Olympiads

Spoiler

Subtask 1 was intended to be done with two nested loops.

Subtask 2 was intended to be done with exhaustive search (but due to low K can also be done with 6 nested loops).

Subtask 3 was intended to be done with a Brute Force on the scores for each event, and then another brute force to find the actual teams. The search was supposed to be done in decending order of total scores and stopped when C teams have been processed.

For constructing the full solution, the time constraints have been set to allow multiple different solutions to pass. The trick here is to perform the search in a manner such that higher total scores will be processed before lower ones. It's possible to solve it with A*, or bounded Brute Force on event scores, however a very fast and elegant solution utilizes something called "Fracture Search".

In principle, fracture search works in the following manner:

1. Pick some team $B = (B_1, \ldots, B_K)$ that gives the best total score.
2. Divide the search space into some number of subspaces such that the subspaces are disjoint and together cover all elements except B.
3. From the "unfractured" search spaces, pick the one where the best team has the highest total score. Repeat the fracture search on that subspace.

For this problem, the search space can be fractured into the following subspaces:

1. Contestant B_1 is excluded.
2. Forced to use contestant B_1, contestant B_2 is excluded.

...

K. Forced to use contestants $B_1, \ldots, B_{K-1}$, contestant B_K is excluded.

It's easy to see that the subspaces are disjoint and cover all teams except B. Next we can make our approach even more elegant by picking the best team in the following manner:

1. Contestant B_1 is the one that has the best score in event 1.
2. Contestant B_2 is the one among those that haven't already been picked that has the best score in event 2.

...

K. Contestant B_K is the one among those that haven't already been picked that has the best score in event K.

It's easy to see that this always gives the best team. It's also good because in the subspaces if you are forced to use contestants $B_1, \ldots, B_S$, then they also correspond to the first S contestants picked by the above approach in the same order. This combination makes our fracture search relatively easy to perform.

```
from heapq import heappush, heappop
from collections import namedtuple
from copy import copy

Shard = namedtuple('Shard', ['score', 'best', 'forced', 'forbitten'])

n, k, c = [int(x) for x in input().split(' ')]
scores = [[int(x) for x in input().split(' ')] for i in range(n)]

def pairwise_max(a, b):
    return [max(a[i], b[i]) for i in range(len(a))]

def evaluate_shard(shard):
    best = []
    event_best = [0] * k
    for x in shard.forced:
        best.append(x)

    for i in range(len(shard.forced), k):
        best.append(-1)
        for j in range(n):
            if not shard.forbitten[j] and j not in best:
                if best[i] == -1 or scores[j][i] > scores[best[i]][i]:
                    best[i] = j

    for c in best:
        event_best = pairwise_max(event_best, scores[c])
    return(Shard(-sum(event_best), best, shard.forced, shard.forbitten))


heap = [Shard(0, [], [], [False]*n)]
heap[0] = evaluate_shard(heap[0])
results = []

while len(results) < c:
    top = heappop(heap)
    results.append(-top.score)

    new_forced = copy(top.forced)
    new_forbitten = copy(top.forbitten)


    for i in range(len(top.forced), k):
        new_forbitten[top.best[i]] = True
        new_shard = evaluate_shard(Shard(0, [], copy(new_forced), copy(new_forbitten)))
        if -1 not in new_shard.best:
            heappush(heap, new_shard)
        new_forced.append(top.best[i])

print(results[c-1])
```

Credits

- Task: David Narum (Norway)
- Solutions and tests: Oliver-Matis Lill, Andres Unt (Estonia), David Narum (Norway)