Quiz 2 Practice

Problem 1. Trip Planning with a Gas Tank

You want to drive from some location $s$ to some location $t$ without running out of gas while spending as little money as possible. You are given a map of the road system designated by a set of locations $V$ and a set of directed roads $E$, where $(u, v) \in E$ means that there is a road going from $u \in V$ to $v \in V$. You know $w(u, v)$, the length of the road from $u$ to $v$. You are also told which locations $S \subseteq V$ have gas stations. When you come to a node $u$ with a gas station, you have the option to fill up the tank of your car by paying $c(u)$ dollars (the gas is free - you only pay for the parking).

Assuming you start with a full tank containing $k$ units of gas, give an efficient algorithm that finds a cheapest route from $s$ to $t$ such that you never run out of gas.

Note: it is okay to run out of gas at the same moment you reach a gas station.

Solution:

Here is the best solution to the problem. A discussion of other solutions is given below. It is worth noting here that a greedy approach does not work because of the nature of the constraints.

Algorithm:

The fastest way we know to solve this problem is to build a new graph over the gas stations and run a standard shortest-path algorithm on that graph.

Specifically, we want to create a new graph $G'$ whose vertices are the locations with gas stations plus $s$ and $t$, i.e. $V' = S \cup \{s, t\}$. Let $\delta(u, v)$ be the length of the shortest path between $u$ and $v$ in $G$. Now, an edge between $u$ and $v$ exists in $G'$ if and only if $\delta(u, v) < k$, i.e. we can get from $u$ to $v$ on one tank of gas. Finally, the edge weights $w'(u, v)$ in $G'$ are the cost of filling up at $v$, i.e. $w'(u, v) = c(v)$.

Our algorithm will be as follows:

2. Find the shortest path in $G'$.
3. Recover the path in $G$ that is associated with the path found in $G'$.

There are many ways to do this with varying runtimes. The easiest solution is to use Floyd-Warshall to find the all-pairs shortest paths in $G$. Then, we can run Bellman-Ford on $G'$. Once we
have run Bellman-Ford, we can look back at the results from Floyd-Warshall to determine which partial paths corresponded to the path found in $G'$.

Floyd-Warshall requires $O(V^3)$ time and $O(V^2)$ space. Bellman-Ford will require $O(VE)$ time and certainly $O(V)$ space plus the space for $G'$. This will give an overall running time of $O(V^3)$ and space requirement of $O(V^2)$.

We can speed this up by recognizing that neither distances nor costs are normally negative. Thus, we can reasonably assume that $w(u,v) \geq 0$ and $c(u) \geq 0$ for all $u, v$. As a result, we can find all-pairs shortest paths by running Dijkstra’s algorithm $|V|$ times (essentially Johnson’s algorithm without reweighting.) We can make this faster by observing that we only really need to run Dijkstra’s algorithm starting at each of the gas stations. Thus, the time to generate $G'$ is the time to run $|S|$ versions of Dijkstra’s algorithm, which is $O(S \cdot (E + V \log V))$ using the Fibonacci heap implementation. Finally, we can use Dijkstra’s algorithm again to find the shortest path in $G'$ in $O(S^2 + S \log S) = O(S^2)$ time. The running time will be dominated by the computation of $G$ for a total of $O(S \cdot E + SV \log V)$.

The space requirement when using Dijkstra’s algorithm is a little more complicated. Dijkstra’s algorithm inherently requires $O(V)$ space. However, we need to somehow recover the path in G. The easiest way to do this is to store the results from all $|S|$ original calls to Dijkstra’s algorithm. This will require $O(S \cdot V)$ space. This is probalby the most practical approach from a time perspective, but you can actually reduce the space requirement to $O(S^2 + V)$ with some tricks. Instead of saving the results from Dijkstra’s algorithm, we discard the information not necessary to generate $G'$. The graph $G'$ itself can easily be stored in $O(S^2)$ space, so producing $G'$ requires $O(S^2 + V)$. (Note that it is possible that $|E'| > E$, so the best bound we can give on $|E'|$ is $O(S^2)$.) Now, to recover the path in G, we run Dijkstra’s algorithm again to find the information we discarded. We will rerun Dijkstra’s algorithm at most $|S|$ times, so the asymptotic running time is not affected. Finally, the extra information we keep corresponds to a path. Consequently, it contains at most $|V|$ nodes. Thus, we will need at most $O(V)$ space for the extra information we find. This gives an overall space requirement of $O(S^2 + V)$ (plus the space requirement for the input graph $G$).

