

🌀 Homework 8 🌀

Due Tuesday, October 27, 2026 at 9pm Central Time

Please review and understand the homework policies, especially [our expectations about describing algorithms](#), and both the [general and more specific grading rubrics for graph algorithms](#).

You may use any algorithm or data structure seen in class **or a prerequisite class such as data structures** as a black box without having to explain how it works.

1. You're visiting a friend in a new city with three overlapping tram systems AirTram, BestTram, and CarryTram (or Systems A, B, and C for short) that share the same stations but offer different connections between them. All three systems charge money to use their connections but offer a discount to card-carrying members of that system.

Your friend had offered to hook you up with, “a membership card for all three tram systems,” but when you first arrived at his home, he revealed that he really had a single card for System C with two stickers stacked on top, each designed to make it *look* like a card for the one of the other systems. The effects of this scheme can be summarized as follows:

- As long as you don't remove either sticker, you'll appear to be a member of System A.
- Once you remove the top sticker, you'll appear to be a member of system B.
- Once you remove both stickers, you'll appear to be a member of System C.
- You'll pay **half** the cost of using a connection run by the system you currently appear to be a member of.

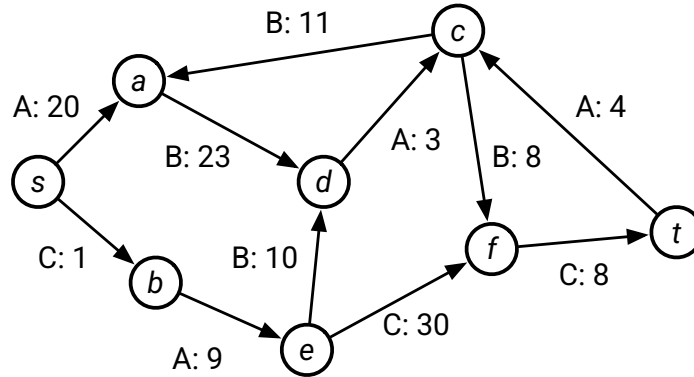
(And no, you cannot remove the bottom sticker before the top one.)

The stickers were produced with cheap adhesive, so neither of them can be reapplied once they are removed. However, nobody at the tram stations will be watching you very closely, so you can remove one or both stickers at any stop including the one at your friend's house. **Your goal is to get from your friend's house to the Transit Authority as cheaply as possible**, so you can buy yourself some *proper* membership cards.

Formally, you are given a directed graph $G = (V, E)$ whose vertices represent tram stations and whose edges represent direct connections between pairs of stations. Every edge e is given a label $system(e) \in \{A, B, C\}$ saying whether that connection is run by System A, System B, or System C, respectively, along with a positive cost $\$(e)$ for **non-members** to use that connection (again, you'll pay half that cost if you appear to be a member of the connection's system.)

Describe an algorithm to compute the minimum cost to get from your friend's house at vertex s to the Transit Authority at vertex t . For example, given the graph below with each edge labeled by its system and cost, your algorithm should return 21.5 for

the plan of traveling along $s \rightarrow b \rightarrow e$ while appearing to be a member of System A at cost $1 + 9/2 = 5.5$, removing the top sticker, traveling along $e \rightarrow d \rightarrow c \rightarrow f$ while appearing to be a member of System B at cost $10/2 + 3 + 8/2 = 12$, removing the second sticker, and traveling along $f \rightarrow t$ while appearing to be a member of System C at cost $8/2 = 4$.



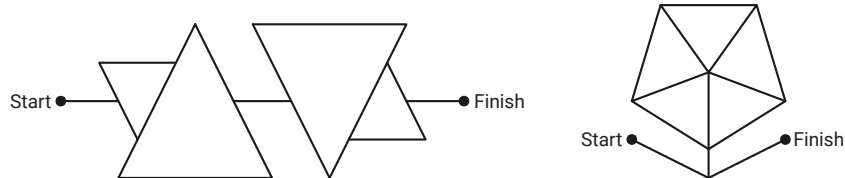
Edit: You may use the following fact without proof: Given a directed graph $G = (V, E)$ with non-negative real valued edge weights $w : E \rightarrow \mathbb{R}^+$ and a vertex s , Dijkstra's algorithm computes for each vertex $v \in V$ the value $\text{dist}(v)$, the distance from s to v . The algorithm runs in $O(|E| \log |V|)$ time.

