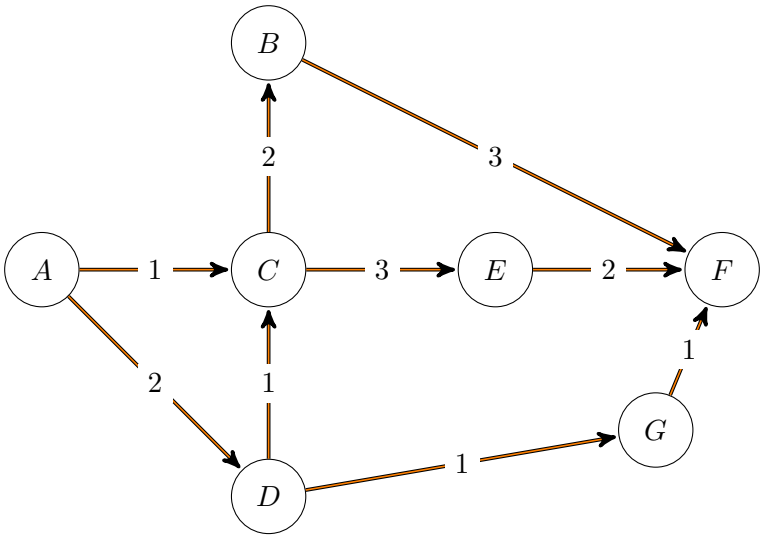
1 Dijkstra's Algorithm

- Dijkstra's algorithm solves the *single-source shortest path problem*: given a graph $G = (V, E, \omega)$ with nonnegative weights, determine the shortest path from a *source vertex* $s \in V$ to every $v \in V$.

```
DIJKSTRA(G(V, E, ω), s ∈ V)
(1)   foreach v ∈ V
(2)       DIST[v] ← ∞, PREV[v] ← NIL
(3)   DIST[s] ← 0
(4)   H ← {(s, 0)}
(5)   while H ≠ ∅
(6)       v ← DELETEMIN(H)
(7)       foreach (v, w) ∈ E
(8)           if DIST[w] > DIST[v] + ω(v, w)
(9)               DIST[w] ← DIST[v] + ω(v, w), PREV[w] ← v, INSERT((w, DIST[w]), H)
```

- The running time of Dijkstra's algorithm depends on the implementation of the heap H. For each vertex, we perform a delete min, while for each edge we perform an insertion. Therefore, the run-time of Dijkstra's is: $O(n \cdot \text{deleteMin} + m \cdot \text{insert})$
- Note that Dijkstra's algorithm does not work for negative edge weights! When a node is popped off the heap, we assume that the distance for that node is completely set. Since all other nodes in the heap have a larger distance than the one just popped off, we know that any path through those nodes cannot be

Exercise. *Run Dijkstra on this graph from A to F.*



Solution.

Looking at	Heap
A	C(1) D(2)
C	D(2) B(3) E(4)
D	B(3) G(3) E(4)
B	G(3) E(4) F(6)
G	E(4) F(4)
F	

Exercise (CLRS 24.3-4)**.** *We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.*

Solution.
The reliability of any path is the product of the probabilities of not failure over each segment of the path. Therefore, this problem is very similar to the shortest path problem except we are trying to maximize the product instead of minimizing the sum. There are two ways we can approach this problem:

1. Modify the probabilities
For each edge $(u, v) \in E$, we replace $r(u, v)$ with $-\log r(u, v)$. If $r(u, v) = 0$, then let $\log r(u, v) = \infty$. The resulting numbers will be between 0 and ∞. This works because by taking the log of a product, we convert multiplication to addition. Then, taking the negative of that quantity makes minimizing the sum of the logs equivalent to maximizing the product of the r's. We have that

$$\max\left(\prod r(u,v)\right) = \min\left(-\log\prod r(u,v)\right) = \min\left(-\sum \log r(u,v)\right)$$

where each sum or product is taken over $(u, v) \in$ path. Therefore, by minimizing the right hand side, we maximize the left side. We can perform a standard Dijkstra's Algorithm after modifying the weights in this manner because the $-\log r$ (where $0 \leq r \leq 1$) is always non-negative.

