1928. Minimum Cost to Reach Destination in Time

Problem Description

There is a country with `n` cities, numbered from 0 to `n` - 1, where all the cities are connected by bi-directional roads. The roads are represented by a 2D integer array `edges`, where `edges[i] = [x_i, y_i, time_i]` denotes a road between cities `x_i` and `y_i` that takes `time_i` minutes to travel. There may be multiple roads of differing travel times connecting the same two cities, but no road connects a city to itself.

Each time you pass through a city, you must pay a passing fee. The passing fee is represented as a 0-indexed integer array `passingFees` of length `n` where `passingFees[j]` is the amount of dollars you must pay when you pass through city `j`.

Initially, you are at city 0 and want to reach city n - 1 in `maxTime` minutes or less. The cost of your journey is the summation of passing fees for each city you passed through at some moment during your journey (including the source and destination cities).

Given `maxTime`, `edges`, and `passingFees`, return the minimum cost to complete your journey, or -1 if you cannot complete it within `maxTime` minutes.

Example

For example, let's suppose we have `maxTime = 10`, `edges = [[0, 1, 2], [1, 2, 4], [2, 3, 2]]`, and `passingFees = [1, 2, 3, 4]`. The cities and roads can be illustrated as follows:

```
1
 \
  2---3
 / \
0---2
```

We will solve the problem step-by-step with an algorithm called Dijkstra.

Solution Approach

The solution uses the Dijkstra algorithm to find the shortest path. Dijkstra helps us find the shortest path between nodes in a graph as well as the minimum total cost for passing each city. For this problem, we use Dijkstra to find the smallest time and cost to reach each city via different roads.

We will use a priority queue (min-heap) to keep track of the minimum cost and time reachable for each city. The algorithm works as follows:

1. Create a graph representing the cities and roads.
2. Initialize cost and distance arrays, and add the starting city to the priority queue.
3. While the priority queue is not empty, pop the next city with the minimum cost and time.
4. If we have reached the destination city, return the cost.
5. If not, loop through the neighboring cities and update their costs and times, adding them to the priority queue if necessary.

Let's now discuss the solution implementation.

C++ Solution

```
cpp
class Solution {
 public:
  int minCost(int maxTime, vector<vector<int>>& edges,
              vector<int>& passingFees) {
    const int n = passingFees.size();
    vector<vector<pair<int, int>>> graph(n);

    // Create the graph with edges
    for (const vector<int>& edge : edges) {
      const int u = edge[0];
      const int v = edge[1];
      const int w = edge[2];
      graph[u].emplace_back(v, w);
      graph[v].emplace_back(u, w);
    }

    // Run Dijkstra algorithm
    return dijkstra(graph, 0, n - 1, maxTime, passingFees);
  }

 private:
  int dijkstra(const vector<vector<pair<int, int>>>& graph, int src, int dst,
               int maxTime, const vector<int>& passingFees) {
    // cost[i] := min cost to reach cities[i]
    vector<int> cost(graph.size(), INT_MAX);
    // dist[i] := min time to reach cities[i]
    vector<int> dist(graph.size(), maxTime + 1);
    using T = tuple<int, int, int>;  // (cost[u], dist[u], u)
    priority_queue<T, vector<T>, greater<>> minHeap;

    cost[src] = passingFees[src];
    dist[src] = 0;
    minHeap.emplace(cost[src], dist[src], src);

    while (!minHeap.empty()) {
      const auto [currCost, d, u] = minHeap.top();
      minHeap.pop();
      if (u == dst)
        return cost[dst];
      for (const auto& [v, w] : graph[u]) {
        if (d + w > maxTime)
          continue;
        // Go from u -> v.
        if (currCost + passingFees[v] < cost[v]) {
          cost[v] = currCost + passingFees[v];
          dist[v] = d + w;
          minHeap.emplace(cost[v], dist[v], v);
        } else if (d + w < dist[v]) {
          dist[v] = d + w;
          minHeap.emplace(currCost + passingFees[v], dist[v], v);
        }
      }
    }

    return -1;
  }
};
```

The C++ solution follows the algorithm mentioned above. It creates a graph to represent the cities and roads, initializes the cost and distance arrays, and uses Dijkstra's algorithm to find the minimum cost with the given time constraint.## Python Solution

```
python
from heapq import heappop, heappush

class Solution:
    def minCost(self, maxTime: int, edges: List[List[int]], passingFees: List[int]) -> int:
        n = len(passingFees)
        graph = {i: [] for i in range(n)}

        for u, v, w in edges:
            graph[u].append((v, w))
            graph[v].append((u, w))

        cost = [float('inf')] * n
        dist = [maxTime + 1] * n
        min_heap = [(passingFees[0], 0, 0)]

        cost[0] = passingFees[0]
        dist[0] = 0

        while min_heap:
            curr_cost, d, u = heappop(min_heap)

            if u == n - 1:
                return cost[u]

            for v, w in graph[u]:
                if d + w > maxTime:
                    continue

                if curr_cost + passingFees[v] < cost[v]:
                    cost[v] = curr_cost + passingFees[v]
                    dist[v] = d + w
                    heappush(min_heap, (cost[v], dist[v], v))
                elif d + w < dist[v]:
                    dist[v] = d + w
                    heappush(min_heap, (curr_cost + passingFees[v], dist[v], v))

        return -1
```

The Python solution follows a similar structure as the C++ solution. Here, we use dictionaries to represent the graph and Python's heapq package to implement the priority queue (min-heap).

JavaScript Solution

```
javascript
class PriorityQueue {
  constructor(compare) {
    this.heap = [];
    this.compare = compare;
  }

  push(val) {
    this.heap.push(val);
    this.heap.sort(this.compare);
  }

  pop() {
    return this.heap.shift();
  }

  empty() {
    return this.heap.length === 0;
  }
}

function minCost(maxTime, edges, passingFees) {
  const n = passingFees.length;
  const graph = Array.from({ length: n }, () => []);
  for (const [u, v, w] of edges) {
    graph[u].push([v, w]);
    graph[v].push([u, w]);
  }

  const cost = new Array(n).fill(Infinity);
  const dist = new Array(n).fill(maxTime + 1);
  cost[0] = passingFees[0];
  dist[0] = 0;

  const minHeap = new PriorityQueue(
    (a, b) => a[0] - b[0] || a[1] - b[1] || a[2] - b[2]
  );
  minHeap.push([cost[0], dist[0], 0]);

  while (!minHeap.empty()) {
    const [currCost, d, u] = minHeap.pop();
    if (u === n - 1) return cost[u];
    for (const [v, w] of graph[u]) {
      if (d + w > maxTime) continue;
      if (currCost + passingFees[v] < cost[v]) {
        cost[v] = currCost + passingFees[v];
        dist[v] = d + w;
        minHeap.push([cost[v], dist[v], v]);
      } else if (d + w < dist[v]) {
        dist[v] = d + w;
        minHeap.push([currCost + passingFees[v], dist[v], v]);
      }
    }
  }

  return -1;
}
```

The JavaScript solution is also similar to the C++ and Python solutions. However, since JavaScript does not have a built-in heapq package, we implement a simple priority queue using arrays and comparator functions. The rest of the solution follows the same Dijkstra's algorithm steps.