Dynamic Programming

- A powerful paradigm for algorithm design.
- Often leads to elegant and efficient algorithms when greedy or divide-and-conquer don't work.
- DP also breaks a problem into subproblems, but subproblems are not independent.
- DP tabulates solutions of subproblems to avoid solving them again.

Dynamic Programming

- Typically applied to optimization problems: many feasible solutions; find one of optimal value.
- Key is the principle of optimality: solution composed of optimal subproblem solutions.
- Example: Matrix Chain Product.
- A sequence $\langle M_1, M_2, \ldots, M_n \rangle$ of n matrices to be multiplied.
- Adjacent matrices must agree on dim.

Matrix Product

Matrix-Multiply (A, B)

1. **Let** A **be** $p \times q$**; let** B **be** $q \times r$**.**
2. **If dim of** A **and** B **don't agree, error.**
3. **for** $i = 1$ **to** p
4. **for** $j = 1$ **to** r
5. $C[i, j] = 0$
6. **for** $k = 1$ **to** q
7. $C[i, j] \; += \; A[i, k] \times B[k, j]$
8. **return** C**.**

- **Cost of multiplying these matrices is** $p \times q \times r$**.**

Matrix Chain

- **Consider 4 matrices:** M_1, M_2, M_3, M_4.
- **We can compute the product in many different ways, depending on how we parenthesize.**

$$(M_1(M_2(M_3M_4)))$$

$$(M_1((M_2M_3)M_4))$$

$$((M_1M_2)(M_3M_4))$$

$$(((M_1M_2)M_3)M_4)$$

- **Different multiplication orders can lead to very different total costs.**

Matrix Chain

- **Example:** $M_1 = 10 \times 100$**,** $M_2 = 100 \times 5$**,** $M_3 = 5 \times 50$**.**
- **Parentheses order** $((M_1 M_2)M_3)$ **has cost** $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500.$
- **Parentheses order** $(M_1(M_2 M_3))$ **has cost** $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75,000!$

Matrix Chain

- **Input: a chain $\langle M_1, M_2, \ldots, M_n \rangle$ of n matrices.**
- **Matrix M_i has size $p_{i-1} \times p_i$, where $i = 1, 2, \ldots, n$.**
- **Find optimal parentheses order to minimize cost of chain multiplying M_i's.**
- **Checking all possible ways of parenthesizing is infeasible.**
- **There are roughly $\binom{2n}{n}$ ways to put parentheses, which is of the order of 4^n!**

Principle of Optimality

- **Consider computing $M_1 \times M_2 \ldots \times M_n$.**
- **Compute $M_{1,k} = M_1 \times \ldots \times M_k$, in some order.**
- **Compute $M_{k+1,n} = M_{k+1} \times \ldots \times M_n$, in some order.**
- **Finally, compute $M_{1,n} = M_{1,k} \times M_{k+1,n}$.**
- **Principle of Optimality: To optimize $M_{1,n}$, we must optimize $M_{1,k}$ and $M_{k+1,n}$ too.**

Recursive Solution

- **A subproblem is subchain** $M_i, M_{i+1} \ldots, M_j$**.**
- $m[i,j]$ $=$ **optimal cost to multiply** $M_i, \ldots, M_j$**.**
- **Use principle of optimality to determine** $m[i,j]$ **recursively.**
- **Clearly,** $m[i,i] = 0$**, for all** i**.**
- **If an algorithm computes** $M_i, M_{i+1} \ldots, M_j$ **as** $(M_i, \ldots, M_k) \times (M_{k+1}, \ldots, M_j)$**, then**

$$m[i,j] \;=\; m[i,k] \;+\; m[k+1,j] \;+\; p_{i-1}p_k p_j$$

Recursive Solution

$$m[i,j] \;=\; m[i,k] \;+\; m[k+1,j] \;+\; p_{i-1}p_kp_j$$

- **We don't know which k the optimal algorithm will use.**
- **But k must be between i and $j-1$.**
- **Thus, we can write:**

$$m[i,j] \;=\; \min_{i \le k < j} \{m[i,k] \;+\; m[k+1,j] \;+\; p_{i-1}p_kp_j\}$$