2. Let $G = (V, E)$ be a directed graph, $\Sigma = \{0, 1, \dots, 9\}$ be the alphabet of decimal digits, and $\ell : V \rightarrow \Sigma$ be an assignment of digit labels to each vertex.
 - (a) Given a walk $W = \langle v_0, v_1, \dots, v_{k-1} \rangle$, let $\sigma(W) = \ell(v_1)\ell(v_2)\dots\ell(v_{k-1})$ be the **string induced by W** . If W has no edges (i.e. $k = 1$) then $\sigma(W) = \varepsilon$.
Describe an algorithm that, given the directed graph $G = (V, E)$, the labeling function $\ell : V \rightarrow \Sigma$, and two vertices $s, t \in V$, returns whether there is an (s, t) -walk W such that $\sigma(W)$ contains the substring **374**.
 - (b) The set of strings containing the substring **374** is regular. Can we generalize the previous problem accordingly?
Describe an algorithm that, given the directed graph $G = (V, E)$, the labeling function $\ell : V \rightarrow \Sigma$, and two vertices $s, t \in V$, and a DFA $M = (Q, \delta, s', A)$ over Σ returns whether there is an (s, t) -walk W such that $\sigma(W) \in L(M)$. The running time of your algorithm should be expressed in terms of $|V|$, $|E|$, and $|Q|$.

3. Practice only. Do not submit solutions.

One of Jeff Erickson's daughter's elementary-school math workbooks¹ contains several puzzles of the following type:

Complete each angle maze below by tracing a path from Start to Finish that has only acute angles.



Describe and analyze an algorithm to solve arbitrary acute-angle mazes.

Your input is a connected undirected graph G , whose vertices are points in the plane and whose edges are straight line segments. Edges do not intersect, except at their common endpoints. For example, a drawing of the letter X would have five vertices and four edges, and the first maze above has 18 vertices and 21 edges. You are also given two vertices Start and Finish.

Your algorithm should return TRUE if G contains a walk from Start to Finish that has only acute angles, and FALSE otherwise. Formally, a walk through G is valid if, for any two consecutive edges $\langle(u, v), (v, w)\rangle$ in the walk, either $\angle uvw = \pi$ (straight) or $0 < \angle uvw < \pi/2$ (acute). Assume you have a subroutine that can determine in $O(1)$ time whether the angle between two given segments is straight, obtuse, right, or acute.

¹Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012. See <https://www.beastacademy.com/resources/printables.php> for several more examples.

Solved problem

4. Professor McClane takes you out to a lake and hands you three empty jars. Each jar holds a positive integer number of gallons; the capacities of the three jars may or may not be different. The professor then demands that you put exactly k gallons of water into one of the jars (which one doesn't matter), for some integer k , using only the following operations:
- Fill a jar with water from the lake until the jar is full.
 - Empty a jar of water by pouring water into the lake.
 - Pour water from one jar to another, until either the first jar is empty or the second jar is full, whichever happens first.

For example, suppose your jars hold 6, 10, and 15 gallons. Then you can put 13 gallons of water into the third jar in six steps:

- Fill the third jar from the lake.
- Fill the first jar from the third jar. (Now the third jar holds 9 gallons.)
- Empty the first jar into the lake.
- Fill the second jar from the lake.
- Fill the first jar from the second jar. (Now the second jar holds 4 gallons.)
- Empty the second jar into the third jar.

Describe and analyze an efficient algorithm that either finds the smallest number of operations that leave exactly k gallons in any jar, or reports correctly that obtaining exactly k gallons of water is impossible. Your input consists of the capacities of the three jars and the positive integer k . For example, given the four numbers 6, 10, 15, and 13 as input, your algorithm should return the number 6 (the length of the sequence of operations listed above).

