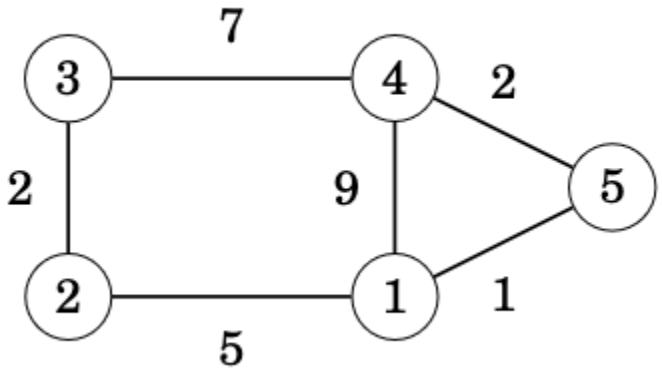
Graph intro

https://cses.fi/book/book.pdf

- Kinds:
  - Directed
  - Undirected
  - DAGs

- Representation:
  - Matrix: `A` where `A[i][j]` is the distance between nodes i and j (0 or INF if no link)
    - Optimization (if number of links is small): keep dict of dicts
  - Adjacency list: `A[i]` gives list of nodes connected to `j`
  - Edge list: list of pairs `(i, j)` where each pair is an edge



	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

Shortest paths: Floyd-Warshall

- **Find shortest path from node i to node j**
- First algo: Floyd-Warshall $O(n^3)$

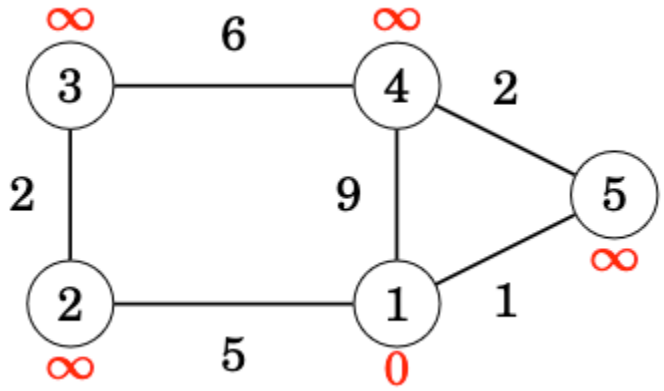
```
def fw(graph):
  dist = copy.deepcopy(graph)
  n = len(graph)
  # Assumes dist[i][j] = 0 for all i
  for k in range(n):
      for i in range(n):
          for j in range(n):
              dist[i][j] = min(
                  dist[i][j],
                  dist[i][k] + dist[k][j]
              )
```

Floyd-Warshall: Arbitrage

- https://www.spoj.com/problems/ARBITRAGE/
- Given list of exchange rates, see if you can somehow execute the list of conversion and end up with more money that you started with
- Same as FW: instead of minimizing distance, maximize current solution or a solution via other currency: `dist[i][j] = max(dist[i][j], dist[i][k]*dist[k][j])`
- `dist[i][i] = 1.0`

Shortest path: Dijkstra

- Shortest paths from a start node to all other nodes



- Key idea: keep a list of sorted best distances so far, and use them to see if we can improve existing distances
- No negative edges: you must use Bellman-Ford for that
  - Also used for negative cycles

```
from heapq import heappush, heappop
def dijkstra(g, start):  # g is adjacency list with weights
    n = len(g)
    dist = [float('inf')] * n  # dist from start node
    dist[start] = 0
    pq = [(0, start)]  # distance of start-start is 0
    while pq:
        d, u = heappop(pq)
        if d > dist[u]:  # no improvements, skip
            continue
        for v, w in g[u]:
            if (nd := d + w) < dist[v]:
```