The DP Approach

- **Thus, we wish to solve:**

$$m[i,j] \;=\; \min_{i \le k < j} \{m[i,k] \;+\; m[k+1,j] \;+\; p_{i-1}p_k p_j\}$$

- **A direct recursive solution is exponential: brute force checking of all parentheses orders.**
- **What is the recurrence? What does it solve to?**
- **DP's insight: only a small number of subproblems actually occur, one per choice of i, j.**

The DP Approach

- **Naive recursion is exponential because it solves the same subproblem over and over again in different branches of recursion.**
- **DP avoids this wasted computation by organizing the subproblems differently: bottom up.**
- **Start with $m[i, i] = 0$, for all i.**
- **Next, we determine $m[i, i+1]$, and then $m[i, i+2]$, and so on.**

The Algorithm

- **Input:** $[p_0, p_1, \ldots, p_n]$ **the dimension vector of the matrix chain.**
- **Output:** $m[i, j]$**, the optimal cost of multiplying each subchain** $M_i \times \ldots \times M_j$**.**
- **Array** $s[i, j]$ **stores the optimal** k **for each subchain.**

The Algorithm

Matrix-Chain-Multiply (p)

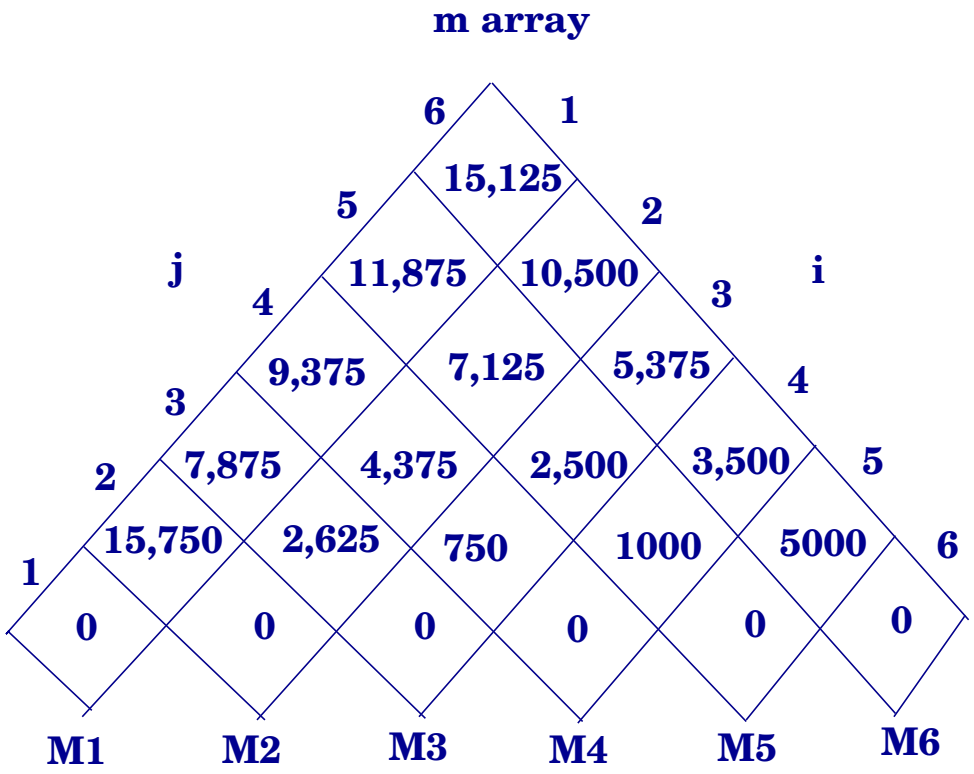
1. **Set** $m[i,i] = 0$**, for** $i = 1, 2, \ldots, n$**.**
2. **Set** $d = 1$**.**
3. **For all** i, j **such that** $j - i = d$**, compute**

$$m[i,j] \;=\; \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\}$$

   Set $s[i,j] = k^*$**, where** k^* **is the choice that gives min value in above expression.**

