Reminder: test 3 in lab today, please don't be late!

TODAY:

- single source shortest path (Dijkstra's algorithm)
- topological sort

Previously:

we used DFS to check if we could go start ⇝ end

① bool reachableDFS(V start, V end, Graph<V,E,W>* g)

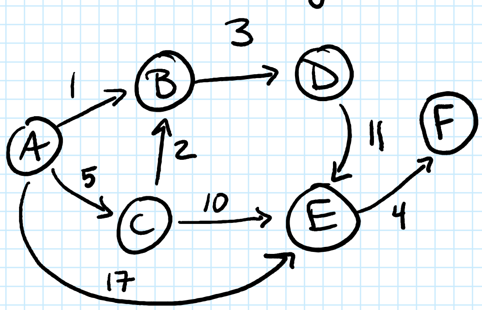
② vector<V> shortestLengthPathBFS(V start, V end, Graph<V,E,W>* g)

//ignores weights and returns the path with fewest number of edges

We used BFS to find the path with fewest edges going start ⇝ end

③ Dictionary<V,W>* singleSourceShortestPath(V start, Graph<V,E,W>* g)

// finds the length of the shortest path from start to every vertex in graph



singleSourceShortestPath("A", graph)

should return:

"A" → 0
"B" → 1
"C" → 5
"D" → 4
"E" → 15
"F" → 19

Q: What sort of data structure might be useful here?

a queue with priorities! in particular, we want a minimum priority queue

Idea: use previously-computed shortest path lengths during computation to avoid duplicate work
This means we need to fill in the dictionary with closer vertices' data before doing further-away vertices.

pseudocode: (Dijkstra's algorithm for single-source shortest paths)

```
Dictionary<V,W>* singleSourceShortestPath(V start, Graph<V,E,W>* g)
    create a Dictionary<V,W> called dist
    create a minimum priority queue<W,V> called pq
```

```
dist.insert(start, Ø)
pq.insert(Ø, start)
while pq is not empty:
    currentPriority = pq.peekPriority()
    currentVertex = pq.remove()
    currentDist = dist.get(currentVertex)
    if currentDist == currentPriority   // make sure we ignore stale priorities that were still in pq
        for every outgoing edge e from currentVertex
            nextV = e's destination
            nextDist = currentDist + e's weight
            if dist doesn't contain nextV:
                dist.insert(nextV, nextDist)
                pq.insert(nextDist, nextV)
            else if nextDist < dist.get(nextV)
                dist.update(nextV, nextDist)
                pq.insert(nextDist, nextV)
return dist
```

case: nextV is a new vertex we haven't seen before

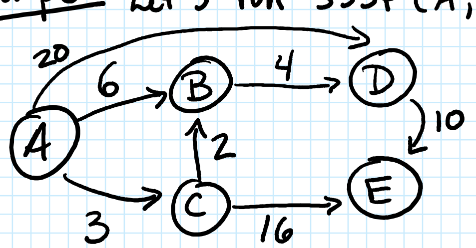
case: we've seen nextV before but we have now found a shorter way to reach it

Dijkstra's algorithm is an example of a greedy algorithm.

It always explores the best possible option first (using a priority queue).
(Compare with BFS, which just explores the next option (using a queue).)

```
Dijkstra's while loop:
while pq is not empty:
    currentPriority = pq.peekPriority
    currentVertex = pq.remove
    currentDist = dist.get(currentVertex)
    if currentDist == currentPriority
        for every outgoing edge e from current Vertex:
            nextV = e's destination
            nextDist = currentDist + e's weight
            if dist doesn't contain nextV
                dist.insert(nextV, nextDist)
                pq.insert(nextDist, nextV)
            else if nextDist < dist.get(nextV)
                dist.update(nextV, nextDist)
                pq.insert(nextDist, nextV)
```

example: Let's run SSSP(A, graph).



