Dynamic Programing

- The main idea is to solve a big problem by combining the solutions to smaller subproblems.
- We usually want to maximize profit or minimize cost.
- A dynamic programming algorithm works by creating an array of related but simpler subproblems, and then, it computes the solution to the big complicated problem by using the solutions to the easier subproblems which are stored in this array.
- A dynamic programming solution always has three parts:
  1. Define an array of subproblems.
  2. Find a recurrence.
  3. Write the program.

Longest Monotone Subsequence Problem

- **input:** $a_1, a_2, \ldots, a_d \in \mathbb{N}$
  output: $L =$ length of the longest monotone non-decreasing subsequence.
- The subsequence need not be consecutive.

  Thus $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ is a monotone subsequence provided:

  1. $1 \leq i_1 < i_2 < \ldots < i_k \leq d$
  2. $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_k}$.

  Ex. The length of the longest monotone subsequence of $\{4, 6, 5, 9, 1\}$ is 3.

1st step: Define an array of subproblems: $R(j) =$ "length of the LMS which ends in a_j". Answer: $L = \max_{1 \leq j \leq n} R(j)$.

2nd step: Find a recurrence: let $R(1) = 1$, and for $j > 1$,

$$R(j) = \begin{cases} 1 & \text{if } \forall i[1 \leq i < j \supset a_i > a_j] \\ 1 + \max\limits_{1 \leq i < j} \{R(i) | a_i \leq a_j\} & \text{otherwise} \end{cases}$$

3rd step: Write a program that computes R:

$R(1) \leftarrow 1$
for $(j = 2; j \leq d; j++)$
  $\max \leftarrow 0$
  for $(i = 1; i \leq j - 1; i++)$
    if $(R(i) > \max$ and $a_i \leq a_j)$ then
      $\max \leftarrow R(i)$
  $R(j) \leftarrow \max +1$

All Pairs Shortest Path Problem

- **Input:**
  1. Directed graph $G = (V, E)$, $V = \{1, 2, ..., n\}$.
  2. Cost $C(i, j) \in \mathbb{R}^+ \cup \{\infty\}$ (think of the costs as distances), $1 \leq i, j \leq n$, $C(i, j) = \infty$ if (i, j) is not an edge.

  Output: Find $D(i, j)$, the length of the shortest directed path from i to j, for all i, j.

- Note that for a general graph on n nodes there are exponentially many (in n) paths, so exhaustive search is not feasible.

- Notational convention: if $k = 0$ then $\{1, 2, \ldots, k\} = \emptyset$.

1st step: define an array of subproblems

- Let $A(k, i, j)$ be the length of the shortest path from i to j s.t. all *intermediate* nodes on the path are in $\{1, 2, \ldots, k\}$.
- Then $A(n, i, j) = D(i, j)$ is the solution.

2nd step: define a recurrence

- Initialization: for $k = 0$, $A(0, i, j) = C(i, j)$.
- Now for $k > 0$. Assume we know $A(k-1, r, s)$ for all r, s.
  If k is not included in the shortest path between i and j, then $A(k, i, j) = A(k-1, i, j)$.
  If k is included, then it is included exactly once, so $A(k, i, j) = A(k-1, i, k) + A(k-1, k, j)$.

$$\therefore A(k, i, j) = \min\{A(k-1, i, j), A(k-1, i, k) + A(k-1, k, j)\}$$

3rd step: write the program

- It turns out that we only need space for a two-dimensional array $B(i,j) = A(k,i,j)$. To compute $A(k,*,*)$ from $A(k-1,*,*)$, we can overwrite $A(k-1,*,*)$.

- The following algorithm is known as **Floyd's algorithm**:

for $(i = 1; i \le n; i++)$
    for $(j = 1; j \le n; j++)$
        $B(i,j) \longleftarrow C(i,j)$

for $(k = 1; k \le n; k++)$
    for $(i = 1; i \le n; i++)$
        for $(j = 1; j \le n; j++)$
            $B(i,j) \longleftarrow \min\{B(i,j), B(i,k) + B(k,j)\}$

Ex. What is the running time? Why does it work? $B(i,k)$ or $B(k,j)$ may have already been updated, e.g. $k < j$ or $k < i$.