4. **If** $d < n$**, increment** d **and repeat Step 3.**

Illustration



M1 = 30 x 35

M2 = 35 x 15

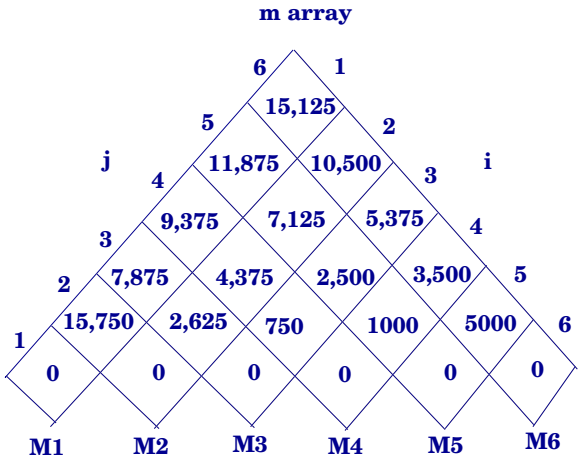
M3 = 15 x 5

M4 = 5 x 10

M5 = 10 x 20

M6 = 20 x 25

Illustration



- **Computing $m[2,5]$.**

$$\min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35.15.20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35.5.20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35.10.20 = 11375 \end{cases}$$

Finishing Up

- **The algorithm clearly takes $O(n^3)$ time.**
- **The m matrix only outputs the cost.**
- **The parentheses order from the s matrix.**

Matrix-Chain (M, s, i, j)

1. **if** $j > i$ **then**
2. $\quad X \leftarrow$ **Matrix-Chain** $(A, s, i, s[i, j])$
3. $\quad Y \leftarrow$ **Matrix-Chain** $(A, s, s[i, j] + 1, j)$
4. **return** $X * Y$

Longest Common Subsequence

- **Consider a string of characters:** $X = ABCBDAB$.
- **A subsequence is obtained by deleting some (any) characters of** X.
- **E.g.** $ABBB$ **is a subsequence of** X**, as is** ABD**. But** $AABB$ **is not a subsequence.**
- **Let** $X = (x_1, x_2, \ldots, x_m)$ **be a sequence.**
- $Z = (z_1, z_2, \ldots, z_k)$ **is subseq. of** X **if there is an index sequence** $(i_1, \ldots, i_k)$ **s.t.** $z_j = x_{i_j}$**, for** $j = 1, \ldots, k$.
- **Index sequence for** $ABBB$ **is** $(1, 2, 4, 7)$.

Longest Common Subsequence

- **Given two sequences X and Y, find their longest common subsequence.**
- **If $X = (A, B, C, B, D, A, B)$ and $Y = (B, D, C, A, B, A)$, then (B, C, A) is a common sequence, but not LCS.**
- **(B, D, A, B) is a LCS.**
- **How do we find an LCS?**
- **Can some form of Greedy work? Suggestions?**

Trial Ideas

- **Greedy-1: Scan X. Find the first letter matching y_1; take it and continue.**
- **Problem: only matches prefix substrings of Y.**
- **Greedy-2: Find the most frequent letters of X; or sort the letters by their frequency. Try to match in frequency order.**
- **Problem: Frequency can be irrelevant. E.g. suppose all letters of X are distinct.**

Properties

- 2^m **subsequences of** X**.**
- **LCS obeys the principle of optimality.**
- **Let** $X_i = (x_1, x_2, \ldots, x_i)$ **be the** i**-long prefix of** X**.**
- **Examples: if** $X = (A, B, C, B, D, A, B)$**, then** $X_2 = (A, B)$**;** $X_5 = (A, B, C, B, D)$**.**

