SACO 2008 Day 1 Solutions

SACO Scientific Committee

1 Cashier

Cashier required to find the largest K consecutive numbers in a list of N numbers. A brute force takes each starting position i from 0 to $N - K$ (since otherwise the end would not fit) and sums from i to $i + K$. This is inefficient, however, and is expected to score 50%, as you sum certain subsets many times.

To get 100% you need to consider the K elements being summed as a K-width "window" with the starting part moving along from 0 to $N - K$ as the end moves from K to N. For the initial position you have to sum the entire window, but as the window moves along one position, the sum decreases by the element that was its first position in the previous step, and add the element that is the last in the current step. This brings the order of the solution down from $O(NK)$ to $O(N)$.

2 Spring Cleaning

One way to solve the task is to iterate through all heights and count the contiguous sections of boxes (a contiguous section is a section with no gaps and that does not have a box immediately to the left or right of it) at each height. To count the sections you only need to find the start of each one. A new contiguous section at height h starts when $H_i \geq h$ and $H_{i-1} < h$. This solution should get 25%.

Another solution is to detonate the highest pile enough times to bring it down to the level of its tallest neighbour. This pile can then be merged with its tallest neighbour, as any detonation that affects one will always affect the other. An implementation which does a linear search for the tallest pile at each iteration is expected get only 25%, but one which sorts the piles by height is expected to get 70%.

The optimal solution is a more efficient implementation of the first one. It uses the fact that if $H_i - H_{i-1} > 0$, then $H_i - H_{i-1}$ contiguous sections start at pile i. The answer is just the sum of these over the range. This solution runs in linear time and is expected to get 100%.

3 Hanoi Reconstruction

This problem was used in both divisions, with lower constraints for the Future Stars.

3.1 Future Stars Division

The brute force solution is the 100% solution for the juniors. Since N is guaranteed to be less than or equal to 20, there is no need to add additional optimisations. The following is an explanation of the algorithm

To move disk N to the destination tower, we must first move the $N-1$ disks above it to the helper tower (the one that is neither the destination nor the source), then move disk N to its destination, then move the $N-1$ disks back on top of it. To move the $N-1$ disks above it, we recursively call the move function. This will move the $N-2$ disks to the helper tower for that movement, then move disk $N-1$ to its destination, then move the $N-2$ disks above it to the destination. This recursion continues until you only need to move a single disk, which is a trivial operation.

3.2 Open Division

3.2.1 50% Solution

The brute force solution is the same as the 100% solution for the junior version of this problem. This is in $O(T)$, but since T grows as $O(2^N)$, it can easily time out when simulating cases for N greater than 20.

3.2.2 100% Solution

The better solution is to optimise the recursive solution slightly. First off we have to notice that, in order to move the bottom disk of an N disk tower to it's final position we require $2^N - 1$ moves. So, at each stage of the recursive solution, we can check that we have enough moves remaining. If we do, we can save a lot of time by just putting the bottom disk at its correct position and moving all the higher disks to the intermediate helper in one go. If not, we just go do the original solution. This solution grows as fast as $O(N)$, which is more than sufficient for 100%.

4 Visiting Grandma

The 30% solution is to recursively generate every route. Along a route you can visit a town at most twice: once with a cookie and once without a cookie. So in the recursive function you keep track of whether you have passed through a cookie town. If you have got a cookie, you only recurse into towns that you have visited with a cookie. If you have not got a cookie, you recurse into any

town you have not visited. When you are in town N with a cookie you do one of the following:

- if the shortest route distance > current route distance, update the shortest distance to the current and set the path count to 1.
- if the shortest route distance == current route distance, increase the path count by 1.

The optimal solution can be solved using a modified version of the Dijkstra algorithm. The modifications to Dijkstra are how to force a path through a cookie town and how to count the number of routes.

Dijkstra's algorithm works by processing the graph using a priority queue sorted by the current distance traveled. Initially the starting town is added to the queue with distance zero. Then while the queue is not empty, the front is removed and processed. Any unprocessed neighbour's of the current town are added to the queue with the current distance + the distance between the current town and the neighbour town.

There are two ways to force a path through a cookie town. The first is to modify the input graph. You do this by creating a copy of the graph, and then add directed edges of weight 0 from every cookie town to its copy. You also delete every edge from a cookie town to every other town in the original graph. Dijkstra then terminates when it visits grandma's town in the copied graph. For example if your cookie town is town 2 and your input graph is represented by the matrix

$$\begin{matrix} \infty & 1 & 2 \\ 1 & \infty & 1 \\ 2 & 1 & \infty \end{matrix}$$

Then it will get transformed to

$$\begin{matrix} \infty & 1 & 2 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty \\ 2 & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 1 & 2 \\ \infty & \infty & \infty & 1 & \infty & 1 \\ \infty & \infty & \infty & 2 & 1 & \infty \end{matrix}$$

where ∞ represents no edge. The other method is to add whether you have a cookie to the path's queue "state". This requires more code, but uses less memory.

To count the number of paths we keep track of the number of paths to any town. Then when processing a neighbour in the Dijkstra loop, if the distance being pushed onto the queue is equal to the neighbour's current distance, we increase the path count of the neighbour by the path count of the current town. If the distance is less, we set the path count of the neighbour to the current path count.

This works because the first time a town comes to the front of queue, the shortest path to it has already been found, and thus we never find a shorter path to a node after it has been removed from the queue.