All-pairs shortest paths:

Given directed edge-weighted graph \( G = (V, E) \).

Want to build array \( \text{dist} \[i][j] \) such that \( \text{dist} \[i][j] = \text{distance from } i \text{ to } j \).

Number vertices 0 to \( n-1 \),

\( \text{dist} \[i][j] \) = length of shortest \( i \) to \( j \) path with internal vertices from \( \{0, \ldots, k\} \).
\[ D_{ij}^{n} = w(i \rightarrow j) \]

Goal: compute \( D_{ij}^{n} \) for all \( i, j \)

\[ D_{ij}^{k} = \min \{ D_{ij}^{k-1} + D_{ik}^{k-1} + D_{kj}^{k-1} \mid i, j \} \]

\( O(n^3) \) subproblems

\( O(1) \) per

\( \Rightarrow O(n^3) \) time

\( O(n^3) \) space
Method takes an adjacency matrix `a` storing edge weights. We'll fill in 2D array `d[][]` where `d[i]` is the single source shortest-paths distance array for source `i`.

```java
public static void allPairs(int[][] a, int[][] d) {
    int n = a.length;

    // Initialize d with D[i][j][-1] (no internal vertices)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            d[i][j] = a[i][j];
        }
    }

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (d[i][k] + d[k][j] < d[i][j]) {
                    d[i][j] = d[i][k] + d[k][j];
                }
            }
        }
    }
}
```

By end of iteration `k`,

\[ \text{dist}(i, j) \leq D_{i,j,k} \cdot n \]

**Floyd-Warshall**
Minimum Spanning Tree

Given an edge-weighted, connected, undirected graph $G = (V, E)$.

Spanning tree: an acyclic (tree) subgraph containing all vertices (spanning).

Minimum spanning tree (MST): spanning tree of minimum weight defined with non-negative weight edges.
All spanning trees have \( IVI-1 \) edges.

\[ \implies \text{if edges all have weight 1, then all spanning trees are min} \]

Fact: If edge weights are distinct, then the MST is unique.
The strategy:

Build a collection of edges that all belong to some MST.

In other words, build up a spanning forest that lives inside some MST.

Initially, all V vertices are their own component.

Say we have a spanning forest F. Which edge(s) can we add?
Say edge $e$ leaves a component of $F$ if it has exactly one endpoint in the component.

Lemma: Take any component $S$, take the (some of) lightest edge leaving $S$. This edge belongs to an MST containing $F$. 
Proof: Let $T$ be any MST containing $F$.

Let $e$ be the lightest edge leaving $S$.

If $e \in T$, then great!

Suppose not.

Let $e = uv$.

$T$ has a $uv$ path.

The path leaves $S$ along some edge $e'$.

$T - e'$ is a spanning tree.
\[ w(T-e') = w(T) \]
\[ \Rightarrow T-e' \text{ is an MST containing } F+e. \]

exchange argument for a greedy algorithm
Prim's algorithm:

Pick any vertex s.
Repeateadly add lightest edge leaving s's component T.

Store vertices outside T in a priority queue, keyed by lightest edge from T to vertex.
Add edges using chained Min.
Check new edges leaving component from new vertex.
- like a graph search!
ArrayList<Edge> prim() {
    PriorityQueue<Vertex> pq = new PriorityQueue<Vertex>();
    List<Edge> mst = new ArrayList<Edge>();

    for each Vertex v {
        v.key = INFINITY;
        pq.insert(v);
        v.incoming = null; // Need to remember cheapest edge to v.
        v.known = false; // Need to track if v is in the main component.
    }

    s.key = 0;
    pq.decreaseKey(s);
    s.known = true;

    while (!pq.isEmpty()) {
        Vertex v = pq.deleteMin();
        mst.add(v.incoming);
        v.known = true;

        for each Vertex w adjacent to v {
            e = edge from v to w;
            cvw = cost of e;
            if (!w.known && cvw < w.key) {
                w.key = cvw;
                pq.decreaseKey(w);
                w.incoming = e;
            }
        }
    }

    return mst;
}

$O(|E| \log |V|)$ with binary heap

$O(|V| \log |V| + |E|)$ with Fibonacci heap
Kruskal's algorithm

Process edges in increasing order of weight. Add e iff it doesn't make a cycle.

(If e makes a cycle, it's useless.)

O.w. e is lightest edge leaving both end point components.

Put all edges in a priority queue & repeatedly delete Min.
ArrayList<Edge> kruskal() {
    numVertices = number of vertices;
    DisjSets ds = new DisjSets(numVertices);
    PriorityQueue<Vertex> pq = new PriorityQueue<Edge>();
    List<Edge> mst = new ArrayList<Edge>();

    for each Edge e {
        pq.insert(e); // Keyed by edge weight.
    }

    while (mst.size() < numVertices - 1) {
        Edge e = pq.deleteMin();
        SetType uset = ds.find(e.getu()); // e = uv
        SetType vset = ds.find(e.getv());

        if (uset != vset) {
            mst.add(e);
            ds.union(uset, vset);
        }
    }
    return mst;
}