1 Problem C: Catering

1.1 Problem Specification

Paul owns a catering company and business is booming. The company has k catering teams, each in charge of one set of catering equipment. Every week, the company accepts n catering requests for various events. For every request, they send a catering team with their equipment to the event location. The team delivers the food, sets up the equipment, and instructs the host on how to use the equipment and serve the food. After the event, the host is responsible for returning the equipment back to Paul's company.

Unfortunately, in some weeks the number of catering teams is less than the number of requests, so some teams may have to be used for more than one event. In these cases, the company cannot wait for the host to return the equipment and must keep the team on-site to move the equipment to another location. The company has an accurate estimate of the cost to move a set of equipment from any location to any other location. Given these costs, Paul wants to prepare an Advance Catering Map to service the requests while minimizing the total moving cost of equipment (including the cost of the first move), even if that means not using all the available teams. Paul needs your help to write a program to accomplish this task. The requests are sorted in ascending order of their event times and they are chosen in such a way that for any $i < j$, there is enough time to transport the equipment used in the i-th request to the location of the j-th request.

1.2 Mathematical Formulation

1.3 Problem Size

We are given an unknown amount of test cases that we have to solve in under three seconds. However, our given variables do have bounds. The number of teams, k, and the number of requests, n, are both bounded by 0 and 100, inclusive. The ith request (counting from 1) has $n - i + 1$ connections to the requests after it.

1.4 Solution Description

To solve this problem, we model the input as a bipartite graph, and then try to find a minimum cost matching. Therefore, the main parts of our algorithm are: read in the input and build an adjacency matrix for a bipartite graph, and then, using that graph and a minimum cost bipartite matching algorithm with Dijkstra's single source shortest paths (with an indexed minimum priority queue), find the minimum cost matching.

The main runner for our program is written below. It is essentially the algorithm described above.

Algorithm 1 Main runner of our program.

```
procedure Main
    in ← new Scanner for system.in
    while in has more integers to read do
        n ← next integer from in
        k ← next integer from in
        g ← BuildGraphFromInput(in, n, k)
        minCost ← GetMinCostMatching(g, n, k)
        Print(minCost)
```

Before we can even solve the problem, we have to build a graph which represents it. When we call the BuildGraphFromInput method, we read in the input from standard input and we build our bipartite graph using that data. The algorithm for building the graph is detailed below:

Algorithm 2 Builds a bipartite graph from the input.

```
procedure BuildGraphFromInput(in, n, k)
    numNodes ← 1 + k + (2 × n)
    leftStart ← 1
    rightStart ← k + n
    g ← new array of size (numNodes × numNodes)
    Fill up all indices of g with ∞
    teamValues ← new array of size (numNodes)

    for i ∈ [0, numNodes) do
        teamValues[i] ← next integer from in
    for u ∈ [leftStart, k] do
        for i ∈ [0, n) do
            g[u][k + n + i] ← teamValues[i]
    for u ∈ [leftStart + k, rightStart) do
        for v ∈ [u + n, numNodes − 1) do
            g[u][v] ← next integer from in
    for i ∈ [leftStart, rightStart) do
        g[0][i] ← 0
    for i ∈ [rightStart, numNodes − 1) do
        g[i][numNodes − 1] ← 0

    return g
```

After building our bipartite graph, we simply run a minimum cost bipartite matching algorithm (our specific implementation utilizes Dijkstra's Single Source Shortest

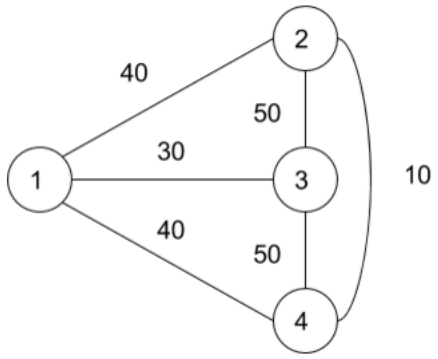
Paths algorithm with an indexed minimum priority queue) to find the minimum matching cost. This is done within the GetMinCostMatching method. After doing that, we just print out the value we found and, thus, the problem is solved.

1.5 Examples and Test Cases

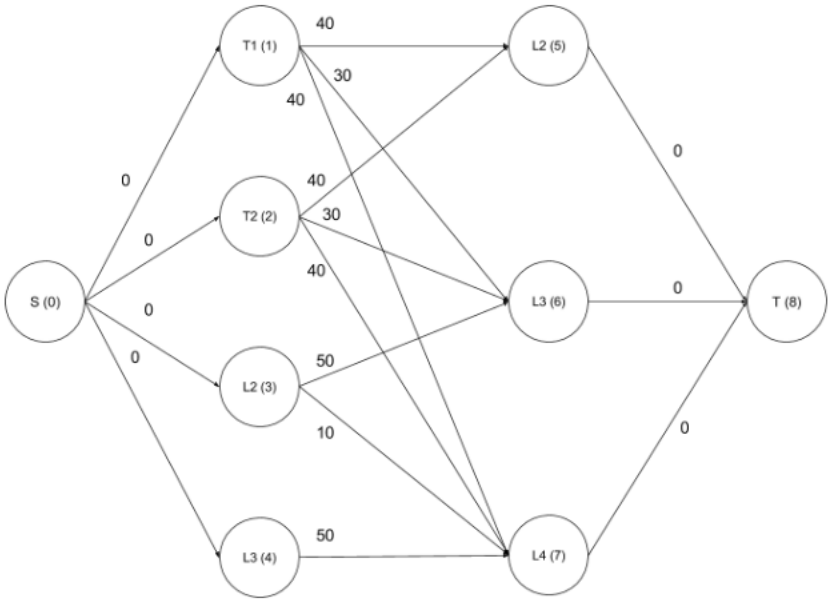
To show how our algorithm actually works, we'll run through a quick example using one of the sample test cases. Say our input is as follows:

```
3 2
40 30 40
50 10
50
```

This means that we have three requests ($n = 3$) and two teams ($k = 2$). We can draw this input as a location graph, showing us the connections between different locations and the costs to move from one location to another.



However, to actually solve this problem, we need to create a bipartite graph so we can use the minimum cost bipartite matching algorithm. The graph and the corresponding adjacency matrix can be found below (vertex numbers are in parentheses, T means team, and L means location):



$$\begin{bmatrix} \infty & 0 & 0 & 0 & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 40 & 30 & 40 & \infty \\ \infty & \infty & \infty & \infty & \infty & 40 & 30 & 40 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 50 & 10 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 50 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

When we run the minimum cost bipartite matching algorithm on this adjacency matrix, we find that the minimum cost is 80. That corresponds to sending team 1 to location 2 then 4, and sending team 2 to location 3.

1.6 Correctness

1.7 Efficiency