Solution: Let A, B, C denote the capacities of the three jars. We reduce the problem to breadth-first search in a directed graph $G = (V, E)$ defined as follows:

- $V = \{(a, b, c) \mid 0 \leq a \leq A \text{ and } 0 \leq b \leq B \text{ and } 0 \leq c \leq C\}$. Each vertex corresponds to a possible **configuration** of water in the three jars. There are $(A+1)(B+1)(C+1) = O(ABC)$ vertices altogether.
- G contains a directed edge $((a, b, c), (a', b', c'))$ whenever it is possible to change the first configuration into the second in one step. Specifically, G contains an edge from (a, b, c) to each of the following vertices (except those already equal to (a, b, c)):
 - $(0, b, c)$ and $(a, 0, c)$ and $(a, b, 0)$ — dumping a jar into the lake
 - (A, b, c) and (a, B, c) and (a, b, C) — filling a jar from the lake
 - $\begin{cases} (0, a+b, c) & \text{if } a+b \leq B \\ (a+b-B, B, c) & \text{if } a+b \geq B \end{cases}$ — pouring from jar 1 into jar 2
 - $\begin{cases} (0, b, a+c) & \text{if } a+c \leq C \\ (a+c-C, b, C) & \text{if } a+c \geq C \end{cases}$ — pouring from jar 1 into jar 3

$$\begin{aligned}
& - \left\{ \begin{array}{ll} (a+b, 0, c) & \text{if } a+b \leq A \\ (A, a+b-A, c) & \text{if } a+b \geq A \end{array} \right\} \text{ — pouring from jar 2 into jar 1} \\
& - \left\{ \begin{array}{ll} (a, 0, b+c) & \text{if } b+c \leq C \\ (a, b+c-C, C) & \text{if } b+c \geq C \end{array} \right\} \text{ — pouring from jar 2 into jar 3} \\
& - \left\{ \begin{array}{ll} (a+c, b, 0) & \text{if } a+c \leq A \\ (A, b, a+c-A) & \text{if } a+c \geq A \end{array} \right\} \text{ — pouring from jar 3 into jar 1} \\
& - \left\{ \begin{array}{ll} (a, b+c, 0) & \text{if } b+c \leq B \\ (a, B, b+c-B) & \text{if } b+c \geq B \end{array} \right\} \text{ — pouring from jar 3 into jar 2}
\end{aligned}$$

Because each vertex has at most 12 outgoing edges, there are at most $12(A+1) \times (B+1)(C+1) = O(ABC)$ edges altogether.

To solve the jars problem, we need to find the **shortest path** in G from the start vertex $(0, 0, 0)$ to any target vertex of the form $(k, \cdot, \cdot)$ or $(\cdot, k, \cdot)$ or $(\cdot, \cdot, k)$.

We can compute this shortest path by calling **breadth-first search** starting at $(0, 0, 0)$, and then examining every target vertex by brute force. If BFS does not visit any target vertex, we report that no legal sequence of moves exists. Otherwise, we find the target vertex closest to $(0, 0, 0)$ and trace its parent pointers back to $(0, 0, 0)$ to determine the shortest sequence of moves. The resulting algorithm runs in $O(|V| + |E|) = O(ABC)$ time.

We can speed up this algorithm by observing that every move leaves at least one jar either completely empty or completely full. Thus, we only need vertices (a, b, c) where either $a = 0$ or $b = 0$ or $c = 0$ or $a = A$ or $b = B$ or $c = C$; no other vertices are reachable from $(0, 0, 0)$. The number of non-redundant vertices and edges is $O(AB + BC + AC)$. Thus, if we only construct and search the relevant portion of G , the algorithm runs in $O(AB + BC + AC)$ time. ■

Rubric: 10 points: standard graph reduction rubric

- Brute force construction is fine.
- 1 for calling Dijkstra instead of BFS
- max 8 points for $O(ABC)$ time; scale partial credit.