All Pair Shortest Paths.

Vamos a ocupar el algoritmo de **Floyd-Warshall** para resolver el problem de APSP(all pair shortest path) es decir, los caminos mínimos por cada par de vértices.

Ya conocemos algunos algoritmos para encontrar caminos mínimos como el algoritmo de Dijkstra y de Bellman-Ford y simplemente podríamos hacer V llamdas de estos algoritmos pero podemos hacer algo mejor y con código más corto.

Usaremos una técnica de programación Dinámica para encontrar esto, el algoritmo es el algoritmo de Floyd-Warshall.

Formulación de programación dinámica.

En el algoritmo de Floyd-Warshall vamos a estar considerando vértices intermedios, ¿ a qué nos referimos con esto? que si tenemos un camino $p_j = \{p_j, p_l \dots, p_k, p_i\}$ el cual representa el camino mínimo de $p_j \to p_i$ entonces cualquier vèrtice en el conjunto $\{p_l, \dots p_k\}$ es un vértice intermedio. Consideramos que tenemos un conjunto de vértices $V = \{0, \dots, n-1\}$ tomemos un subconjunto de esto caracterizado por k como $\{0, \dots, k\}$ entonces consideremos para cada $i, j \in V$ consideremos todos los caminos de `i a j` cuyos vértices intermedios están en $\{0, \dots, k\}$ tomemos el mínimo en peso de ellos sea p este, notemos que si k es un vértice intermedio del camino mínimo de p entonces el camino $i \to k$ y $k \to j$ son los caminos mínimos entre esos vértices también.(desigualdad del triángulo en caminos minimos.) $\delta(i,j) \leq \delta(i,k) + \delta(k,j) \forall k \in V$ pero sabemos que el vértice k podría no estar en el camino p por lo que tomamos dos decisiones en cada momento, que es lo que pasaría si k esta y que pasa si k no está. De esta forma podemos formar nuestra función recursiva para nuestra programación dinámica.

$$d_{i,j}^k = \begin{cases} w_{i,j} & si \quad k = -1 \\ \\ \min(d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}) & si \quad k \geq 0 \end{cases}$$

Es decir, tomamos el minimo entre no tamar el vértice k que es considerar $i \to j$ con $\{0, \dots, k-1\}$ y tomarlo que es $i \to k + k \to j$ como ya lo tomamos ya consideramos $\{0, .., k-1\}$.

Podemos implementar esto en la forma Bottom-Up con tres sencillos loops.

Notemos que **NECESITAMOS UNA MATRIZ DE ADYACENCIA** donde si existe una arista entre `i,j` entonces `adjMat[i][j] =` $w_{i,j}$ y `adjMat[i][j] =` ∞ en caso contrario, teneiendo lista esta matriz ña forma bottom del algoritmo queda.

```
for(int k=0; k < V; k++)
    for(int i = 0; i < V; i++)
        for(int j = 0; j < V; j++)
            adjMat[i][j] = min(adjMat[i][j], adjMat[i][k] + adjMat[k][j]);
```

Ahora veamos como podriamos reconstruir estos caminos, podemos llevar una matriz de padres es decir `p[i][j]` guarda el ultimo vértice antes de `j` en el camino `i to j`.

```
///inicialización
for(int i=0; i<V; i++){
    for(int j=0; j<V; j++){
        p[i][j] = i; //inicialmente el camino es únicamente i -> j
    }
}

for(int k=0; k<V; k++){
    for(int i=0; i<V; i++){
        for(int j=0; j<V; j++){
            if(adjMat[i][j] > adjMat[i][k] + adjMat[k][j]){
                adjMat[i][j] = adjMat[i][k] + adjMat[k][j]; //i - - - > k -
- - > j
                p[i][k] = p[k][j];
            }
        }
    }
}
```

y para imprimir los caminos basta usar recursividad.

```
void printpath(int i, int j){
    if( i != j) printpath(i, p[i][j]);
    cout << j << " ";
}
```

¿Complejidad? claramente por los tres ciclos es $\mathbb{O}(V^3)$ asumiendo realizas 10^8 operaciones por segundo si tienes un segundo V tiene que ser del orden de $\sqrt[3]{10^8} \approx 464$