Lecture 16 – Dynamic Programming

Reading: KT Sections 6.1 and 6.2

Algorithm techniques

Data structures
- Use extra data structures
- Exploit the structure to improve complexity

Greedy algorithms
- Build up a solution incrementally
- Myopically optimizing some local criterion

Divide and conquer
- Break up a problem into independent subproblems
- Solve each subproblem
- Combine solutions to subproblems to form solution to original problem

Dynamic Programming
- Break up a problem into a series of overlapping subproblems
- Build up solutions to larger and larger subproblems
A bit of history

Bellman. Pioneered the systematic study of dynamic programming in 1950s.

Etymology.
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
Weighted Interval Scheduling problem

Weighted interval scheduling problem:
• Job \( j \) starts at \( s_j \), finishes at \( f_j \), and has weight or value \( v_j \).
• Two jobs compatible if they don’t overlap.
• Goal: find maximum-weight subset of mutually compatible jobs.

Let’s try solving it

• Will greedy work?

• How about divide and conquer?

• Is there some structure in the problem that we can exploit?
Let’s define a few notions

**Notation.** Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

**Def.** $p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.

**Ex.** $p(8) = 5, p(7) = 3, p(2) = 0$.

---

More notations

**Notation.** $OPT(j) =$ value of optimal solution to the problem consisting of job requests $1, 2, \ldots, j$.

**Goal.** $OPT(n) =$ value of optimal solution to the original problem.

**Case 1.** $OPT(j)$ selects job $j$.
- Collect profit $v_j$.
- Can’t use incompatible jobs $\{ p(j) + 1, p(j) + 2, \ldots, j - 1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$.

**Case 2.** $OPT(j)$ does not select job $j$.
- Must include optimal solution to problem consisting of remaining jobs $1, 2, \ldots, j - 1$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ v_j + OPT(p(j)), OPT(j-1) \} & \text{otherwise} \end{cases}$$
A brute force solution

**BRUTE-FORCE** \((n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)\)

Sort jobs by finish time so that \(f_1 \leq f_2 \leq \ldots \leq f_n\).
Compute \(p[1], p[2], \ldots, p[n]\).
**RETURN** \(\text{COMPUTE-OPT}(n)\).

**COMPUTE-OPT**\((j)\)

**IF** \(j = 0\)

**RETURN** \(0\).

**ELSE**

**RETURN** \(\max \{ v_j + \text{COMPUTE-OPT}(p[j]), \text{COMPUTE-OPT}(j-1) \} \).

Memoization

**Top-down dynamic programming (memoization).** Cache result of each subproblem; lookup as needed.

**TOP-DOWN** \((0, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)\)

Sort jobs by finish time so that \(f_1 \leq f_2 \leq \ldots \leq f_n\).
Compute \(p[1], p[2], \ldots, p[n]\).
\(M[0] \leftarrow 0\). \quad \text{global array } M[ ]
**RETURN** \(\text{M-COMPUTE-OPT}(n)\).

**M-COMPUTE-OPT**\((j)\)

**IF** \(M[j] = \text{uninitialized}\)

\(M[j] \leftarrow \max \{ v_j + \text{M-COMPUTE-OPT}(p[j]), \text{M-COMPUTE-OPT}(j-1) \} \).
**RETURN** \(M[j]\).
Bottom-up dynamic programming

Bottom-up dynamic programming. Unwind recursion.

\[ \text{BOTTOM-UP} \ (n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n) \]

Sort jobs by finish time so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

Compute \( p[1], p[2], \ldots, p[n] \).

\[ M[0] \leftarrow 0. \quad \text{previously computed values} \]

\[ \text{FOR } j = 1 \text{ TO } n \quad \text{FOR } j = 1 \text{ TO } n \]

\[ M[j] \leftarrow \max \{ v_j + M[p[j]], M[j-1] \}. \]

\[ \text{Running time. The bottom-up version takes } O(n \log n) \text{ time.} \]