Correctness:

To see that this indeed correct, first observe that any path in $G'$ corresponds directly to a path in $G$, since each edge itself corresponds to a path in $G$. Thus, if the cheapest route in $G'$ is not a cheapest path, it must be more expensive than the cheapest route in $G$.

Now, let $P$ be the cheapest route in $G$, and decompose it into subsections between gas stations at which we fill up the tank. Clearly, each subsection corresponds to a distance that can be traversed on one tank of gas, which implies that the section generates an edge in $G'$. It follows that each subsection of $P$ has an edge in $G'$ and therefore the entire path must have a counterpart in $G'$. Moreover, observe that the cost of a path in $G'$ is exactly the gas stations along the route, since you pay $c(u)$ on any incoming edge to $u$ but never on an outgoing edge. Thus, each route in $G$ corresponds to a route of equivalent cost in $G'$.

Therefore, the cheapest route in $G'$ must also correspond to a route in $G'$, meaning that there is a path through $G'$ with optimal cost.
Comments and Other Ideas:
Many people tried to copy each gas station so that there was one copy where the person filled up their gas tank and one where they didn’t. (Either that, or they split a path when it reached a gas station.) This is handled in the above algorithm by allowing edges in $G'$ to represent partial paths that go through gas stations.

There were also a handful of people who attempted to remove (one-by-one) all the vertices that were not gas stations. This is a valid idea, but the approaches presented did not have good analysis of the runtime necessary to perform this operation.

The insidiously tantalizing (yet wrong) approach to this problem is to modify Dijkstra’s algorithm such that it ranks first by cost then by gas amount. The modification doesn’t change the runtime or other properties of Dijkstra’s, giving a possible $O(E + V \log V)$ algorithm. Unfortunately, this doesn’t work. Consider the following graph of 5 nodes (with one gas station at $v$):

![Graph Image]

The greedy choice will pick the path to $w$ avoiding the gas station at $v$, but won’t have enough gas to get to $t$. Thus, the above algorithm won’t work.

Another interesting approach useds the correct algorithm but replace Dijkstra’s with a modification on BFS. The result is a slightly faster running time; however, it fails in a manner similar to the greedy approach. When you decide that a node is not reachable in this problem, you need to make sure you checked the shortest path to that node. This is not possible with BFS.

There was one significant alternative to constructing $G'$. If you assume that all distances are integers, then you can maintain a data structure at each node with $k$ entries and change the notion of “visited” in Dijkstra’s algorithm. (Alternatively, you can make $k$ copies of each vertex to get a new graph with the same idea.) Now, you extend a path as long as the amount of gas in the tank is above the maximum already attained at that node. Then, since weights are integers, there are only $k$ possible amounts of gas you can have in the tank, so you can go through each node at most $k$ times. (Note that without the integer assumption, you may have $2^{|E|}$ possible distinct amounts of gas, so this approach doesn’t work anymore.) Apart from the assumption that $k$ is an integer, the only problem with this algorithm is that $k$ may be very large, resulting in high runtimes.

Problem 2. Karp and Rabin compute the median
Karp and Rabin each hold a set of $n$ numbers, denoted by $K$ and $R$, respectively. The sets $K$ and $R$ are disjoint. They would like to compute the median of the union set $K \cup R$. However, they live on different coasts, and they can communicate only by exchanging simple messages. Each message can either contain a number from their sets, or an integer in the range $[1, n]$. 
Give an efficient communication protocol that Karp and Rabin can use to compute the median of the set \( K \cup R \). The efficiency of the protocol is measured by the number of messages exchanged.

**Solution:** The most efficient solution to this problem uses \( O(\log n) \) messages in the worst case. There are many ways to achieve this. However, all efficient solutions have some “binary search flavor”. We note that a solution using \( O(n) \) messages can be achieved by transferring the whole data set to one of the parties and performing the computation locally.

