All pairs Shortest Paths
Input: a connected graph $G = (V, E)$ with edge weights $w(e)$ for all $e \in E$ Vertex $s \in V$.
Output: For each $u, v \in V$, a shortest path (i.e., minimum weight) from u to v.

W.L.O.G let's assume $V = \{1, 2, \cdots, n\}$. We define the following semantic array.

$V[u, v, k]$ =the weight of the shortest path from u to v such that the path just goes through nodes in the subset of $\{1, 2, \cdots, k\}$.
In other words, among all paths from u to v that use just the vertices in $\{1, 2, \cdots, k\}$ we get the minimum weight.

These values can be calculated as follows:

$$V[u, v, 0] = \begin{cases} 0 & \text{if } u = v \\ w(u, v) & \text{if } (u, v) \text{ is an edge} \\ \infty & \text{otherwise} \end{cases}$$

and

$$V[u, v, k] = \min(V[u, v, k-1], V[u, k, k-1] + V[k, v, k-1])$$

Explanation for $V[u, v, k]$: This value is the weight of the shortest path from u to v when the path goes through some nodes in the subset of $\{1, 2, \cdots, k\}$. There are two possibilities:

1. The path does not go through node k. Therefore, the minimum weight among all these paths is $V[u, v, k-1]$.
2. The path goes through node k. Therefore this path can be segmented into two parts: in the first part it goes from u to node k using some nodes in the subset of $\{1, 2, \cdots, k-1\}$; the second part it continues from node k and finishes at node v using some nodes in the subset of $\{1, 2, \cdots, k-1\}$. The minimum weight among all these paths occurs when we find the shortest path in each part. Thus the minimum weight here is $V[u, k, k-1] + V[k, v, k-1]$.

This algorithm is called **Floyd-Warshall** algorithm. This algorithm creates a 3 dimensional table that stores the values of V. The first dimension is for different source nodes (n nodes), the second dimension is for different destination nodes (n nodes) and the third dimension is for k ($n + 1$ values). We start from $k = 0$, calculate the values of $V[u, v, 0]$ for all $u, v \in V$. Increment k till $k = n$. Calculating the value for each entry in this table takes constant time. Thus, the worst-case time complexity of this algorithm is $\mathcal{O}(n^3)$.

Identify all shortest paths: The shortest path from node u to v is Path(u, v, n).

Function Path(u, v, k)

```
   // Note:  We have already calculated and stored V values.
1  if k = 0 then
2      if u = v then
3          return []
4      else
5          return [(u,v)]
6  if V[u, v, k-1] ⩽ V[u, k, k-1] + V[k, v, k-1] then
7      return Path(u, v, k-1)
8  else
9      return [Path(u, k, k-1),Path(k, v, k-1)]
```

The Knapsack problem
Input: We are given a set of n items $I_1, I_2, \cdots, I_n$ and a knapsack with an integer capacity C. Each item I_j has an integer weight w_j and a value v_j.
Output: Identify a subset of items S that (1) maximizes the sum of the values of items in S, and (2) the sum of the weight of items in S is at most C.

Note: This version of the problem is also called $\{0,1\}$-knapsack because each item can be picked at most once (two options for item I_j: (1) $I_j \in S$, and (2) $I_j \notin S$).

Semantic array:
$\boldsymbol{V[i,c]}$ **= the maximum value possible using only the first** $\boldsymbol{i}$ **items and a knapsack of capacity** $\boldsymbol{c}$**.**

These values can be calculated as follows:

$$V[i,c] = \begin{cases} 0 & \text{if } i = 0 \text{ or } c = 0 \\ \max\{V[i-1,c], V[i-1,c-w_i] + v_i\} & \text{otherwise} \end{cases}$$

Explanation: To find the optimal value $V[i,c]$, either item I_i is needed or it's not needed.

Note: In the calculation of $\max\{V[i-1,c], V[i-1,c-w_i] + v_i\}$, the second part is invoked only if $w_i \leqslant c$.

The solution of this problem is $V[n,C]$. To calculate this value we create a 2 dimensional table V. There are $n+1$ rows and $C+1$ columns. Each entry in this table takes constant time. Thus, the running time is $\mathcal{O}(nC)$.

The last step is to identify the items in the optimal solution. We use V values to decide if we should choose item I_i or not. Try to find out how you can make this decision. It's similar to other DP examples and is easy!

Chain Matrix Multiplication

Suppose we want to multiply three matrices A, B, C. To calculate the result of this multiplication, we should iteratively multiply two matrices each time. Note that matrix multiplication is not commutative (i.e., $A \times B \neq B \times A$); however, it is associative (i.e., $A \times (B \times C) = (A \times B) \times C$). So to do this multiplication we can parenthesize in different ways (e.g., A(BC) or (AB)C). All will yield same answer but not same running time. Using classical matrix multiplication, multiplying a matrix with dimensionality $p \times q$ and a matrix with dimensionality $q \times r$ costs pqr operations.

Example: Consider three matrices A, B, C with dimensions (respectively) 5×10, 10×100, 100×50. Using $(A \times B) \times C$ costs $5000 + 25000 = 30000$ operations. Using $A \times (B \times C)$ costs $50000 + 2500 = 52500$ operations. As you can see the order of multiplications can significantly change the total run time.

The goal is to identify the optimal order.

Input: There are n matrices $M_1, \cdots, M_n$. Matrix M_i has dimension $d_{i-1} \times d_i$.
Output: Identify the optimal parenthesized product with smallest total cost.

Note: The input in this problem is the dimensions of the matrices not the actual matrix entries.

The optimal parenthesization (the optimal order of pairwise products) can be represented by an optimal parse tree. The leaves are the matrices. Each internal node represents a pairwise matrix multiplication. The root is the last pairwise multiplication. The subtrees are the subproblems that must be computed optimally.

Semantic array:
$\boldsymbol{C[i,j]}$ **= the cost of an optimal product ordering of** $\boldsymbol{M_i \times \cdots \times M_j}$**.**

These values can be calculated as follows:

$$C[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\{C[i,k] + C[k+1,j] + d_{i-1}d_k d_j : i \leqslant k < j\} & \text{if } i < j \end{cases}$$

Identifying the optimal way to parenthesize the product: store the optimal k (the break point) for each $C[i,j]$ in another array $B[i,j]$.

$$B[i,j] = \begin{cases} -1 & \text{if } i = j \\ \arg\min\{C[i,k] + C[k+1,j] + d_{i-1}d_k d_j : i \leqslant k < j\} & \text{if } i < j \end{cases}$$

The following function identifies the best way to parenthesize the matrices.

Function Parenthesize(i, j)

```
1 if i = j then
2 |  print M_i
3 print "(" + Parenthesize(i, B[i, j]) + "×" + (B[i, j] + 1, j) + ")"
```

The optimal way to parenthesize $M_1 M_2 \cdots M_n$ is Parenthesize$(1, n)$.