Travelling Salesman Problem?

This problem was worth 2000 points.
The author of this problem is Aryan V S.

Note: GitHub does not support LaTex in Markdown. If you want a more readable version of the problem, download the PDF file instead.

Statement

Don't worry, this isn't *the* Travelling Salesman Problem. Or is it? 🙂

Aryan is now working as a salesman. He is currently at city 1 and needs to reach city n, where he has a meeting with Dhruv!

To meet Dhruv, he first needs to travel through cities $[2, \ldots, n-1]$ in that exact order. Formally, he must travel to city 2 from city 1, $\ldots$, city $i+1$ from city i, $\ldots$, city n from city $n-1$.

He has k modes of transport T_j $(1 \leq j \leq k)$ to move from one city i to another city $i+1$.

To travel from city i to city $i+1$ $(1 \leq i \leq n-1)$ using transport T_j, it requires T_{ji} time. However, there's a catch!

If the mode of transport from previous city $i-1$ to current city i is **different** from the choice he makes to travel from current city i to next city $i+1$, he has to wait for additional time w_{ji} where T_j is the mode of transport he chooses at the current city i. If they are the **same**, then he does not need to wait for any time because he can continue to travel in the same vehicle.

Aryan wants to reach city n as fast as possible. Find the minimum time that he would spend travelling.

Note: Read sample explanations if you have difficulty understanding the statement.

Remember that at the initial city 1, Aryan *must* wait w_{j1} time for his transport T_j to arrive (explained in a sample test).

Some additional information about the testcases

Excluding sample testcases, they have the following properties:

- 1 testcase has $k = 1$
- 2 testcases have $k = 2$
- 2 testcases have $k = 3$
- 3 testcases have $k^n \leq 10^8$
- Remaining testcases have the constraints as mentioned in the problem

Can you use this information to your benefit to maximise your total points?

Input Format

The first line contains a two space separated integers n and k.

The following k lines contain the arrays T_j. Each array contains $n - 1$ elements where the i^{th} element is T_{ji} (the travel time to city $i + 1$ from city i using mode of transport T_j).

The following k lines contain the arrays w_j. Each array contains $n - 1$ elements where the i^{th} element is w_{ji} (the wait time at city i if the previous mode of transport **was not** T_j).

Output Format

Output a single integer - the minimum time Aryan spends travelling to city n from city 1.

Constraints

$2 \leq n \leq 10^4$

$1 \leq k \leq 10$

$1 \leq T_{ji},\ w_{ji} \leq 10^7$

Sample Tests

Sample Test 1

Input

```
5 2
1 2 3 4
4 2 2 2
4 3 1 2
2 4 1 4
```

Output

```
12
```

Explanation

The optimal solution is to choose:

- Transport T_1 from city 1 to city 2
- Transport T_1 from city 2 to city 3
- Transport T_2 from city 3 to city 4
- Transport T_2 from city 4 to city 5

At city 1, we have to wait for $w_{1,1} = 4$ time for transport T_1 and then spend $T_{1,1} = 1$ time to travel to city 2. Total time so far: 5.

At city 2, we can choose to travel with transport T_1 itself and not have to wait at all. We spend $T_{1,2} = 2$ time to travel to city 3. Total time so far: 7.

At city 3, we change our mode of transport to T_2. Therefore, we have to wait for $w_{2,3} = 1$ time to wait for our transport to arrive. We then spend $T_{2,3} = 2$ time to travel to city 4. Total time so far: 10.

At city 4, we continue using transport T_2 and do not have to spend any time waiting. It takes us $T_{2,4} = 2$ time to travel to city 5. Total time so far: 12.

The minimum time it takes Aryan to travel from city 1 to city 5 is 12.

Sample Test 2

Input

```
10 3
4 5 2 6 5 9 7 4 8
6 2 4 9 3 8 6 3 10
3 5 8 5 10 1 1 5 2
5 7 2 10 5 8 10 1 2
4 9 5 3 7 6 4 4 6
1 9 1 5 7 8 3 8 9
```

Output

```
41
```

Solution

Time Complexity: $O(n \cdot k^2)$

Space Complexity: $O(n \cdot k)$

I will first explain the brute force approach and then try to optimise that into the final complexities.

$O(k^n)$ **approach:**

For every city, we have k possible choices of transport. We can simply try all possible choices at every city (also keeping track of previous transport mode to determine waiting-time, if any) using DFS (Depth First Search). We observe that testing all possibilities will definitely not be optimal.

What could we optimise so that we don't have to check all possibilities at every city? Instead of checking recursively all k modes of transport at each city, we do a check beforehand from every city to every next city with a total of k^2 transport pairs at each city. This, too, can be implemented using DFS directly. But, it is more commonly implemented as iterative DP (Dynamic Programming).

DP is just another form of DFS that is applied on a Directed Acyclic Graph where nodes represent "DP states" and edges represent "DP transitions". Usually, you will first need to find a topological ordering for a graph to apply DP on it but in this case, the topological ordering $t_i = i$ for all i.

$k = 1$ **approach:**

We only have one mode of transport. So, we only have to wait at city 1 initially, after which we can continue using the same mode of transport to just travel the distances between cities. The solution in this case is $w_{1,\,1} + \sum_{i=1}^{n-1} T_{1,\,i}$.

$k = 2$ **approach:**

Similar to $k = 1$ approach, we now try all $k^2 = 4$ combinations of the modes of transport at each city.

Final Solution:

Let us define our dp states as $dp_{j,i}$ as the minimum time it takes to get to city i if we use transport mode j.

