Shortest Paths Algorithms
## What to compute?

### Types of Problems
- All Pairs Shortest Paths (APSP): Floyd-Warshall, Johnson’s Algorithm
- Single Source Shortest Paths (SSSP): Bellman-Ford, Dijkstra

### Types of Graphs
- Weighted vs. Unweighted Graphs
- Negative Weights allowed?
- Check for negative cycles necessary?
- Simple graph? Loops?
**Unweighted Graph**

### BFS
- use queue
- store parents
- update dist from parent dist
- or use 2 queues and dist = level
Bellman-Ford

**(incomplete) Bellman**

```cpp
vector<bool> queued (n, false);
vector<int> parent (n,-1);
queue<int> q;
q.push(start);
while (!q.empty()) {
    int curr = q.front(); q.pop();
    // for every neighbour update dist and parent
    // if dist updated and !queued q.push(neighb)
}
```
Bellman-Ford correctness

Correctness

- correctness → trivial
- termination?
- a shortest path has at most $n - 1$ edges
- the head of a shortest path is a shortest path
- after $m$ steps all shortest paths with 1 edge have been found
- induction: after $k \cdot m$ steps all shortest paths with $k$ edges have been found
- → at most $m \cdot n$ steps
- with the above implementation $m \cdot (\text{max number of edges on a shortest path})$
no decrease key for pqueue in c++ (header „queue“)
assume distance type \( T \)

```cpp
typedef pair<T, int> pq_pair;
priority_queue<pq_pair, vector<pq_pair>,
greater<pq_pair> > pq;
```

instead of decreasing put new items with new dist in pq
mark nodes with „known“ dist and don’t process them again
at most \( 2m \leq n^2 \) vertices in the queue
time for pq-operations is thus smaller than
\[ O(\log(n^2)) = O(2 \log n) \]
Floyd-Warshall

explanation in dynamic programming lecture

```c
for (int mid = 0; mid < n; ++mid) {
    for (int i = 0; i < n; ++i) {
        if (i == mid)
            continue;
        for (int j = 0; j < n; ++j) {
            if (j == mid || i == j)
                continue;
            if (f[i*n+mid] != -1 && f[mid*n+j] != -1 &&
                (f[i*n+j] == -1 ||
                f[i*n+mid] + f[mid*n+j] < f[i*n+j])) {
                f[i*n+j] = f[i*n+mid] + f[mid*n+j];
            }
        }
    }
} return f;
```