SACO 2008 Day 2 Solutions

SACO Scientific Committee

1 Factor

Notice that the numbers q and r are the roots of $ax^2 + bx + c$. They can be found using the quadratic formula

$$q = \frac{-b - \sqrt{\Delta}}{2a}$$

and

$$r = \frac{-b + \sqrt{\Delta}}{2b},$$

where $\Delta = b^2 - 4ac$. To find p, multiply out the factorised form:

$$p(x - q)(x - r) = px^2 - p(q + r)x + pqr.$$

Comparing this to the original quadratic polynomial $ax^2 + bx + c$ we see that $p = a$.

The tricky part of the problem was to handle the cases where the quadratic polynomial couldn't be factorised. This can happen in three ways:

The first case is when $\Delta < 0$. In this case, there are no real roots (you can't take the square root of a negative number) and so the quadratic polynomial cannot be factorised.

The second case is when Δ is not a perfect square. In this case $\sqrt{\Delta}$ is an irrational number (e.g. $\sqrt{2} = 1.141\cdots$) and this makes the roots irrational. Since q and r are required to be integers, the quadratic polynomial cannot be factorised. The easiest way to check whether Δ is a perfect square or not is to take the square root, cast it to an integer and then square it to see if you get back the original Δ.

The final case is when the numerator in the quadratic formula is not a multiple of $2a$. In this case the roots are not integers and so, again, the quadratic polynomial cannot be factorised.

2 Popular Room

Poproom required us to find the most common element within a list (the mode). The problem's brute force solution of going through the list and incrementing a count for each element has its drawbacks, mainly in terms of wasting memory or needing to do inefficient lookups, depending on the data structure used to represent the rooms list. This method should get 50% of the marks.

Our correct solution was to sort the list in place, using only a constant amount of additional memory, and then to pass through the list, keeping a running count of only the current element that is being processed, as well as the best current solution. This would only require a single pass, scoring the full 100% of the marks.

3 Hotel

Hotel required you to find the maximum number of guests in the hotel at a time. A variety of brute force solutions exist. The one we thought most plausible was to, for each point in time, to check if each guest was in the hotel at that time and if so, increment some counter, keeping a tab of when the most people were there. This is slow however, since it has order O(N^2) due to the lookup taking linear time, and there being N guests. The optimal solution was to make a single list of arrivals and departures and the time that it occurs, sort this list by time, and then simulate the hotel for all the guests. This is only O(N), getting 100%.

4 Containment

It is very easy to get an incorrect solution for this problem, being mislead into believing that a greedy solution is correct. A typical greedy solution places the first block starting at the first magnet and loops around the ring continuously find the next uncovered magnet and adding a new block to cover that and any magnets in the next $K - 2$ locations.

To see why this is wrong, just try it on the sample case (`o` represents a magnet, `-` represents a blank; $K = 2$):

```
o---o--o--o
```

The greedy fails on this by using three magnets. With a little bit of effort we can get a "more correct" solution that will get the correct answer for many more test cases. One idea is to search for the location for which a block would cover the most magnets and greedily place blocks from there the above algorithm. However, this is still incorrect.

A brute force solution which loops through all possible permutations of the block locations, finding the permutation using the least blocks is $O(N!)$ and expected to score 30%. It should be clear that this solution is correct.

With a little insight into the problem we can improve on this solution by noticing that many permutations use overlapping blocks. We can alter the incorrect greedy solution as follows. Instead of pre-selecting a starting position for the first block, loop through all possible starting positions and assign the blocks greedily. This solution is $O(NR)$ and is expected to get 70%.

To get the full 100% you have to take one more step. Notice that it is not worth starting at a position without any magnets, instead starting only at the magnet positions. This is $O(NK)$.

5 Lasers

Lasers is a competitive task — the idea is to find a better solution than your competitors. For this reason, we chose a problem for which there is no efficient optimal solution known. The approach is to use heuristics. This solution will describe a number of possible heuristics that can be used to improve the estimate for the lowest total cost, but it is by no means exhaustive. There are many other tricks that can be used, but of course the time remaining in the contest is the constraint on how many you can implement.

The first approach is to use a greedy algorithm. A good starting greedy algorithm works by greedily assigning merchandise that is the farthest away from its closest robot. In other words, for each item of merchandise, sort all the robots by the distance that they are from that item. Then you know the closest robot to each item. Call this distance d_i, where i indexes the items. Find the maximum of the values d_i, and give the corresponding robot a laser that can just scan that item. Now repeat this, making sure that when you find the maximum, you do not consider items that are already being scanned. At the end, you should have given the robots lasers such that all the items are scanned. This solution is roughly $O(MN \log N)$, and so is fast enough to run within the contest time.