2. Modify the algorithm

- We can replace the min-heap in Dijkstra's Algorithm with a max-heap because now we are trying to maximize some quantity.
- Instead of checking $\text{dist}[w] > \text{dist}[v] + w(v, w)$ for some edge (v, w), we replace the $+$ with a $\times$ and the $<$ with a $>$. The resulting expression is: $\text{dist}[w] < \text{dist}[v] \cdot w(v, w)$. We update the distance of some node w a larger value can be achieved by $\text{dist}[v] \cdot w(v, w)$. In other words, going through v to get to w gets us a larger total distance.

Exercise (2014 Problem Set 2)**.** *Consider the shortest paths problem in the special case where all edge costs are non-negative integers. Describe a modification of Dijkstra's algorithm that works in time $O(|E|+|V|\cdot L)$, where L is the maximum cost of any edge in the graph.*

Solution.
We will modify the underlying data structure of Dijkstra's to perform better with the given restrictions. Consider a new data structure consisting of an array of linked lists. The length of the array is $|V| \cdot L + 1$. All elements in the linked list coming out of index i of the array have a key of i in this data structure. We will also let the last index of the array hold a linked list of vertices with key ∞. We start with just the source vertex at a linked list at index 0.

This is a valid way to store vertices because the key of any vertex will never be larger than $|V| \cdot L$. The longest non-cycling path in the graph can be at most $|V| - 1$ edges, and each edge is at most L long.

Here's how the **insert/changeKey(v)** and **deleteMin()** functinos will work:

- **insert/changeKey(v)**: We use the pointer to v to remove its current node (if it exists) and then we compute the key of v and add this vertex to the linked link at the array indexed by the new key.
- **deleteMin()**: we keep a global variable which starts out at 0. Every time we want to delete the minimum keyed vertex, we go to the linked list indexed by this global variable, and pop off the first element in the linked list. If there is no element there, then we increment this global variable until we get to an index in the array that doesn't contain a null pointer.

In order to prove the correctness of this data structure, we need to show the following claim:

Claim: After a deleteMin() occurs, any vertices inserted into this data structure will have a key greater than that of the element just popped off. This is important because we need to make sure we can keep on incrementing our indexing variable in deleteMin() without worrying about elements appearing to the left of it.

Proof: Whenever we insert a vertex v into our array of linked lists after deleting u, it means that we have found a shorter path from the source to v by going through u. The key of v is obtained by taking the key of u and adding the length of (u, v) so the key of v must be at least as big as that of u.

Run-Time Analysis:

Each insert/changeKey(v) takes $O(1)$ time. This operation is called at most $|E|$ times. The deleteMin() operation takes $O(|V| \cdot L)$ time *overall* because the indexing variable in deleteMin() can only increase by at most $|V| \cdot L$ times, the length of the entire array. Therefore, the overall run-time is $O(E + |V| \cdot L)$.

2 Bellman-Ford Algorithm

Basic Properties:

- Finds the shortest path from a given source vertex to every other vertex in a graph with arbitrary length edges (negative is okay).
- Can be used to detect negative cycles in a graph.
- Can be implemented using recursion and memoization or iteration and dynamic programming.