Define \( \text{rank}_S(a) \) to be the number of elements in \( S \) that are smaller than \( a \). Note that according to this definition, the smallest element has rank 0. The first observation is that, for any element \( a \in K \), it is possible for Karp and Rabin to determine \( \text{rank}_{K \cup R}(a) \) using only constant number of messages. This is done as follows: Karp sends \( a \) to Rabin; Karp (or Rabin, respectively) compute the rank \( r_K(a) \) of \( a \) in \( K \) (or \( r_R(a) \) of \( a \) in \( R \), respectively) using the local information; Karp and Rabin exchange the ranks and set the rank of \( a \) in \( K \cup R \) to be \( r_K(a) + r_R(a) \).

According to the definition in CLRS, p. 183, the median of \( K \cup R \) is the element \( a \) with rank \( n - 1 \) (but we accepted solutions with more exotic median definitions). Assume for the time being that the median is contained in \( K \). Since the function \( \text{rank}_{K \cup R}(a) \) is monotone in the argument \( a \) (i.e., for any \( a, a' \in K \cup R \), if \( a < a' \) then \( \text{rank}_{K \cup R}(a) < \text{rank}_{K \cup R}(a') \)), we can find an element with rank \( n - 1 \) using binary search on elements of \( K \). We first convert the set \( K \) into a sorted array \( \text{SoK}[1 \ldots n] \). Then we perform binary search on the array \( \text{SoK} \) with respect to the ranks. Conceptually, we binary search for the value \( n - 1 \) in the sorted array \( \text{rank}_{K \cup R}(\text{SoK}[1]), \ldots, \text{rank}_{K \cup R}(\text{SoK}[n]) \). Each time we need to compare \( \text{rank}_{K \cup R}(\text{SoK}[i]) \) to \( n - 1 \), we exchange \( O(1) \) messages to compute the rank. Since binary search terminates after \( O(\log n) \) steps, it follows that the median is located using \( O(\log n) \) messages. The correctness of this procedure follows directly from the correctness of the binary search procedure.

Therefore, the problem is solved if the median element is held by Karp. If this is not the case, the aforementioned protocol fails to locate an element in \( K \) with rank \( n - 1 \). In that case the protocol is repeated, with the roles of Karp and Rabin reversed.

This concludes the description of the simplest variant of the protocol. It exchanges \( O(\log n) \) messages, performs \( O(n \log n) \)-time computation, and is in-place, assuming that an in-place sorting algorithm is used. Various optimizations are possible. They do not decrease the asymptotic number of messages exchanged, but can reduce the running time to \( O(n) \).

There are many variants of the aforementioned procedure. For example, the elements \( a \) for which we compute \( \text{rank}_{K \cup R}(a) \) could be chosen from \( K \) in odd rounds and from \( R \) in even rounds. We could also choose those elements at random from \( K \) or \( R \), in a manner analogous to the Randomized Select procedure. This however leads only to \( O(\log n) \) messages in the expectation. In addition, the analysis of Randomized Select needs to be modified slightly, given that at each step we choose a random element from either \( K \) or \( R \), not from the whole data set.

**Problem 3.** The lightest path
Consider a weighted connected graph $G$, with $m$ edges and $n$ vertices. Each edge weight is an integer in the range $[1, 10n]$. Consider any two nodes $u, v \in G$, and a path $P$ between $u$ and $v$. The weight of the path $P$ is defined as the maximum weight of any edge in $P$.

Give an efficient algorithm that given the graph $G$ and vertices $u$ and $v$, finds the minimum weight path between $u$ and $v$ in $G$.

