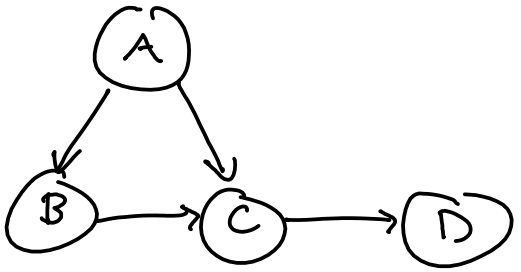
```
BFS_All(g, start):
    q ← new Queue
    q.enqueue(start)
    costs ← new Dictionary
    costs[start] ← 0
    While q not empty:
        v ← q.dequeue()
        For each edge leaving v in g:
            new_cost ← 1 + costs[v]
            If edge.target not in costs OR costs[edge.target] > new_cost:
                costs[edge.target] = new_cost
                q.enqueue(edge.target)
    Return costs
```



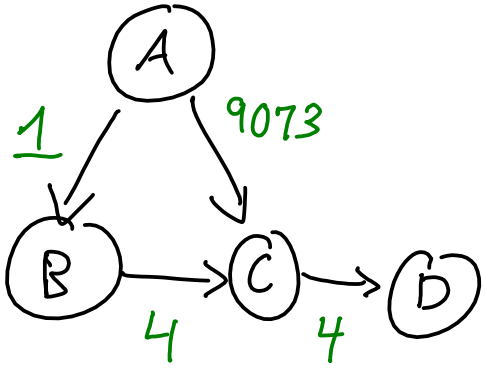
v = D

new_cost = 2

q = [ ]

costs = { A ↦ 0, B ↦ 1, C ↦ 1, D ↦ 2 }

```
Function Dijkstra's(g, start):
    q ← new PrioQueue
    q.enqueue(0, start)
    costs ← new Dictionary
    costs[start] = 0
    While q not empty:
        v ← q.dequeue()
        For each edge leaving v in g:
            new_cost ← edge.weight + costs[v]
            If edge.target not in costs OR costs[edge.target] > new_cost:
                q.enqueue(new_cost, edge.target)
                costs[edge.target] = new_cost
    Return costs
```



q [ ]

costs { A↦0, B↦1, C↦5, D↦9 }

v =

new_cost =

E log E

O(E)

O(E · log E)

O(E)

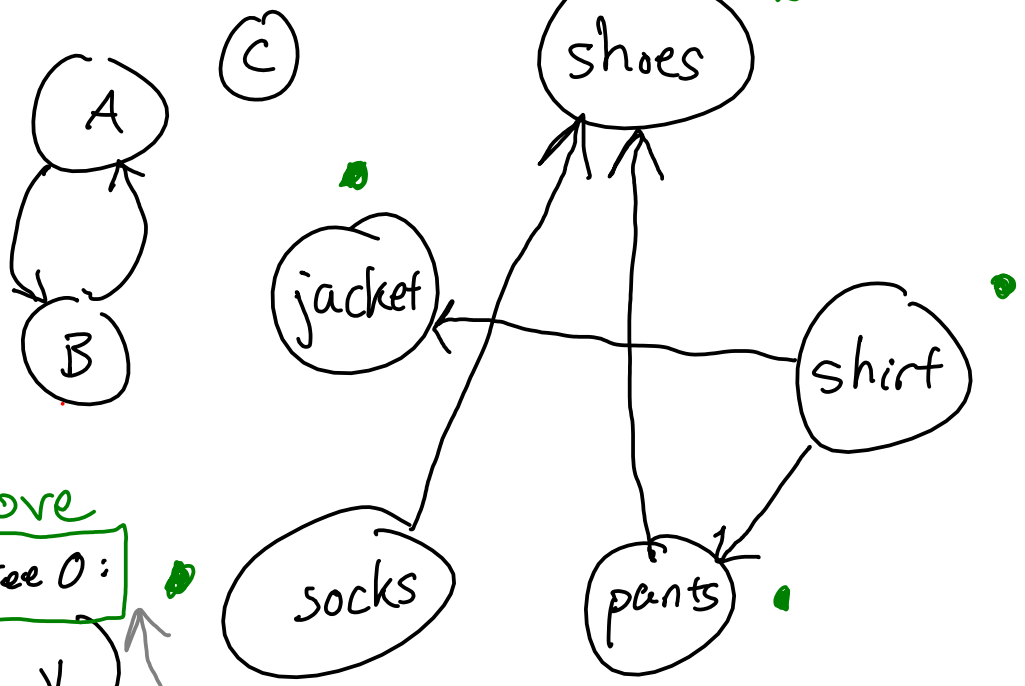
$O(E \cdot (\log E))$

Dependency Graph (sort of DFS)

```
Topological_Sort(g):
    answer ← new List
    visits ← new Dictionary
    For each v in g:
        visits[v] = ●
    For each v in g w/ in-degree 0:
        TSH(g, answer, visits, v)
    Make sure in visits, no ●
Function TSH(g, answer, visits, v):
    If visits[v] == ●: Return
    If visist[v] == ●: !! (exception)
    visits[v] ← ●
    For each neighbor of v:
        TSH(g, answer, visits, neighbor)
    visits[v] ← ●
    answer.insert_at_front(v)
```

remove

either-or



New ■

Active ■

Complete ■

Shirt, Pants, Jacket, Socks, Shoes