LCS Structure

- **Suppose $Z = (z_1, z_2, \ldots, z_k)$ is a LCS of X and Y. Then,**

  1. **If $x_m = y_n$, then $z_k = x_m = y_n$ and**
     $$Z_{k-1} = LCS(X_{m-1}, Y_{n-1}).$$
  2. **If $x_m \neq y_n$, then $z_k \neq x_m$ implies**
     $$Z = LCS(X_{m-1}, Y).$$
  3. **If $x_m \neq y_n$, then $z_k \neq y_n$ implies**
     $$Z = LCS(X, Y_{n-1}).$$

Recursive Solution

- **Let $c[i,j] = |LCS(X_i, Y_j)|$ be the optimal solution for X_i, Y_j.**
- **Obviously, $c[i,j] = 0$ if either $i = 0$ or $j = 0$.**
- **In general, we have the recurrence:**

$$c[i,j] = \begin{cases} 0 & \text{if } i \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\{c[i, j-1],\ c[i-1, j]\} & \text{if } x_i \neq y_j \end{cases}$$

Algorithm

- **A direct recursive solution is exponential:** $T(n) = 2T(n-1) + 1$**, which solves to** 2^n**.**
- **DP builds a table of subproblem solutions, bottom up.**
- **Starting from** $c[i, 0]$ **and** $c[0, j]$**, we compute** $c[1, j], c[2, j]$**, etc.**

Algorithm

LCS-Length (X, Y)

$c[i, 0] \leftarrow 0$, $c[0, j] \leftarrow 0$, **for all** i, j;
for $i = 1$ **to** m **do**
  for $j = 1$ **to** n **do**
    if $x_i = y_j$ **then**
      $c[i, j] \leftarrow c[i-1, j-1] + 1$; $b[i, j] \leftarrow D$
    else if $c[i-1, j] \geq c[i, j-1]$ **then**
      $c[i, j] \leftarrow c[i-1, j]$; $b[i, j] \leftarrow U$
    else
      $c[i, j] \leftarrow c[i, j-1]$; $b[i, j] \leftarrow L$
return b, c

LCS Algorithm

- **LCS-Length (X, Y) only computes the length of the common subsequence.**
- **By keeping track of matches, $x_i = y_j$, the LCS itself can be constructed.**

