT}(i-1) \right)

\Rightarrow \text{OPT}(i) \geq \max \left( v_j + \text{OPT}(p(j)) , \text{OPT}(i-1) \right)

\Rightarrow \text{OPT}(i) \geq \max \left( v_j + \text{OPT}(p(j)) , \text{OPT}(i-1) \right)

\Rightarrow \text{OPT}(i) \geq \max \left( v_j + \text{OPT}(p(j)) , \text{OPT}(i-1) \right)
Suggests obvious algorithm:

```plaintext
Schedule(j) {
    if j = 0 return 0
    else return max(schedule(j-1), v_j + schedule(p(j))
}
```

**Thm:** Schedule(i) returns OPT_C(i)

**PF:** Induction on j.

- **Base case:** j = 0. Schedule(0) returns 0 = OPT_C(0)

- **Inductive case:** Schedule(i) returns
  
  ```plaintext
  max(Schedule(i-1), v_i + Schedule(p(i)))
  = max(OPT_C(i-1), v_i + OPT_C(p(i))) (induction)
  = OPT_C(i) (previous thm)
  ```
Running Time: \( S(p) = 3(2^j) \) 

Recursion tree:

\[ T(n) = \text{total # of calls for} \]
\[ S\left(\frac{c}{2}\right) \]
\[ T(n) = T(n-1) + T(n-2) \]
\[ \approx 2 \times e^c(n) \]
Table $M$ of size $n$. Initially all empty

schedule($i$) {
  if $i=0$ then return 0
  else if $M[i]$ nonempty then return $M[i]$
  else if
    $M[i] = \max\{\text{schedule}(i-1), v_i + \text{schedule}(p(i))\}$
    return $M[i]$
}

Correctness: Same as before

Running Time:

On call to schedule($i$):
- either return entry from table ($O(1)$ time), or
- two recursive calls, then fill in table entry which was empty

So total time = $O(1) \times \#\text{recursive calls}$

Fill in table entry every $2$ recursive calls

$\Rightarrow \leq 2n$ recursive calls

$\Rightarrow O(n)$ running time
Finding the Solution:

```java
Solution(j) {
    if (j = 0) return ∅
    else if (v_i + M[p(j)] > M[j-1])
        return {i} U Solution(p(j))
    else return Solution(i-1)
}
```
Memoization vs. Iteration:

Previous alg was "top-down"
- started at n, worked down with recursion

"Bottom up": start from 0, work up (iterate)

Schedule {
    M[0] = 0
    For (i = 1 to n) {
        M[i] = max (v[i] + M[p(i)], M[i-1])
    }
    return M[n]
}

Correctness: induction on i that M[i] <= OPT(i)

Running time: O(n), obvious!
Principles of Dynamic Programming

1) Polynomial # subproblems

2) Optimal Substructure: Optimal solution to subproblem can be easily computed from optimal solutions to "smaller" subproblems