Dynamic Programming

CMSC250H

February 16, 2026

Dynamic Programming

Definition

Dynamic Programming: An algorithm technique that explores the space of all possible solutions by carefully decomposing things into a series of subproblems, and building up correct solutions to larger and larger subproblems.

Dynamic Programming

1. Sounds like brute force, but systematically works through the exponential large set of possible solutions
2. Doesn't look at all solutions, takes optimal subproblem solution to build up larger ones
3. Store subproblem solutions, so we don't have to compute again (memoization)

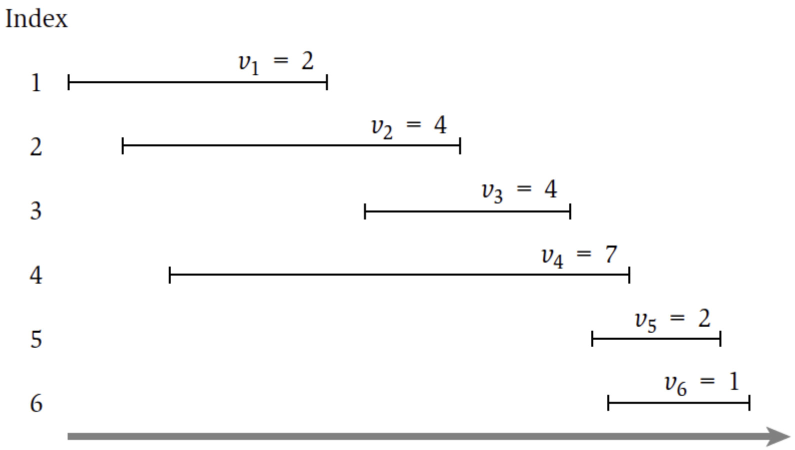
Example) Weighted Interval Scheduling

Weighted Interval Scheduling

Given a set of scheduling requests $\{1, 2, \ldots n\}$. Each has a start and finish time $s(i)$ and $f(i)$, as well as a weight $w(i)$. Generate a set S of requests that maximizes the total weight

$$\max_{S \subseteq \{1,\ldots,n\}} \left[\sum_{i \in S} w(i) \right]$$

Example



Creating a Solution

- We know that if we add an interval i to the set, then we can't add any conflicting requests.
- Let $p(j)$ denote the largest index $i < j$, such that i and j are disjoint
- Observe the following two cases (working backwards):
  - Either n is in the optimal set
  - Or n is not in the optimal set

Case 1)

- What can we say about S if n is in the optimal solution?

Case 1)

- What can we say about S if n is in the optimal solution?
    - If n is in the solution, then the intervals $p(n)+1, p(n)+2, \ldots, n-1$ cannot be in the S (since they overlap with n).
    - So, the optimal solution is reduced to finding the optimal solution up until $p(j)$

$$\mathrm{OPT}(n) = w(n) + \mathrm{OPT}(p(n))$$

Case 2)

- What can we say about S if n is not in the optimal solution?
  - S may contain any interval from the set $\{1, 2, \dots, n-1\}$
  - So, our problem is reduced to inspecting these elements

$$\mathrm{OPT}(n) = \mathrm{OPT}(n-1)$$

Putting It Together

- These two cases now develop into a recurrence to solve the problem:

Optimal Solution For Weighted Interval Scheduling

$$\mathrm{OPT}(n) = \max(n + \mathrm{OPT}(p(j)), \mathrm{OPT}(p(j)))$$

Memoization

- There are still an exponential amount of subproblems here (The number of subsets 2^n)
- We can instead store the solution to a polynomial amount of smaller problems and build up

Algorithm

$M[0] = 0$

for $i = 1, \ldots, n$ **do**

$\quad M[i] = \max(w(i) + M[p(i)], M[i-1])$

or recursively

if $i = 0$ **then**

$\quad$ **return** 0

else if $M[i] \neq$ empty **then**

$\quad$ **return** $M[i]$

else

$\quad$ **return** $M[j] = \max(w(i) + M[p(i)], M[i-1])$

Runtime

Observe that the runtime of the algorithm is $O(n)$, since the number of recursive calls and iterations is bounded by the size of M, which is size $n + 1$

Example 2) Weighted Knapsack

Weighted Knapsack

Say we have a single knapsack, that can hold at most weight W. We also have a list of items $\{1, \ldots, n\}$, each with weight $w(i)$. We want a set S with maximum weight, without going over W

$$\max_{S \subseteq \{1,\ldots,n\}} \sum_{i \in S} w(i) \leq W$$

New Strategy

- We can't rule out conflicting items (like in Interval Scheduling), i.e selecting n eliminates $p(n) + 1, \ldots, n - 1$
- So what does happen if we add n?

Considering the Weight

- We do know that if we add n to S, then we now have remaining weight $W - w(n)$ for all the remaining items
- If we don't add n to S then we have weight W for all the remaining items

The Recurrence

We can now parameterize the recurrence by both the weight and the items:

Optimal Solution for Weighted Knapsack

$$\mathrm{OPT}(n, W) = \max(\mathrm{OPT}(n-1, W), w(n) + \mathrm{OPT}(n-1, W-(w(n)))$$

This gives us an algorithm like before, but with a $n \times W$ size array. Hence, the algorithm runs in $O(W \cdot n)$ time