**Solution:**

A common approach taken by students was to modify Dijkstra’s algorithm. Instead of storing the shortest known distance to a node, store the smallest known path weight to the node. In particular, change the RELAX subroutine (which potentially updates the value at node $v$ using the best-known path to $u$) to the following:

```plaintext
RELAX(u,v,w)
1 if d[v] > max(d[u], w(u,v))
2 then d[v] ← max(d[u], w(u,v))
3 π[v] ← u
```

We essentially replace the ‘+’ operator with a ‘max’ operator. The arguments used in the proof of correctness for Dijkstra’s algorithm still holds after doing so. In particular, path weights are nondecreasing whenever they are extended (this is true even if we allowed negative edge weights) because the max operator will always keep the larger value. Thus we know that for the set of vertices that has not yet been relaxed, the vertex that has the minimum known cost can not get an improved cost by going through any other intermediary vertex. It is always therefore safe to relax that vertex and add it to the set of finished vertices to which we have found optimal paths. Interestingly enough, this problem loses optimal substructure in that it is no longer a strict requirement that all subpaths must be optimal for the entire path to be optimal. That is, if two parts of the path contain two different edges that both have the maximum edge weight required for the path, if we improve the first half of the path, the entire path is not made more optimal. Points were taken off for making incorrect arguments and for not making any.

Simply using a Fibonacci heap results in a running time of $O(m + n \log n)$. However, because the edge weights can only be integers in the range $[1, 10n]$, a monotone priority queue of size $10n$ can be used as described in Problem Set 5 since the only path weights possible are the edge weights. This results in an overall running time of $O(n + m)$. (Note: Because the graph is connected, we know that $n = O(m)$, so we can also simplify the running time to $O(m)$.)

The other main approach students took was to “grow” a copy of the graph by adding in the edges of the original graph in order of their edge weights. Once the two nodes $u$ and $v$ are connected, we know that the last edge added to the graph is the minimum path weight possible. The simplest way of implementing this algorithm is to first sort the edges by edge weight (by using counting sort, we can get $O(m + n)$ because the $m$ edge weights can only be of $10n$ different values), add the edges of the smallest edge weight not yet in the graph, test the connectivity of $u$ and $v$, and then iterate. Testing connectivity can be done by using Dijkstra’s (rather suboptimal), using BFS or DFS.
(which takes $O(m + n)$ each time for $O(mn)$ total running time), or maintaining the connected components using the Union-Find data structure described in CLRS (for $O(m\alpha(n))$ time). While the last of these techniques depends on growing the graph edge by edge, the former two techniques can be sped up by performing a binary search on the minimum edge weight required to connect $u$ and $v$. In other words, try the graph with only edge weights at most $5n$, then either $2.5n$ or $7.5n$, etc. This would yield a running time of $O(m\log n)$ when using BFS or DFS to test connectivity.

The Union-Find approach is implemented by first making each vertex into a set by itself. Then, whenever an edge is added, we union the two sets that contain the endpoints of the edge. We test connectivity by finding the set representatives of $u$ and $v$ and check if they’re the same. Using the union by rank heuristic gives an amortized bound of $O(\alpha(n))$ time per Union-Find operation, where $\alpha(n)$ is the inverse Ackerman function and grows incredibly slowly (its value is on the order of 4 or 5 on reasonable input sizes).

Another clever solution actually achieves $O(m)$ running time also by growing the graph but saves time by only keeping track of which nodes are reachable from $u$ using a variable stored at each node. At the beginning, $u$ is the only one marked as reachable from $u$. When adding the edges in sorted order, if one endpoint of the edge is marked as reachable and the other isn’t, mark the second vertex as reachable and perform a BFS to mark everything connected to that node as reachable. If $v$ is ever encountered, we’re done. If neither endpoint of the added edge is reachable or both of them are, simply insert the edge and do nothing. This algorithm only performs a constant amount of work when adding an edge to the graph for a total of $O(m)$ for all the edges, and the series of BFSs only traverse each vertex and edge once for a total of $O(m + n)$, so the entire algorithm runs in $O(m)$.

Another related approach is to build a minimum spanning tree. In an MST, the path between $u$ and $v$ necessarily has the minimum path weight possible in the graph. Thus we can simply run an MST algorithm and use DFS to find the path between $u$ and $v$. Since the MST is a tree, there is only one unique path for the MST. To see how the path in the MST is correct, it is useful to think of how Kruskal’s algorithm works. (This is fact pretty much like the Union-Find approach except we don’t add edges that are unnecessary.) We sort the edges by weight, and we keep adding edges if they do not form cycles. If the path in the MST is wrong, then that means that there is another path in the graph that connects through smaller weight edges in favor of some larger edge used in the MST. Obviously, not all of those edges are in the MST, or otherwise the MST would not be a tree or we would be taking this path. In the process of Kruskal’s algorithm, we would have considered adding all those lighter edges before having added in the larger edge. If we chose not to add some of those edges, it would be because adding them would not increase the connectivity; the MST would already have contained a path that connected the proposed edge’s endpoints with lesser or equal cost. This implies that if this better path of lighter edges existed, it or another path of equal or lesser cost would be added into the MST before the larger edge was added. Because this path is in existence, the larger edge can no longer be part of the path between $u$ and $v$. Similar to the Dijkstra’s algorithm approach, the traditional running time for finding the MST is not quite $O(m)$ but can be improved because of the integer edge weights.