LCS Algorithm

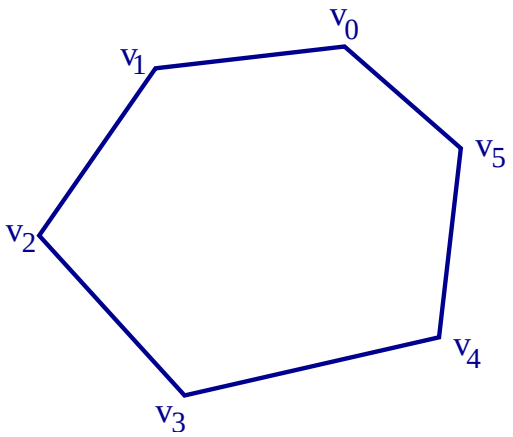
PRINT-LCS(b, X, i, j)

```
if i = 0 or j = 0 then return
if b[i, j] = D then
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] = U then
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

- **Initial call is PRINT-LCS**$(b, X, |X|, |Y|)$**.**
- **By inspection, the time complexity of the algorithm is** $O(nm)$**.**

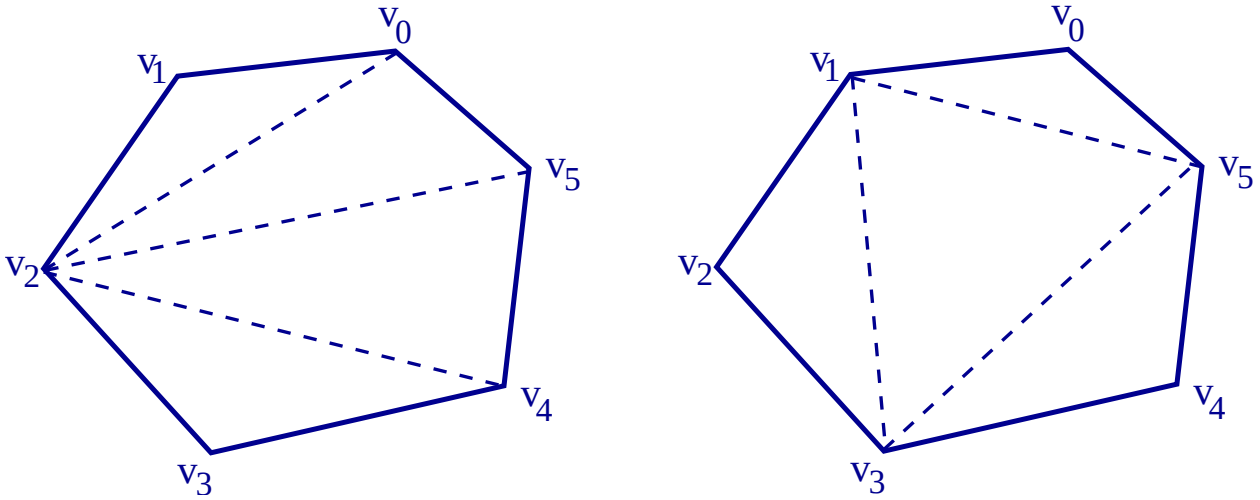
Optimal Polygon Triangulation

- **Polygon is a piecewise linear closed curve.**
- **Only consecutive edges intersect, and they do so at vertices.**
- **P is convex if line segment xy is inside P whenever x, y are inside.**



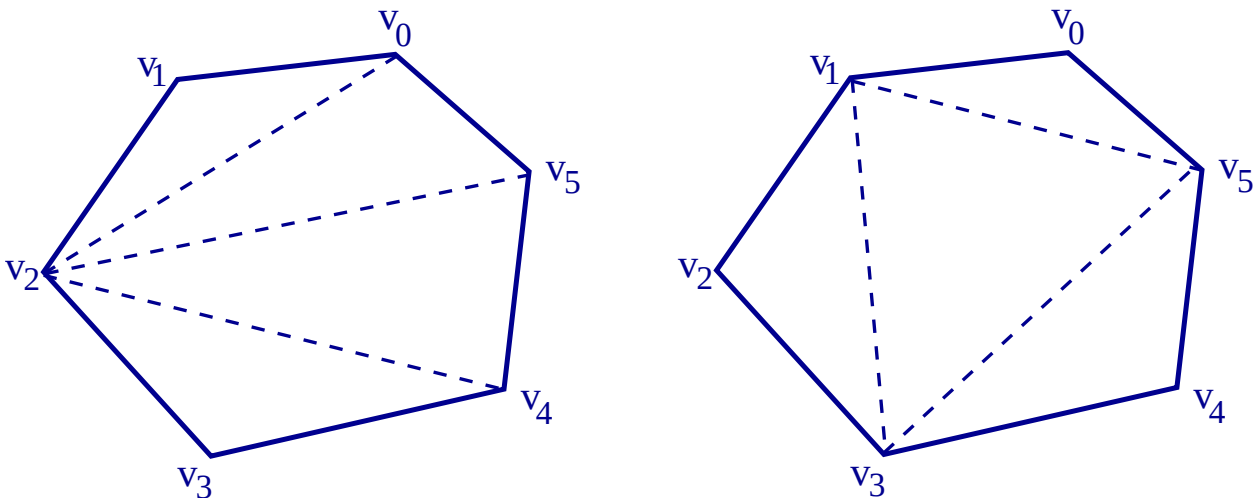
Optimal Polygon Triangulation

- **Vertices in counter-clockwise order:** $v_0, v_1, \ldots, v_{n-1}$. **Edges are** $v_i v_{i+1}$**, where** $v_n = v_0$**.**
- **A chord** $v_i v_j$ **joins two non-adjacent vertices.**
- **A triangulation is a set of chords that divide** P **into non-overlapping triangles.**



Triangulation Problem

- **Given a convex polygon $P = (v_0, \ldots, v_{n-1})$, and a weight function w on triangles, find a triangulation minimizing the total weight.**
- **Every triangulation of a n-gon has $n-2$ triangles and $n-3$ chords.**



Optimal Triangulation

- **One possible weight:**

$$w(\triangle v_i v_j v_k) \;=\; |v_i v_j| + |v_j v_k| + |v_k v_i|$$

- **But problem well defined for any weight function.**

Greedy Strategies

- **Greedy 1: Ring Heuristic. Go around each time, skipping one vertex; after logn rounds, done.**
