SACO 2010 Day 1 Solutions

SACO Scientific Committee

1 Missiles

1.1 Partial solutions

The brute-force solution is to simply try every possible key. To check if a key is valid, subtract it from the encrypted code, and then check whether the remembered portion appears in the code.

This will work on short codes with any key, but to get 50% one must also solve cases where the code is long but the key is small. To solve these, simply test the small keys ($-10 \leq \text{key} \leq 10$) first.

1.2 Full solution

The essential idea for a full solution is that the differences between adjacent numbers are preserved by the encryption. To find the remembered portion without knowing the key, we look for the sequence `64 31 -21` in the sequence of differences of the encrypted key. Once we have found where the remembered portion appears in the encrypted code, we can find the difference between the two to obtain the key.

2 Bentopia

This problem asks for any subpath of Carl's path that still starts and ends in the same rooms, but visits each room at most once. There are several different solutions for the problem each with varying difficulty and score.

2.1 Brute force

This solution works by generating all possible subpaths of Carl's path until finds one that matches the constraints. There are N rooms in the path, so this algorithm can take at most $O(2^N)$ time to find the solution. There are several techniques that can be used to optimise this algorithm. For instance, a lookup table can be used to keep track of rooms that have been visited in a current path. Another optimisation that works well is to

Even though the solution is particularly fast on random test cases, the worst case performance still remains. This solution should score 30%.

2.2 Naïve Cycle Detection

The key observation that a valid path from the start to the end must not contain any cycles. A cycle is a path that starts and ends in the same room. We can fix a path that contains a cycle by simply deleting the other rooms in the cycle. For example, consider the path:

$$1 \rightarrow \mathbf{2} \rightarrow \mathbf{3} \rightarrow \mathbf{4} \rightarrow \mathbf{5} \rightarrow \mathbf{2} \rightarrow 6$$

We can see that there is a cycle from second item in the path to the second last item. We can fix this by deleting all the elements in between to obtain:

$$1 \rightarrow 2 \rightarrow 6$$

The key observation that must be made is that if a path from the start to the end rooms contains a cycle, then it is obviously not a correct answer. We can easily fix this by deleting the cycle.

We can obtain an answer by deleting all cycles in Carl's original path. A naïve method represents Carl's path as a list(array in Java/C++) of rooms. We then test each pair of rooms in the path. If the rooms are the same then there is a cycle between them and we remove it by deleting the elements in the cycle. The resulting elements list will always be a valid solution.

This solution has a worst case of $O(N^2)$, because each pair of elements may need to be tested. Another problem is that deleting may take up to $O(N^2)$ time to perform all deletions. Although this can be remedied easily by deleting at the end of the list only, it is still too slow and will score 60%.

2.3 Optimal Solution

The major problem is that it takes $O(N^2)$ time to test all pairs. We remedy this by using a lookup table to store whether we have visited a particular room. Scanning through the path from start to finish, if we encounter a room that we have not visited, we update the lookup table. If we have seen the room before, we simply delete from the end of list until we find the duplicate entry.

This solution runs in $O(N)$ worst case.

3 Seating

This solution is a form of the linear assigment problem which can be solved in $\mathcal{O}(N^3M^3)$ using the Hungarian algorithm. This algorithm is out of the scope of the SACO and is far to slow for this problem as NM can go up to 200 000.

In order to approach this problem, you have to implement a heuristic in order to come up with a faster approximate solution. There are various general approaches which can be tried.

3.1 Random Positioning

Randomly assigning positions does a very bad job on average. This can be slightly improved by running multiple times and taking the arrangement with highest score, but even this is unlikely to score more than 10% due to the exponential form of the scoring function.

3.2 Greedy

Probably, the most obvious heuristic is to greedily assign each child to the free position that is furthest from their initial condition. This does well, but the naive solution runs in quadratic time, making it unsuitable for the larger test cases. A well implemented greedy would score around 50%.

The greedy method can be greatly improved for dense cases by assigning the children in an order. Children who initially sat near the centre of the grid are assigned first and childen in the corners last. This helps to even out the resulting distribution by avoiding cases where only a few children are moved very far from their initial positions. This solution scores 65%.

3.3 Geometric Transformations

Certain geometric transformations, such as as reflection and rotation, have the property of displacing the children fairly far from their original positions. Swapping the left and right halves of the grid and then the top and bottom halves is notable in that it has the property of displacing every child the same distance and it obtains the largest score of solution which do this.

Alone, geometric tranformations get around 30% due to their failure to take advantage of open space, but the reflection and half-swapping algorithms can be sucessfully modified to score well by 'pushing the children to the corners'. This involves dividing the grid into quadrants and sorting the children by distance to the corners and then packing the children into the corners in order of proximity. This achieves 70%.

3.4 Case by Case Analysis

In order to beat the solutions found by the Scientific Committee it is advantageous to exploit special cases in the input data.

The most obvious statistic to exploit is the differing densities of the grids. Certain algorithms will be suited to dense grids, while others will handle sparse grids better. In particular, one of the test cases has every seat occupied by a child.

Then certain individual cases with a special structure. In some cases the children are clumped in the centre or all along the one side and there is a linear case where the grid has only one column.

4 Funroll

This problem is an example of Dynamic Programming. Dynamic Programming, for those of you that are not familiar with it, is a technique to solve a category of problems efficiently by caching solutions to subproblems.

A key observation that will help any solution is that we do not need to explicitly store the Funrollers to test for symmetry. We only need to calculate the total number of all valid Funrollers and subtract out the number of symmetric Funrollers.

Suppose we called the number of all Funrollers(mod 4993) that can be made out of a **total** T tracks from a starting height of H, as:

$$\text{Total} = F(T, H)$$

It is easy to see that $F(2N, H)$ will be give us the total number of Funrollers including the symmetric ones: The only way we can get a valid solution is to use equal numbers of up tracks and down tracks.

Finally, the number of symmetric Funrollers can be calculated as:

$$\text{Symmetric} = \sum_{i=0}^{N} F(N, i) \pmod{4993}$$

This gives us a final answer of:

$$\text{Answer} = F(2N, H) - \sum_{i=0}^{N} F(N, i) \pmod{4993}$$

Our only problem now is to calculate $F(T, H)$.

4.1 Brute Force

We can calculate $F(T, H)$ by a simple recursive formula:

$$F(T,H) = \begin{cases} 0, & \text{if } H < 0 \\ 0, & \text{if } H > T \\ 1, & \text{if } N = 0 \ \& \ H = 0 \\ F(T-1, H-1) + F(T-1, H+1), & \text{otherwise} \end{cases}$$

Essentially, this formula allows to enumerate all Funrollers and discard invalid ones. This solution is very poor as it has a running time of $O(2^{2N})$. This algorithm scores 30%.

4.2 Memoisation

We can improve the above algorithm substantially by noticing that we make a large number of calls to F. We can stop the algorithm from doing this by caching the results for each $F(T, H)$ in array or hashtable. This cuts the running time down to $O(N^2)$. A particularly efficient algorithm may score 100%, but the overhead of recursion means that solutions may score only 90%.

4.3 Dynamic Programming

In order to be guaranteed 100%, we need to use Dynamic Programming. We keep a two dimensional array to store the values. By analysing the recurrence of F, we notice that answers in the array for T tracks can computed directly from the answers in the array for $T - 1$. All we need to do is initialise an array with starting at $T = 0, H = 0$ and working back until $T = 2N$. This method is several times faster as it requires no recursion, although it also runs in $O(N^2)$. This method will score 100%.