All of these approaches can be performed using $O(m + n)$ space, and only $O(n)$ space for im-
implementing Dijkstra’s. Both were an acceptable use of space. Some students’ solutions stored the entire path at each node, resulting in using \( O(n^2) \) space and therefore \( \Omega(n^2) \) time, depending on what else their algorithm did. Note that this modification to Dijkstra’s is entirely unnecessary since the method shown in CLRS of simply maintaining the predecessor of a vertex is sufficient and only uses \( O(n) \) space. The path can be reconstructed after running Dijkstra’s simply by tracing back and following the predecessors.

**Problem 4. Updating Max-Flow** Let \( G = (V, E) \) be a flow network with source \( s \), sink \( t \), and integer capacities. Suppose that we are given a maximum flow in \( G \) obtained using the Ford-Fulkerson method. Suppose that the capacity of a single edge \((u, v) \in E \) is increased by 1. Give an \( O(V + E) \)-time algorithm to update the maximum flow.

**Solution:** Let \( f^* \) denote the maximum flow of \( G \) given to us by the Ford-Fulkerson method. By the max-flow min-cut theorem (27.7 in CLR), the fact that \( f^* \) is a maximum flow implies that the residual network \( G_{f^*} \) has no augmenting path. When we increase the capacity of \((u, v) \) by 1, the residual capacity \( c_{f^*}(u, v) \), defined as \( c(u, v) - f^*(u, v) \), also increases by 1. Notice that \( f^* \) is a valid flow in the modified flow network \( G' \). Thus, the value of the maximum flow cannot decrease; it can either stay the same or increase by at most 1 (an increase by more than 1 would contradict the maximality of \( f^* \) in \( G \)).

If there is no augmenting path in the residual network \( G'_{f^*} \), then \( f^* \) is a maximum flow (max-flow min-cut theorem). Otherwise, there is an augmenting path \( p \) in \( G'_{f^*} \) and we can ship some additional amount of net flow along the edges of an augmenting path \( p \). This amount is equal to the residual capacity of \( p \), which is greater than zero by the definition of the residual network. However, by the integrality theorem (27.11 in CLR), it must be an integer, because the capacities and flows of all the edges are integers. It follows that the residual capacity must be at least 1. Since the flow cannot increase by more than one unit, the residual capacity of \( p \) must be exactly 1.

To update the maximum flow, modify the capacity of \((u, v) \) and construct the residual network of \( G' \) with flow \( f^* \). This can be done in \( O(V + E) \) (see section 27.2). Next, find an augmenting path \( p \), using either BFS or DFS, which take \( O(V + E) \). If there is an augmenting path \( p \), for every edge \((v_1, v_2) \) on \( p \), increment \( f^*(v_1, v_2) \) by 1. The resulting flow is the maximum in \( G' \). If not, no update of the maximum flow is necessary: \( f^* \) is the maximum flow of \( G \) with the capacity of \((u, v) \) increased by 1.

**Problem 5. Search Engine Excerpts**

Most popular Internet search engines feature document excerpts for each found document. The goal of search engines is to provide the smallest document excerpt possible that helps the user realize if the found document is relevant to their search. One possible heuristic is to find the smallest excerpt from a found document that includes all of the keywords used in the search.

You are given \( k \) search keywords, and a document that is a sequence of \( N \) words. All \( k \) search keywords are present in the document.
Describe an efficient algorithm that finds a smallest range \([i, j]\) in the document \((i, j \in [1, N])\), that contains at least one instance of every search keyword.