Dijkstra: Highways

https://www.spoj.com/problems/HIGHWAYS/

```python
import sys
input = sys.stdin.readline
from heapq import heappush, heappop
def dijkstra(g, start, end):  # g is adjacency list with weights
    n = len(g)
    dist = [float('inf')] * n  # dist from start node
    dist[start] = 0
    pq = [(0, start)]  # distance of start-start is 0
    while pq:
        d, u = heappop(pq)
        if d > dist[u]: continue # no improvements, skip
        if u == end: return d # found end
        for v, w in g[u]:
            if (nd := d + w) < dist[v]:
                dist[v] = nd
                heappush(pq, (nd, v))
    return "NONE"

T = int(input())
for _ in range(T):
    n, m, s, t = map(int, input().split())
    g = [[] for _ in range(n + 1)]
    for _ in range(m):
        u, v, w = map(int, input().split())
        # bidirectional graph
        g[u].append((v, w))
        g[v].append((u, w))
    print(dijkstra(g, s, t))
```

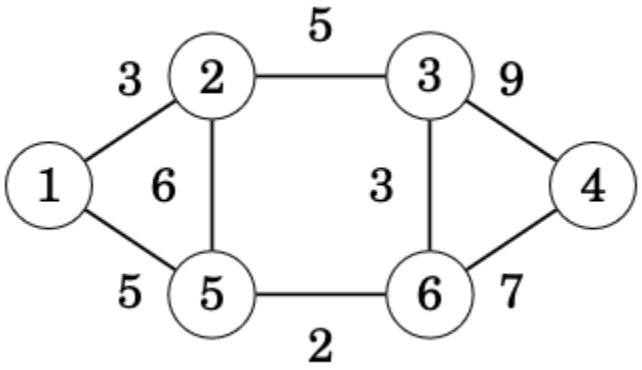
Mice and maze: Dijkstra

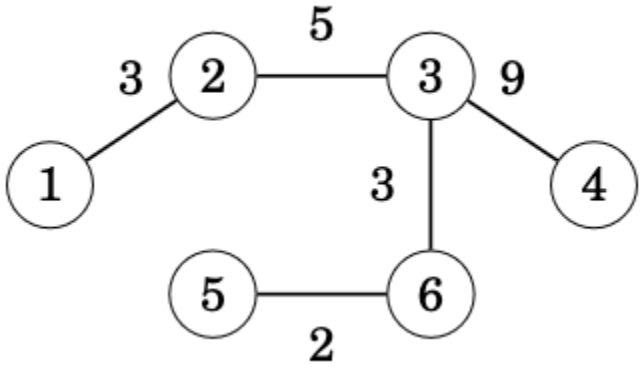
https://www.spoj.com/problems/MICEMAZE/

- How many exits you can reach in a given amount of time?
- Trick:
  - Reverse edges
  - Start from exits with Dijkstra, see how far you can get

MST: Prim

- Minimum spanning tree (MST): a tree that connects whole graph and whose weight is minimal
  - e.g., given a map, min amount of wiring needed to connect all cities in the map to the electric grid





MST: Prim

- Prim's algorithm: same as Dijkstra, BUT just add weight of node, not the sum

```
from heapq import heappush, heappop
def prim(g):
    n = len(g)
    visited = [False] * n
    pq = [(0, 0)]  # (weight, node); start from node 0
    cost = 0
    count = 0
    while pq:
        w, u = heappop(pq)
        if visited[u]: continue
        visited[u] = True
        cost += w
        count += 1
        for v, weight in g[u]:
            if not visited[v]:
                heappush(pq, (weight, v))
    return cost if count == n else -1
```