Although this is a valid solution and it can be submitted, there are optimizations to be made. One optimization is, after the greedy algorithm above has been done, to iterate through each robot and consider the outer-most item that it is scanning. Then run through all the other robots and check if the total cost is reduced by making that robot scan the item instead. This optimization does not influence the overall complexity.

What has been described so far is a good start, and should ensure that you do well against competitors. To make sure you have the best solution, however, it helps to investigate some of the test cases:

- *Cases 1 and 2:*

  These are small enough to do by hand or to brute force and get the optimal answer. Note that coding and debugging a brute force algorithm might take more time than it's worth, so it's probably better to run the greedy algorithm described and tweak the results by hand.

- *Cases 3, 4, 5, 6:*

  These cases consist of randomly generated points, but are far too large to draw and work out by hand, or for a brute force algorithm. The best approach here would be to rely on the greedy algorithm described above.

- *Case 7:* This case has a lot of randomly scattered robots, each with a number of items at an equal distance around them. This means that the items are arranged in circles, and at the center of each circle is a robot. Not many of the circles overlap. The input data is in a random order to make it hard to see this pattern when investigating the file by hand.

  The greedy algorithm does well in this case, but there is a better solution. The idea is to use a clustering algorithm, such as the k-means clustering algorithm. A clustering algorithm takes as its input a set of points in the plane and tries to assign each point to a cluster. The k-means algorithm does this by first dividing the points into arbitrary clusters. Then, calculate the centroid of each cluster. A centroid is the point with the cluster's mean x-value and its mean y-value. For each centroid, find all the points that are closer to that centroid than to any other cluster's centroid. This set of points will form the new cluster. This process can be repeated with the new set of clusters. Repeating enough times will yield a good estimate for the clusters that the points form.

  For this test case, then, a good approach is to run the clustering algorithm on the items of merchandise. Then, for each cluster, find a single robot that can scan the entire cluster.

- *Cases 8, 9:*

  The greedy algorithm works by assigning each item to the nearest robot. This does not work so well if you have a large cluster of items, with one robot in the center and a number of robots around the edge of the cluster. The better solution would be to make the central robot scan the entire cluster, while the robots around the edges scan nothing. Cases 8 and 9 each have a large number of such clusters.

  The solution is similar to the one for test case 7. A clustering algorithm can be used to detect clusters of items. Afterwards assign one robot to scan each cluster entirely. Loop through all robots and see which one scans the entire cluster with the lowest cost. All other robots do not scan anything. As usual, modifications can be made to this algorithm to make it work slightly better.

- *Case 10:*

  The greedy algorithm gets an answer for this case that is particularly far from the optimal. Investigating the case by hand will reveal that all the robots and items of merchandise lie on a straight line. On the line, the robots and items are positioned in the following repeating pattern:

  $$o \quad x \quad o\,o \quad x \quad o\,o \quad x \quad o\,o \quad x \quad o$$

  Here an x is a robot and an o is an item. The greedy algorithm would make each robot scan the two items nearest to it. A much better solution is for every second robot to scan the four items nearest to it, and the rest to scan nothing. Once you see this, it is quick to write a program to generate the output.

6 Disgusting Banquet

6.1 Brute force

The simplest solution is to generate all 2^N possible eating/bathroom schedules, and counting the number of bathroom visits and total tastiness for each schedule. This is very slow—it would take more than a 10^{15000} years to solve the largest test run—and it should score 30%.

6.1.1 Compression

A little bit of cunning allows one to improve this solution slightly without much more work. It is fairly easy to see that one will never go to the bathroom or leave the bathroom between two positive or two negative courses (it's always better to postpone in these cases, and get more tasty courses or skip more nasty ones). Thus one can merge adjacent courses of the same sign, to reduce the problem size (in some cases).

Combining this with the brute force algorithm should score 40%.

6.2 Dynamic programming

A much better solution is to use 2-dimensional dynamic programming. The first dimension is the number of courses, and the other is the remaining number of bathrooms visits allowed. This runs in $\mathrm{O}(NK)$ time, and should get 60%.

The trick of compression works here too, to get around 70%.

6.3 Greedy

The optimal solution (that we know of) is a greedy one. First, we compress the input into intervals of positive and negative courses. Then, as long there are too many bathroom visits, we find the interval that makes the least difference to the total tastiness (i.e., the one with the smallest absolute value) and merge it with its two neighbours. This reduces the number of bathroom visits by one: if it was positive and its two neighbours were negative, we now have one negative interval; otherwise it was negative and its neighbours were positive, and we now have one positive interval.

This can be implemented by keeping the intervals in a list (or array, or linked list). However, a much better solution is to store the intervals in a data structure which allows one to find the interval with the smallest absolute value in sublinear time. Either a heap or a balanced binary tree allow $\mathrm{O}(\log N)$ access to the smallest element, and thus the total is around $\mathrm{O}(N \log N)$. This solution should score 100%.