Note: you can assume that each word or search keyword is represented by a number. **Solution:**

There were three \(O(N)\) solutions to this problem, as well as numerous variations that provided slower runtimes. All solutions begin by hashing each keyword \(k_i\) into a table \(T\) (either using perfect hashing, or some mechanism for resolving false positives in lookups).

The first general solution involves a “sliding window” pass over the document words from 1 to \(N\). More specifically, two pointers, \(i\) and \(j\), define a window over the document. The value associated with each \(k_i\) in \(T\) tracks the number of instances of \(k_i\) within this window, and a counter \(m\) tracks the number of entries with non-zero values. Both \(i\) and \(j\) start at index 1 of the document. \(j\) then slides forwards until \(m = 0\). Whenever a keyword is identified during the slide (in constant time via hash table lookup), the appropriate counter in \(C\) is incremented. Whenever an entry’s value goes from zero to one, \(m\) is incremented.

Once \(m = k\), we have a complete excerpt. To make the excerpt minimal, we now slide \(i\) forward, decrementing hash table entries and \(m\) when keywords are encountered, until incrementing \(i\) would make our excerpt incomplete. At this point we record our range as the current best complete range, and then slide \(i\) to the next keyword instance. We then repeat our process of sliding \(j\), then \(i\), and updating our best observed range, until \(j\) hits the end of our document.

This solution takes \(O(N)\) time and \(O(N)\) space.

The second general solution was to traverse the document, tracking the latest instances of all \(k_i\), and recording best ranges at each new keyword. More specifically, a linked list \(L\) of latest keyword instances is created during initialization, and populated with one element per \(k_i\) that stores the value -1. The values of the hash table \(T\) are each assigned one of these list elements. We now start a pointer \(i\) at the first document word and scan to the right. At each identified keyword at index \(j\) (again in constant time via \(T\)), the linked list element for \(k_i\) has its value updated to \(j\), and is spliced to the end of the linked list \(L\) (because we only ever increase values, causing elements to move to the end of the list, we are in effect implementing a monotone priority queue). Whenever a list element value is changed from -1, a counter \(m\) is incremented. Once \(m = k\), the interval \([1, i]\), contains at least one instance of each keyword. Whenever we find a new keyword, we take the value of the head of our linked list and our current index to form a best complete range, and continue updating this value as \(i\) slides forward. We terminate when \(j\) reaches the end of the list.

This solution takes \(O(N)\) time and \(O(N)\) space.

The third general solution involves splitting the keyword instances into \(k\) separate linked lists and processing them in quasi-parallel. More specifically, the document is first run through once, and linked lists of matched instances (again identified in \(O(1)\)) are formed off of each \(k_i\) entry in \(T\). These lists are of necessity sorted, and have aggregate length \(O(N)\). The range defined by the heads of each of these linked lists will define our current complete excerpt range. We record the maximum head pointer in \(\text{max}\) after a \(O(k)\) search. We now begin scanning the document a second time. Whenever a keyword \(k_i\) is matched, three actions occur. First, the range defined by the head
element of the list hashed to be $k_i$ and $max$ is checked for being the best range (which we track during the scan). Second, the head element of the list $k_i$ hashes to is removed. Third, the new head element is compared to $max$, and if found greater, takes $max$’s place. The scan stops whenever any of the linked lists is emptied.

This solution takes $O(N)$ time and space.

In general, full credit was given for a bugless $O(N)$ solution. Penalties were applied for bugs, significant omissions, extremely confusing writeups, or incomplete correctness or analysis. Partial credit was given to $O(N \lg k)$ solutions that used different data structures to either track latest keyword instances for solution two, or to find minimums in solution three. Less credit was given for similar solutions that used less efficient data structures to get $O(Nk)$ algorithms. Less credit was given to $O(N^2)$ algorithms that used a different, more exhaustive approach.

**Problem 6. Languages of the new empire**

In the glory days of the Roman empire, Julius Caesar seeks to establish efficient communications between Rome and the myriad provinces it commands in close and remote corners of the empire. One-way broadcast messages are sent frequently from Rome to all of the $N$ provinces along previously-specified paths, to announce new taxes, new wars, and other urgent matters. The problem is that many different languages are spoken across the empire, and thus a lot of consecutive translations may need to happen along the way to a remote province when a message is sent, causing unwanted delays. Each province speaks only one of the $K$ total languages in the empire, and Caesar insisted that they receive messages in their own language (his magnanimity knows no limits).