In the following recursive function, let $f(u, k)$ represent the length of the shortest path from the source s to u using at most k edges. Then, we have the following recursion:

$$f(u,k) = \begin{cases} 0 & : u = s \text{ and } k = 0 \\ \infty & : u \neq s \text{ and } k = 0 \\ \min\{f(u,k-1), \min_{v \in V,(v,u) \in E} f(v,k-1) + w(v,u)\} & \text{otherwise} \end{cases}$$

Our goal is to compute $f(u,n)$ for all $u \in V$. If there are no negative cycles, then $f(u,n-1)$ should be equal to $f(u,n)$ because the latter implies that a cycle was used. We can do this in two ways:

- **Recursion and Memoization:** While we are computing $f(u,n)$ for each vertex u, anytime we need to compute $f(v,k)$ for some vertex v and integer k, first check a global lookup table to see if that value has been calculated before. If so, then take the value from the table. If not, then compute the value recursively and store it in the lookup table.
- **Iteration and Dynamic Programming:** Building up our solution from the base case upwards. We compute $f(u,0)$ for all u, and then proceed to calculating $f(u,1)$ for all u...etc. This works because each $f(u,k)$ only depends on the values of $f(v,k-1)$ for some set of v's which we can assume were already filled up during the previous iteration.

Exercise. *Explain why the recursion above is correct. Then, analyze the run-time of Bellman-Ford.*

Solution.
Explanation of Recursion: The shortest path from s to u using at most k edges could have potentially used at most $k-1$ edges. In addition, the shortest path from s to u using at most k edges must have reached u from one of its neighbors. Therefore, we loop over all v such that $(v,u) \in E$, looking at the shortest path to each v in at most $k-1$ edges. The minimum of these two conditions represents the optimal location a path could have been one step earlier before arriving at u.

Run-Time Analysis: Fix a particular value of k. We want to find $f(u,k)$ for each $u \in V$. Using either memoization or dynamic programming, we already have access to $f(u,k-1)$ for all $u \in V$ in constant time. Therefore, we just need to figure out how much time it takes to take the minimum. Let $u_1, u_2 \ldots u_{|V|}$ represent the $|V|$ vertices in the graph. For each vertex, we perform $O(1)$ work to compute the minimum of 2 elements (the outer min) and then $O(\text{indegree}(u_i))$ work to compute the minimum of that many items. Therefore, the total run-time is:

$$O(1) + O(\text{indegree}(u_1)) + O(1) + O(\text{indegree}(u_2)) + \ldots + O(1) + O(\text{indegree}(u_{|V|})) = O(|V| + |E|)$$

because the sum of the indegrees is $O(|E|)$. This computation was for a fixed value of k, and there are n possible values of k, so we get $O(|V|(|V| + |E|)$ or $O(n(n+m))$ as the overall run-time.

Exercise (2014 Problem Set 3)**.** *The* risk-free currency exchange problem *offers a risk-free way to make money. Suppose we have currencies $c_1, \ldots, c_n$. (For example, c_1 might be dollars, c_2 rubles, c_3 yen, etc.) For every two currencies c_i and c_j there is an exchange rate $r_{i,j}$ such that you can exchange one unit of c_i for $r_{i,j}$ units of c_j. Note that if $r_{i,j} \cdot r_{j,i} > 1$, then you can make money simply by trading units of currency i into units of currency j and back again. This almost never happens, but occasionally (because the updates for exchange rates do not happen quickly enough) for very short periods of time exchange traders can find a sequence of trades that can make risk-free money. That is, if there is a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$, then trading one unit of c_{i_1} into c_{i_2} and trading that into c_{i_3} and so on will yield a profit.*

Design an efficient algorithm to detect if a risk-free currency exchange exists. (You need not actually find

it.)

Solution.
We will use Bellman Ford to detect negative cycles. First, we can do something similar to the first Exercise on Dijkstra's where we either take the log of all the currency exchange rates. Using the method, we can again take the negative log of each exchange rate because: $r_{ij}r_{ji} > 1 \Leftrightarrow -\log r_{ij}r_{ji} < 0 \Leftrightarrow = (-\log r_{ij}) + (-\log r_{ji}) < 0$. This can of course be extended to more than 2 rates. Now, all we need to do is detect for negative cycles. We can start Bellman Ford at any vertex because we have a complete graph, where every two vertices are connected by an edge. The run-time of this algorithm is of course the same as that of Bellman Ford: $O(n(n+m))$.