- **Motivation—joining closeby vertices.**
- **Not always optmal. Consider a flat, pancake like convex polygon. The optimal will put mostly vertical diagonals. Greedy's cost is roughly $O(\log n)$ times the perimeter.**

Greedy Strategies

- **Greedy 2: Always add shortest diagonal, consistent with previous selections.**
- **Counter-example by Lloyd.** $P = (A, B, C, D, E)$, **where** $A = (0, 0); B = (50, 25); C = (80, 30); D = (125, 25); E = (160, 0)$**.**
- **Edge lengths are** $BD = 75$; $CE < 86$; $AC < 86$; $BE > 112$; $AD > 127$**.**
- **Greedy puts** BD**, then forced to use** BE**, for total weight** $= 187$**.**
- **Optimal uses** AC, CE**, with total weight** $= 172$**.**

Greedy Strategies

- **$GT(S)$ is within a constant factor of $MWT(S)$ for convex polygons.**
- **For arbitrary point set triangulation, the ratio is $\Omega(n^{1/2})$.**

The Algorithm

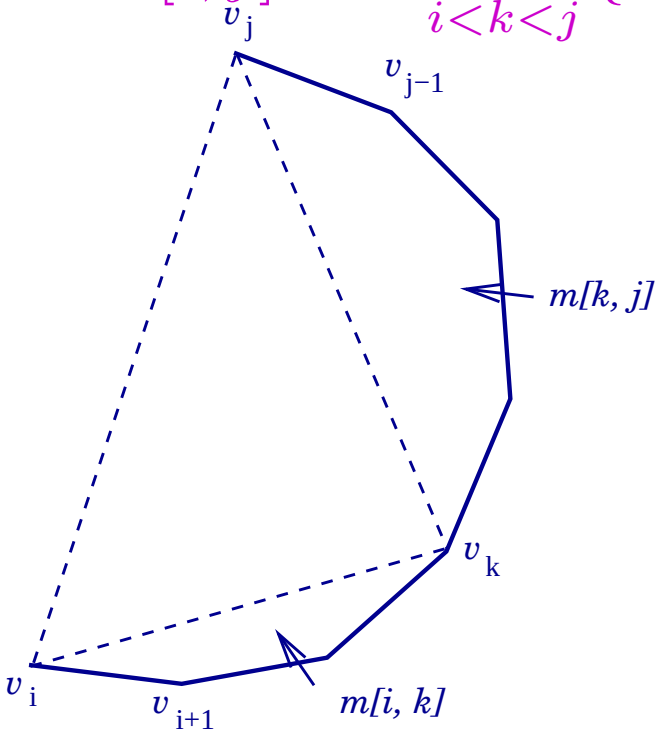
- $m[i,j]$ **be the optimal cost of triangulating the subpolygon** $(v_i, v_{i+1}, \ldots, v_j)$**.**
- **Consider the** $\triangle$ **with one side** $v_i v_j$**.**
- **Suppose the 3rd vertex is** k**.**
- **Then, the total cost of the triangulation is:**

$$m[i,j] \;=\; m[i,k] + m[k,j] + w(\triangle v_i v_j v_k)$$

The Algorithm

- **Since we don't know k, we choose the one that minimizes this cost:**

$$m[i,j] = \min_{i<k<j} \{m[i,k] + m[k+1,j] + w(\triangle v_i v_j v_k)\}$$



All-Pairs Shortest Paths

- **Given $G = (V, E)$, compute shortest path distances between all pairs of nodes.**