Communication happens by messengers carrying stone tablets, along a graph $G = (V, E)$ of roads and sailing routes. Since all roads lead to Rome and only three-way intersections are present, the communication graph is a binary tree, with Rome at the root and provinces at the leaves. At each intersection (each internal node) lies an outpost where a single stone tablet arrives, and is then copied and passed on (exactly once for each of the two outgoing paths), in the same language when possible, or translated to a new language when necessary. For every broadcast, a single message tablet is created in Rome, in Latin, and arrives at Rome’s outpost, where it starts its journey throughout the empire. After each intersection, exactly one tablet, in a specified language, leaves down each subsequent path. Since translation of messages takes a long time, you need to select the language of tablets to be used along each segment in a way that minimizes the total cost of needed translations.

In addition to the graph $G$, and the language spoken at each province, you are given the cost of translating from each language to each other language, represented as a $K \times K$ matrix $M$ of positive numbers for the cost $M[i,j]$ of translating from language $i$ to language $j$. Note that translation costs need not be symmetrical, that the cost of translating a language to any other language is always strictly positive, and that all diagonal entries of matrix $M$ are zero as no translation is necessary from a language to itself. You can also assume that the costs reported in $M[i,j]$ are optimal,
namely you cannot get a lower translation cost from \( i \) to \( j \) by translating through a series of other languages.

Design an efficient algorithm to select the language of tables to be used on each segment (and the translations done at each outpost) in order to minimize the total cost of translations. For partial credit, you can simplify the problem by assuming that any translation has unit cost, i.e. \( M[i,j] = 1 \) for \( i \neq j \).

**Solution:**

We set up a Dynamic Programming (DP) Cost table \( \text{Cost}_v[1..K] \) for each vertex \( v \), to be filled in starting from the leaves and towards the root. At each entry \( \text{Cost}_v[i] \), we store the minimum cost of translations for sending a message from vertex \( v \) to all of the provinces in that subtree, if language \( i \) is used in the road leading to that outpost from Rome. We update the \( \text{Cost}_v \) table for each outpost \( v \) based on the two tables \( \text{Cost}_{v\text{right}} \) and \( \text{Cost}_{v\text{left}} \) at its two children \( v\text{right} \) and \( v\text{left} \), by evaluating all combinations of costs at that intersection and taking the min. Once the min is found, we also store the choice of assignment leading to the optimal score at each vertex in a table \( \text{Assignment}_v[i] \), with entries \((j,k)\) if language choices \( j \) and \( k \) at the children led to minimum value, thus enabling a traceback back towards the leaves after the root is reached, in order to find the final language assignment for each outpost \( v \).

We start by the proof that DP is applicable here. First, there’s only a finite number of subproblems, as a leaf with \( N \) leaves has \( N - 1 \) internal vertices, each of which has only \( K \) possible languages. Second, the problem exhibits optimal substructure: a solution using the minimum cost of translations to get from Rome to any of the provinces must be made of optimal solutions to get from any of the internal outposts to the provinces it leads to. The proof is by contradiction using a cut-and-paste argument: if a solution from a given province existed with a smaller cost of translations, that portion could be replaced in the overall solution, leading to an overall solution better than the optimal, hence a contradiction.

Next, we set up the dynamic programming update rule, to compute the table of optimal translation costs for a given outpost \( v \) whose children \( v\text{right} \) and \( v\text{left} \) have their optimal costs already computed. In the general case, the cost of using language \( l \) at outpost \( v \) would be computed as:

\[
\text{Cost}_v[l] = \min_{i,j} \{ \text{Cost}_{v\text{right}}[i] + \text{Cost}_{v\text{left}}[j] + M[l,i] + M[l,j] \}
\]

This general update rule would have a cost of \( k^3 \), as each of \( k \) entries are filled in, each time testing \( k^2 \) combinations of assignments for the two children. However, since translation costs are independent between the two sides, a speedup is possible:

\[
\text{Cost}_v[l] = \min_i \{ \text{Cost}_{v\text{right}}[i] + M[l,i] \} + \min_j \{ \text{Cost}_{v\text{left}}[j] + M[l,j] \}
\]