From our dp definition, it follows that: $dp_{j,1} = 0$ for all transport modes j. At this point, we *must* wait for our transport to arrive so that we can travel to city 2. As we're going to add it anyway for all j modes of transport, instead of handling that inside our loop as a special case, we will modify our definition of dp only for city 1. Let $dp_{j,1} = w_{j,1}$ for all transports i.e. the minimum wait time for transport j to reach us in city 1. Remember that we are not changing our initial dp definition for any other city except city 1 (which we did to make our implementation a little easier).

Now, for every city, we simply try all k^2 transport pair combinations and find which one is the most optimal (has the smallest (wait + travel) time). You need to take care about adding the wait time if a transport pair is different. If they are the same, we can choose to use the same vehicle and just travel the distance between the cities without any wait time. Once we've calculated the transitions for every transport and every city, we find the minimum $dp_{j,n}$ - the minimum time that it would take any of the transports to reach city n.

Brute force solution in C++:

```cpp
/* Arrow */

#ifdef LOST_IN_SPACE
#  if   __cplusplus > 201703LL
#    include "lost_pch1.h" // C++20
#  elif __cplusplus > 201402LL
#    include "lost_pch2.h" // C++17
#  else
#    include "lost_pch3.h" // C++14
#  endif
#else
#  include <bits/stdc++.h>
#endif

constexpr bool test_cases = false;

void solve () {
  int n, k;
  std::cin >> n >> k;

  std::vector <std::vector <int64_t>> T (k, std::vector <int64_t> (n));
  std::vector <std::vector <int64_t>> w (k, std::vector <int64_t> (n));

  for (int j = 0; j < k; ++j)
    for (int i = 1; i < n; ++i)
      std::cin >> T[j][i];

  for (int j = 0; j < k; ++j)
    for (int i = 1; i < n; ++i)
      std::cin >> w[j][i];

  const int64_t inf = 1e18;
  int64_t best = inf;

  auto dfs = [&] (auto self, int u, int transport, int64_t time) -> void {
    if (u == n) {
      best = std::min(best, time);
      return;
    }
    for (int i = 0; i < k; ++i)
      if (i == transport)
        self(self, u + 1, i, time + T[i][u]);
      else
        self(self, u + 1, i, time + T[i][u] + w[i][u]);
  };


  dfs(dfs, 1, -1, 0);

  std::cout << best << '\n';
}

int main () {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  std::cout.precision(10);
  std::cerr.precision(10);
  std::cout << std::fixed << std::boolalpha;
  std::cerr << std::fixed << std::boolalpha;

  int cases = 1;
  if (test_cases)
    std::cin >> cases;
  while (cases--)
    solve();

  return 0;
}
```

Solution in C++:

```
/* Arrow */

#ifdef LOST_IN_SPACE
#  if   __cplusplus > 201703LL
#    include "lost_pch1.h" // C++20
#  elif __cplusplus > 201402LL
#    include "lost_pch2.h" // C++17
#  else
#    include "lost_pch3.h" // C++14
#  endif
#else
#  include <bits/stdc++.h>
#endif

constexpr bool test_cases = false;

void solve () {
  int n, k;
  std::cin >> n >> k;
```

```cpp

  std::vector <std::vector <int64_t>> T (k, std::vector <int64_t> (n));
  std::vector <std::vector <int64_t>> w (k, std::vector <int64_t> (n));

  for (int j = 0; j < k; ++j)
    for (int i = 1; i < n; ++i)
      std::cin >> T[j][i];

  for (int j = 0; j < k; ++j)
    for (int i = 1; i < n; ++i)
      std::cin >> w[j][i];

  const int64_t inf = 1e18;
  std::vector <std::vector <int64_t>> dp (k, std::vector <int64_t> (n + 1, inf));
  // dp[0][i] = minimum time if I use transport mode P to get to city i
  // dp[1][i] = minimum time if I use transport mode Q to get to city i
  // ...
  // dp[j][i] = minimum time if I use transport mode j to get to city i

  // initially we must wait for w[j][1] time for each transport because we haven't started travelling yet
  for (int j = 0; j < k; ++j)
    dp[j][1] = w[j][1];

  for (int i = 2; i <= n; ++i) {
    // x is the mode of transport we plan to use in the current city i
    // y is the mode of transport we used in the previous city (i - 1) to get to current city
    for (int x = 0; x < k; ++x) {
      for (int y = 0; y < k; ++y) {
        if (x == y)
          dp[x][i] = std::min(dp[x][i], dp[y][i - 1] + T[x][i - 1]);
        else
          dp[x][i] = std::min(dp[x][i], dp[y][i - 1] + T[x][i - 1] + w[x][i - 1]);
      }
    }
  }

  int64_t best = inf;
  for (int i = 0; i < k; ++i)
    best = std::min(best, dp[i][n]);

  std::cout << best << '\n';
}

int main () {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  std::cout.precision(10);
  std::cerr.precision(10);
  std::cout << std::fixed << std::boolalpha;
  std::cerr << std::fixed << std::boolalpha;

  int cases = 1;
  if (test_cases)
    std::cin >> cases;
  while (cases--)
    solve();

  return 0;
}
```

Solution in Python:

```python
read = lambda: list(map(int, input().split()))

n, k = read()
T = []
w = []

for i in range(k):
  T.append([0, *read()])

for i in range(k):
  w.append([0, *read()])

inf = 10 ** 18
dp = [[inf for _ in range(n + 1)] for j in range(k)]

for j in range(k):
  dp[j][1] = w[j][1]

for i in range(2, n + 1):
  for x in range(0, k):
    for y in range(0, k):
      if x == y:
        dp[x][i] = min(dp[x][i], dp[y][i - 1] + T[x][i - 1])
      else:
        dp[x][i] = min(dp[x][i], dp[y][i - 1] + T[x][i - 1] + w[x][i - 1])

best = inf
for j in range(k):
  best = min(best, dp[j][n])

print(best)
```