- **Run single-source shortest path algorithm from each node as root. Total complexity is $O(nS(n, m))$, where $S(n, m)$ is the time for one shortest path iteration.**
- **If non-negative edges, use Dijkstra's algorithm: $O(m \log n)$ time per iteration.**
- **With negative edges, need to use Bellman-Ford algorithm: $O(nm)$ time per iteration.**

Floyd-Warshall Algorithm

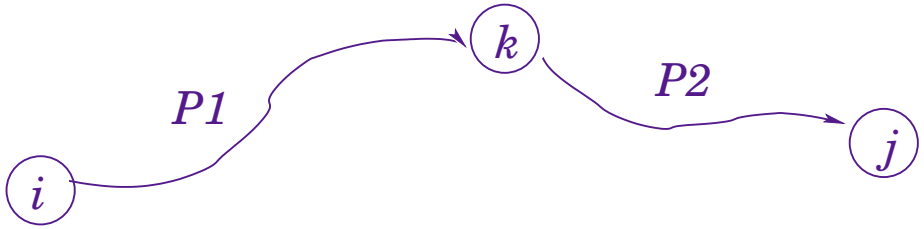
- **$G = (V, E)$ has vertices $\{1, 2, \ldots, n\}$. W is cost matrix. D is output distance matrix.**

algorithm Floyd-Warshall

1. $D = W$**;**
2. for $k = 1$ **to** n
3. **for** $i = 1$ **to** n
4. **for** $j = 1$ **to** n
5. $d_{ij} = \min\{d_{ij}, \quad d_{ik} + d_{kj}\}$
6. return D**.**

Correctness

- P_{ij}^k : **shortest path whose intermediate nodes are in $\{1, 2, \ldots, k\}$.**
- **Goal is to compute P_{ij}^n, for all i, j.**



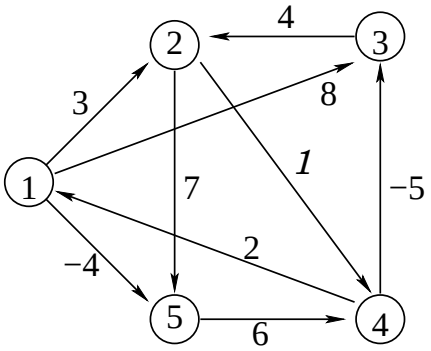
- **Use Dynamic Programming. Two cases:**
  1. **Vertex k not on P_{ij}^k. Then, $P_{ij}^k = P_{ij}^{k-1}$.**
  2. **Vertex k is on P_{ij}^k. Then, neither P_1 nor P_2 uses k as an intermediate node. in its interior. (Simplicity of P_{ij}^k.) Thus, $P_{ij}^k = P_{ik}^{k-1} + P_{kj}^{k-1}$**

Correctness

- **Recursive formula for P_{ij}^k:**

  1. **If $k = 0$, $P_{ij}^k = c_{ij}$.**
  2. **If $k > 0$, $d_{ij}^k = \min\{d_{ij}^{k-1}, \quad d_{ik}^{k-1} + d_{kj}^{k-1}\}$**

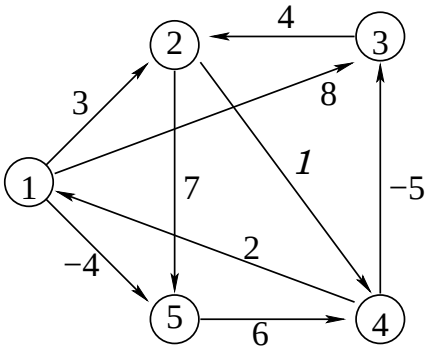
Example



- **Matrices D_0 and D_1:**

$$\begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Example



- **Matrices D_2 and D_5:**

$$\begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$