SACO 2009 Day 2 Solutions

SACO Scientific Committee

1 Counting Truths

The most important observation to make is that if guard i is telling the truth and $K_i \geq K_j$ then guard j must also be telling the truth (if at least K_i guards are telling the truth, then at least K_j guards must also be). So the set of guards always consists of the first M guards (after sorting the K_i sequence), for some M.

If exactly M guards are telling the truth, then $K_i \leq M$ for $1 \leq i \leq M$ – the first M guards are truthful – and $K_j > M$ for $M < j \leq N$ – the remaining guards are lying. Since the guards are sorted, it suffices to check that $K_M \leq M$ and $K_{M+1} > M$. So we simply run through all $M \in 1, \ldots, N$ to find the smallest one that satisfies these two conditions.

This search runs in linear time; the overall time depends on the sorting algorithm used. An $O(N \log N)$ sort like the one built into Python is sufficient to get a full score, but it is possible to use a counting sort to get $O(N)$ overall.

2 As Cunning As A Fox

A straight bruteforce of this problem should give you 50%, assuming you brute-force each cage, and then crudely sum ($O(N^3)$) all the cages.

To get slightly better marks, your initial observation you should be the symmetrical pattens of the cube, such that any cage can be decomposed into the sum of it's components. This allows one to only store a 1-d array of the cube, and derive all values [in constant time] for queries.

The generation of the cube can be done either with a BFS[1], or using Dynamic Programming with the following recurrence relation:

$$\text{grid}[x] = \begin{cases} 1 & \text{if } x \text{ is a power of two} \\ 1 + \min(\text{grid}[x - 2^k], \text{grid}[2^{k+1} - x]) & \text{otherwise} \end{cases}$$

where $k = \lfloor \log_2(x) \rfloor$.

We now have generated the entire array in linear time, but this is pointless if we naively sum the cube. Luckily for us, the summation of the grid can be sped up by using the following formula:

[1] `http://en.wikipedia.org/wiki/Breadth-first_search`

$$\text{Total} = 3 \times N^2 \times \text{ sum(grid)}$$

3 Travelling Fred

Travelling Fred is a problem in graph theory. The first step (and the one which dominates most of the time) is finding the shortest distance between the T cities you have to travel between. You then follows this with brute force solution of every path visiting the T cities.

Say we already know the shortest distance from city a to b, and it is $\mathbf{dist}(a, b)$ (this can equal "infinity" if there is not a path). We then want to find the shortest distance that starts and ends at city 0, but also visits cities $1, 2, \ldots, T-1$. So we create a list $path = [1, 2, \ldots, T-1]$ and for each permutation you calculate

$$\begin{aligned} d_{path\_perm} = &\mathbf{dist}(0, path[0]) + \mathbf{dist}(path[0], path[1]) + \mathbf{dist}(path[1], path[2]) \\ &+ \cdots + \mathbf{dist}(path[T-3], path[T-2]) + \mathbf{dist}(path[T-2], 0) \end{aligned}$$

The answer is then the smallest d_{path_perm}. This takes $O(T!)$ time.

To calculate $\mathbf{dist}(a, b)$ we can do Floyd-Warshall[2] which takes $O(C^3)$ time. However this will only score 80%, since we end up calculating the distance between all C cities, instead of just the first T cities. Instead we do a Breadth-First Search (BFS)[3] starting at each city which we must travel to. This takes $O(CT)$ time. In total the algorithm takes $O(CT + T!)$ and will score 100%

[2] `http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm`
[3] `http://en.wikipedia.org/wiki/Breadth-first_search`