Lecture 17 – 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: \( j_1 \leq j_2 \leq \ldots \leq j_n \).

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

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

More notations

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

- Goal. \( OPT(0) \) = 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, ..., \( p(j) \).

- Optimal substructure property (proof via exchange argument).

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

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

**Brute-Force** (n, r1, r2, ..., f1, f2, ..., f_n)

Sort jobs by finish time so that f_o ≤ f_1 ≤ ... ≤ f_n.
Compute f_{p[1]}, f_{p[2]}, ..., f_{p[n]}.
RETURN Compute-Opt(n).

**Compute-Opt(j)**

If j = 0
RETURN 0.
ELSE
RETURN max { c_j + Compute-Opt(j/j), Compute-Opt(j-1) }.

Memoization

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

**Top-Down** (n, r1, r2, ..., f1, f2, ..., f_n)

Sort jobs by finish time so that f_o ≤ f_1 ≤ ... ≤ f_n.
Compute f_{p[1]}, f_{p[2]}, ..., f_{p[n]}.
M[0] = 0.
RETURN M-Compute-Opt(n).

**M-Compute-Opt(j)**

If M[j] = uninit:
M[j] = max { c_j + M-Compute-Opt(j/j), M-Compute-Opt(j-1) }.
RETURN M[j].

Bottom-up dynamic programming

**Bottom-up**. Unwind recursion.

**Bottom-Up** (n, r1, r2, ..., f1, f2, ..., f_n)

Sort jobs by finish time so that f_o ≤ f_1 ≤ ... ≤ f_n.
Compute f_{p[1]}, f_{p[2]}, ..., f_{p[n]}.
M[0] = 0.
FOR j = 1 TO n
M[j] = max { c_j + M[p[j]], M[j-1] }.
RETURN M[n].

**Running time**. The bottom-up version takes O(n log n) time.