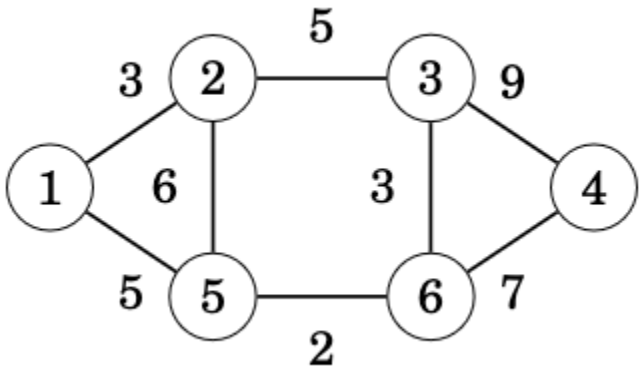
Prim: Island hopping

- https://open.kattis.com/problems/islandhopping
- List of island coordinates, build bridges to connect all islands and minimize cost

```python
import math
from heapq import heappush, heappop

T = int(input())
for _ in range(T):
    n = int(input())
    points = [tuple(map(float, input().split())) for _ in range(n)]
    visited = [False] * n
    pq = [(0, 0)]  # (distance, island)
    ans = 0
    while pq:
        d, u = heappop(pq)
        if visited[u]: continue
        visited[u] = True
        ans += d
        x1, y1 = points[u]
        for v in range(n):  # add a bridge to all unvisited islands
            if not visited[v]:
                x2, y2 = points[v]
                heappush(pq, (math.hypot(x1 - x2, y1 - y2), v))
    print(f"{ans:.6f}") # no need to check if tree was formed
```

MST: Kruskal



- Another greedy approach based on colouring
- Sort edges, add edges but do not create cycles
  - Each node has separate colour
  - Adding edge means merging the colours of the nodes that it connects
  - If colours are already same, do not add
- Optimization: merging is basically union-find

MST: Kruskal

```python
def kruskal(n, edges):  # graph is represented as a list of edges
    # UF structure
    parent = list(range(n))
    size = [1] * n
    def find(x):
        while parent[x] != x:
            parent[x] = parent[parent[x]]
            x = parent[x]
        return x
    def union(a, b):
        a, b = find(a), find(b)
        if a == b: return False
        if size[a] < size[b]:
            a, b = b, a
        parent[b] = a
        size[a] += size[b]
        return True

    edges.sort()  # (weight, u, v)
    cost, count = 0, 0
    for w, u, v in edges:
        if union(u, v):
            cost += w
            count += 1
            if count == n - 1:
                break
    return cost if count == n - 1 else -1
```

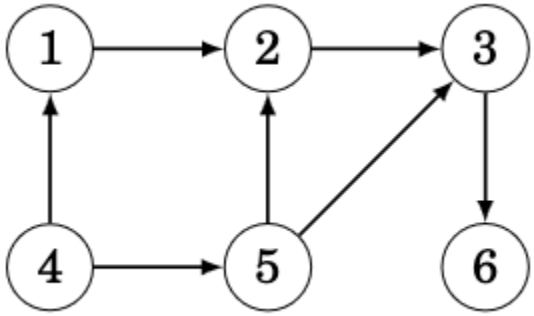
Kruskal: Driving range

- https://open.kattis.com/problems/drivingrange
- Given roads between cities and their distances, find min. range that electric car must have to be able to visit all cities (assuming there is a charger in each city)
- Boils down to finding MST and then max. node in that MST

```
for w, u, v in edges:
  if union(u, v):
      count += 1
      ans = w  # largest selected weight (edges sorted, so last w is the largest)
      if count == n - 1:
          break
```

Topsort

- DAGs: directed graphs, no cycles



- Topological order: if there is a path from node a to node b, then node a appears before node b in the ordering
  - Done via DFS

```
from collections import deque
def topsort(n, edges):  # given list of edges
    g = [[] for _ in range(n)]
    indeg = [0] * n  # number of nodes getting "in" a node
    for u, v in edges:
        g[u].append(v)
        indeg[v] += 1

    q = deque(i for i in range(n) if indeg[i] == 0)
    order = []
    while q:
        u = q.popleft()
        order.append(u)
        for v in g[u]:
            indeg[v] -= 1  # remove it from the graph
            if indeg[v] == 0:
```

Topsort: Build Dependencies

- https://open.kattis.com/problems/builddeps

```
6
gmp:
solution: set map queue
base:
set: base gmp
map: base gmp
queue: base
gmp
```

```python
import sys
from collections import defaultdict

input = sys.stdin.readline
n = int(input())
g = defaultdict(list)
for _ in range(n):
    line = input().replace(":", "").split()
    target = line[0]
    for dep in line[1:]: g[dep].append(target)

changed = input().strip()
visited = set()
ans = []
def dfs(u):
    visited.add(u)
    for v in g[u]:
        if v not in visited:
            dfs(v)
    ans.append(u)  # key trick: add it once everything else is added
dfs(changed)
print(*ans[::-1], sep="\n")
```