SACO 2007 Day 1 Solutions

1 Protecting The Castle Aaaaaagggh

There is only one solution to this problem and that is to simulate the delivery of the cannons by selecting where to place them as they come in, keeping running totals of the total number of cannons at each defence point. At the end of the simulation, determine the minimum and maximum number of cannons and the answer is their difference.

2 Round Garden

We know that $S_i \leq C_i$. A brute-force solution therefore generates all possible S_i values. Each combination is checked by calculating what the C_i values would be given these S_i and compares them to those given. Once a match is found we know that must be the answer. This should get 50%, but as soon as N increases this solution will be very slow.

All arithmetic on i that follows is done modulo N, e.g. C_i refers to $C_{i \bmod N}$. $C_{i+1} - C_i = S_{i+M+1} - S_i$. Let $\delta_i = S_{i+1} - S_i$. We then iteratively calculate $\alpha_{i+M} = \alpha_i + \delta_i$ by first setting $\alpha_1 = 0$. Given the α values, we can then calculate $S_1 = \frac{C_i - \sum_{i=1}^{M} \alpha_i}{M}$. From here we can calculate all values $S_i = S_1 + \alpha_i$.

3 Longest Prefix

A brute-force solution finds the longest prefix shared between W_i and W_j for all i, j and then chooses the longest as the answer. Notice that if you add the restriction $i < j$ then you will still compare all words while halving the number of checks. This solution will get you 50%.

There are several solutions which will get you 100%. Probably the easiest of these involves first sorting the list of words. You then only have to compare successive words, W_i and W_{i+1}. Another solution is to generate all possible prefixes of a word and store them in a hash table (dictionary in Python or HashMap in Java). Before adding a prefix you check to see if it's already in the hash table and if it is compare it to the current longest common prefix found.

4 The Knights Who Say Ni

The x, y-coordinates in the 50% constraints are small enough to generate a full grid of the minimum number of moves required to get from any point to $(0, 0)$. The grid can be generated by a breadth-first search (BFS).

We know that getting to $(0, 0)$ requires 0 moves. We therefore know that the points $(1, 2)$, $(2, 1)$ and the other six movements each require 1 move. From each of those points with a shortest distance of 1, we can take another step in each direction for a shortest path of 2. However, some moves will bring us back to a point we have already found a shorter distance for and we therefore discard such moves. We continue this until the grid is full and then use the grid to find the distances for each knight.

To get 100%, a lot of scribbling on paper is required to make some key observations and work out some formulae. The first thing to note is that the grid is symetrical along the x, y axis and the lines $y = \pm x$. You can therefore convert all points (x, y) into an equivalent point such that the new $0 \leq y \leq x$.

The magic formula is:

$$f(x, y) = \begin{cases} 2\lfloor \frac{y-\delta}{3} \rfloor + \delta & \text{if } y > \delta \\ \delta - 2\lfloor \frac{\delta - y}{4} \rfloor & \text{otherwise} \end{cases}$$

where $\delta = x - y$.

5 Trojan Badger

We use a breadth-first search to find the distance between the starting state and ending state. For each position there are three different possible states: standing upright, lying northwards, or lying eastwards (which we will denote $\mathcal{U}$, $\mathcal{N}$ and $\mathcal{E}$ respectively). So are states look like this: $(x, y, \mathcal{U})$. For $\mathcal{N}$, we will record the southernmost of the two points, and for $\mathcal{E}$, the westernmost.

5.1 Neighbouring states

Now we need a way to calculate neighbouring states from a given state (i.e. the states that you can get to in one move from the given state). A bit of scribbling shows us that:

- The neighbours of state $(x, y, \mathcal{U})$ are $(x, y{+}1, \mathcal{N})$, $(x, y{-}2, \mathcal{N})$, $(x{+}1, y, \mathcal{E})$, and $(x - 2, y, \mathcal{E})$.
- The neighbours of state $(x, y, \mathcal{N})$ are $(x + 1, y, \mathcal{N})$ and $(x - 1, y, \mathcal{N})$ (by rolling), and $(x, y - 1, \mathcal{U})$ and $(x, y + 2, \mathcal{U})$ (by standing up).
- The neighbours of state $(x, y, \mathcal{E})$ are analogous to those of $(x, y, \mathcal{N})$: they are $(y{+}1, x, \mathcal{E})$ and $(y{-}1, x, \mathcal{E})$ (by rolling), and $(x{-}1, y, \mathcal{U})$ and $(x{+}2, y, \mathcal{U})$ (by standing up).

5.2 Breadth-first search

Breadth-first search is an algorithm for finding the shortest path between two states if the distance between any two neighbouring states is the same (in this problem, to get from one state to a neighbouring always takes one move).

Let the starting position be (x_S, y_S), and the ending position be (x_F, y_F). Then the start state is $(x_S, y_S, \mathcal{U})$ and the end state is $(x_F, y_F, \mathcal{U})$.

1. Create a queue.
2. Add the start state to the queue; mark it as visited, and record its distance as zero.
3. If the queue is empty, there is no path from the start to the end state, so exit.
4. Remove a state from the front of the queue.
   - If it is the end state, the length of the path is the distance you recorded, so exit.
   - Otherwise, for each valid unvisited neighbour (i.e. neighbours which are not on obstacles or standing on a trap), mark it visited, and record its distance as the distance of the current state plus 1.
5. Go to step 3.