Dictionary<V,W> dist

A → 0
B → ~~6~~ 5
C → 3
D → ~~20~~ 9
E → 19

minPQ <W,V> pq

~~0 A~~ 1
~~3 C~~ 2
~~5 B~~ 3
~~6 B~~ 4 ignored
~~9 D~~ 5
~~19 E~~ 6
~~20 D~~ 7 ignored

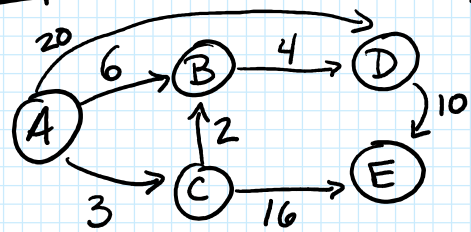
currentVertex: ~~A~~ ~~C~~ ~~B~~ ~~B~~ ~~D~~ ~~E~~ D

currentPriority: ~~0~~ ~~3~~ ~~5~~ ~~6~~ ~~9~~ ~~19~~ 20

currentDist: ~~0~~ ~~3~~ ~~5~~ ~~5~~ ~~9~~ ~~19~~ 9

This dist Dictionary gets returned.

example: Let's run SSSP(C, graph).




```
Dijkstra's while loop:
while pq is not empty:
    currentPriority = pq.peekPriority
    currentVertex = pq.remove
    currentDist = dist.get(currentVertex)
    if currentDist == currentPriority
        for every outgoing edge e from current Vertex:
            nextV = e's destination
            nextDist = currentDist + e's weight
            if dist doesn't contain nextV
                dist.insert(nextV, nextDist)
                pq.insert(nextDist, nextV)
            else if nextDist < dist.get(nextV)
                dist.update(nextV, nextDist)
                pq.insert(nextDist, nextV)
```

Dictionary<V,W> dist

C → 0
B → 2
E → 16
D → 6

This dist Dictionary gets returned.

minPQ<W,V> pq

~~0 C~~ 1
~~2 B~~ 2
~~6 D~~ 3
~~16 E~~ 4

currentVertex: ~~C~~ ~~B~~ ~~D~~ E
currentPriority: ~~0~~ ~~2~~ ~~6~~ 16
currentDist: ~~0~~ ~~2~~ ~~6~~ 16

Runtime analysis: For a graph with $n = |V|$ vertices and $m = |E|$ edges, Dijkstra's algorithm for single-source shortest paths takes $O((m+n)\log_2(n))$ time in the worst case (and needs to use good implementations of Dictionary and priority queue). Optimizations are possible!

Runtime analysis for reachableDFS: $O(n+m)$

(basically just the runtime of DFS)

Runtime analysis for shortestLengthPathBFS: $O(\overbrace{n+m}^{BFS} + n) = O(n+m)$

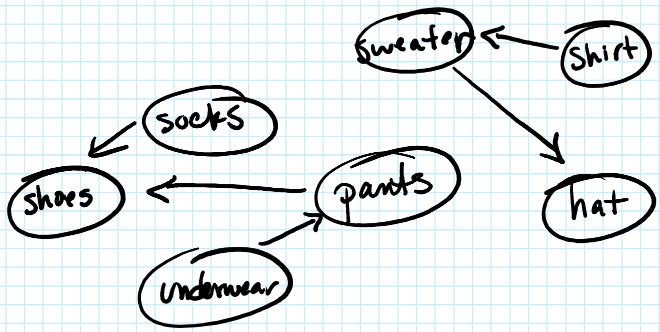
reconstructing the path

(basically just BFS and kept track of previous to reconstruct the path)

TOPOLOGICAL SORT

Consider a graph which represents tasks which must be completed in a particular order.

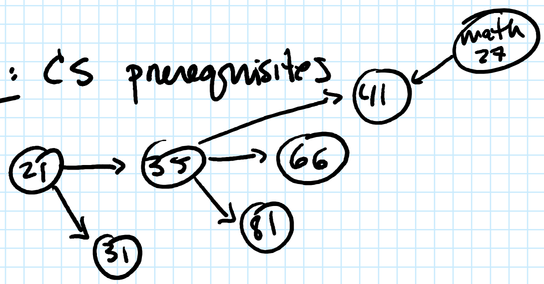
example: getting dressed



working ordering:

shirt	underwear
sweater	shirt
hat	pants
underwear	sweater
socks	socks
pants	hat
shoes	shoes

example: CS prerequisites



Goal: find a valid ordering so that the "X before Y" requirement is always satisfied.

Observation: we need to start the ordering with a vertex with in-degree 0.

Q: What data structures would be useful for this? Want to track current in-degree of all vertices when it hits 0, we're allowed to use it.

pseudocode for topological sort

```
initialize a dictionary to store vertices with their in-degree
push all vertices with in-degree 0 onto stack
while stack is not empty
    pop from stack
    add that vertex to the linear ordering we're building
    for all outgoing edges:
        decrement the in-degree of the destination vertex
            and update it in the dictionary
        if the neighbor's new in-degree is now zero
            push the neighbor onto the stack
return the linear ordering
```