This independence relies on the fact that no optimal solution requires that two consecutive translations are necessary at a single node \( v \) (from \( i \rightarrow j \) and then \( j \rightarrow k \)) and that only \( i \rightarrow j \) and \( i \rightarrow k \) solutions are allowed at any one node. The reason for this is that instead of sending \( i \) to node \( v \), and then doing two consecutive translations at that node, we can simply translate from \( i \) to \( j \) at the parent of \( v \), and then send \( j \) down that node which we then translate to \( k \) at node \( v \).
The optimized update rule allows us to pay only cost $k^2$ for each outpost, since each of $k$ entries is updated in $O(k)$ time. Note that $Assignment_v[i]$ records the assignment of the left and right children of $v$ that lead to the optimal translation cost when the language at $v$ is set to be $i$. Also note that the two subloops can be run in any order, as the 'relaxation' step is never surpassing the optimal solution (additional relaxation steps don’t hurt).

We initialize the DP computation by setting the cost table for each leaf $p$ (corresponding to province $p$) as follows:

$$Cost_p[j] = M[j,i],$$

where $i$ is the language spoken at province $p$.

(an alternative strategy may be to set $Cost_p[i] = 0$ and $Cost_p[j] = \infty$ for all other languages, as we know that no optimal solution will use a different language at the last outpost than the language spoken at that province).

Upon termination, the root contains a vector of the value of the globally optimal solution for each choice of language starting at the Rome outpost. Since the incoming language there is always Latin, we have to add to that value $Cost_{Root}[i]$ the cost of translating Latin to that language $M[Latin,i]$. We choose the language $i$ minimizing that sum, set that language for the Rome outpost. The $Assignment_{Rome}[best]$ then starts the series of pointers to children vertex assignments that we trace back to construct the optimal solution, assigning languages to all outposts.

The run time analysis for the entire algorithm is simple. $O(N \times K^2)$ for computing the optimal solution, since a binary tree with $n$ leaves has $n-1$ internal vertices, and since the table of each vertex is computed exactly once, and each computation takes $O(k^2)$ time. The traceback takes linear time as each optimal solution is immediately looked up by following the $Assignment_v[best]$ pointers. The space requirements are $O(N \times K)$ for storing $K$ values and $K$ tuples at each vertex.

Notes:

Note that a greedy algorithm is not possible. The reason is that the local information within a subtree is insufficient to make a globally-optimal choice. For example, if A and B at the root of a subtree have equal cost within the subtree, the choice will not be resolved until we know whether A and B are dominant outside the subtree. More generally, even if some language L1 is giving lowest cost within a subtree, another language L2 may in fact come to be cheapest for that subtree considering the outside information.

In particular, even for the unit-cost version of the problem, simple voting schemes will not work (of counting the provinces beneath a node that speak each language, and choosing the majority). The reason is that it’s the layout of the languages that matters, not the total count. For example, the subtree $((B,(B,(B,((A,A),(A,((A,A),(A,A))))))))))$ contains 7 A’s and 6 B’s, hence at every node, the majority-voting scheme will choose A, leading to a total of 6 translations, from the top-most A into each individual B. The optimal solution however starts with a top-most B and only requires 1 translation from B to A at the root of the all-A subtree.

It should also be noted that the overlapping subproblems nature of the problem can only be exploited in bottom-up algorithms that move from the leaves to the root. The reason is that optimal solutions to the subproblems (given a starting language at the root) can only be computed for
fully-explored subtrees at a time. This is not possible from the root down, and would lead to exponential-time algorithms that have to travel all the way down to the leaves at each iteration to collect information.

To receive full credit, solutions had to demonstrate a working Dynamic Programming (DP) solution, including set-up of cost variables, bottom-up propagation of costs, a precise and correct update rule with initialization and termination conditions, a traceback step after appropriate update of max pointers, and a proof of correctness and analysis of running time. Partial credit was received for Greedy solutions that propagated sums to the root and traced back to maximum-frequency choices, as they demonstrated the principles of DP, despite the misunderstanding described above.