5. Dynamic Programming

When to use dynamic programming?

- A ~~greedy~~ dynamic programming algorithm proceeds by:
  - Making a choice ~~based on a simple, local criterion~~.
  - Solving the subproblem that results from that choice.
  - Combining the choice and the subproblem solution.
- This technique is useful when problem instances contains overlapping subproblems.

When to use dynamic programming?

- Overlapping subproblems:
  - If S has subproblems A and B,
  - Then some smaller subproblems are subproblems of both A and B.
- So
  - If you have a simple, local criterion to make a choice that leads to an optimal solution → Greedy.
  - If you don't and subproblems overlap → Dynamic Programming.
  - If subproblems don't overlap → Divide and Conquer.

Designing a DP algorithm

- Step 1: determine the subproblems we may get by making a choice.
  - e.g.: intervals 1 to j (where $j \le n$) for the weighted interval scheduling problem.
  - e.g.: points p_1 to p_i (where $i \le n$) for the segmented least square problem.
  - e.g.: items 1 to i and weight w (where $i \le n$ and $w \le W$) for the knapsack problem.

Designing a DP algorithm

- Step 2: define a recurrence relation for
  - the optimal value of the objective function for the problem

  in terms of
  - its optimal value(s) for one or more subproblems.

- E.g. $OPT(j) = \max \{ v_j + OPT(p(j)), OPT(j-1) \}$

Designing a DP algorithm

- Every DP algorithm relies on a table:
  - it contains the optimal values of the objective functions for the subproblems.
  - it's used to avoid solving each subproblem more than once.

M[0,0]	M[0,1]	M[0,2]	M[0,3]	M[0.4]	M[0.5]
M[1,0]	M[1,1]	M[1,2]	M[1,3]	M[1,4]	M[1,5]
M[2,0]	M[2,1]	M[2,2]	M[2,3]	M[2,4]	M[2,5]

Designing a DP algorithm

- Step 3: determine the “shape” of the table
  - its dimensions will depend on the number and range of the parameters used to define the subproblems.
  - it is usually an array with as many dimensions as a subproblem has defining parameters.
    - e.g. M[1 ... n] for weighted interval scheduling.
    - e.g. M[0 ... n, 0 ... W] for knapsack.
  - sometimes we store the table information elsewhere
    - e.g. in the nodes of a graph if the algorithm is solving a problem on this graph.

Designing a DP algorithm

- Step 4: implement the algorithm
  - Approach 1: memoization (recursion)
    - Like the straightforward recursive approach to compute the value given by the recurrence relation
    - But we first look at the table to see if this solution has already been computed.
    - If so, we return it immediately.

Designing a DP algorithm

- Step 4: implement the algorithm
  - Approach 2: iteration (some people only call this approach "dynamic programming")
    - Loop over the subproblems
      - Start by base cases.
      - Then solve increasingly large subproblems.
      - Make sure that by the time subproblem S needs the solution to suproblem S', it's already been computed.
    - For each subproblem
      - Compute the optimal solution to the subproblem.
      - Using table lookups instead of recursive calls.

Designing a DP algorithm

- Step 5: retrieve the optimal solution
  - Step 4 gives us the value of the objective function for the optimal solution, but not the solution.
  - To retrieve the solution:
    - start with the original problem and determine the last choice we made (to get the value of the objective function).
    - insert this choice into the solution (usually at the end).
    - then repeat starting from the subproblem we get after making that choice.
  - We do not want to store solutions for every subproblem, because it would increase the space (and possibly time) requirements by a linear factor.