Editorial for the NZPC 2017

Written by Malcolm Corney, Darcy Best and Bill Rogers

Problem Statements

Problem A: Angles (Simple Math - 3 points)

For each test case, read in the three angle values and total them. If the total value is not 180 output `Check`, otherwise output `Seems OK`.

Problem B: Golf Croquet (Simulation, careful - 3 points)

This is a straightforward simulation problem with care needed to detect the end of a game.

Read in the team names. The first team read in gets the first stroke. Set up a Boolean to indicate if it is the first team's turn. Work through the strokes one at a time adding points to a score total for the team who earns the points based on which team made the stroke. Flip the team Boolean so the other team makes the next stroke.

Check if the score of either team is ≥ 7 . If so, reset the score to 7. A D stroke may have made the score 8 but the maximum allowed score is 7.

Do the output printing the teams and their score in order, then the status of the game.

Problem C: Fitness (Simulation - 3 points)

Another simulation, this time utilisation of the modulus operator will help.

To keep track of the fitness stations that have been visited, set up a Boolean array or an int array for co unting the number of times the station has been visited. Keep another variable for how many fitness stations have been visited in total.

There are only 8 stations. Read the starting station and subtract 1 to 0-index the station numbers. This m akes moving to the next station easier with modulo arithmetic. Mark the initial station as visited.

While there are more stations to read move clockwise or anticlockwise by the suggested amount. station = (station + 8 +/- step) % 8. Adding 8 keeps the value positive before the modulo operation.

If a station has already been visited keep track of the fact that a station has been visited twice so the p lan can be rejected after reading all of the input. If it has not been visited, mark it visited. Asw each station is visited, output its number.

If after visiting all the stations, a station has been visited twice, or the count of stations visted is le ss than 5, output "reject" after the station numbers.

Problem D: Beautiful Music (3 points)

Starting at position 1 in the input string (a single note gets passed over), calculate the difference betwe en the current note and the previous note. Tones in this music only ascend and there are 7 notes in the sc ale so diff = (note[i] - note[i-1] + 7) % 7.

If the diff is not 2 or 4 or 6, output the failure message, otherwise after getting to the end, output the success message.

Problem E: Byte Me (Array handling - 10 points)

You have to find the byte that has an error and then the bit in that byte that has the error. Bytes are numbered from 1 to N as read in. Bits are numbered 1 to 8 from left to right.

Read each bit value into a 2D array. While reading the bytes, keep track of the parity (odd or even) of the byte and keep a count of the number of odd parity bytes and even parity bytes. The parity with a count of 1 will be the incorrect parity for this transmission. Knowing which byte that is tells us the byte in error.

Read the parity byte.

For each bit column in the 2D array, check the parity of the column, including the parity byte this time. When you find the column that does not match the desired parity, you have found the bit that is causing the error.

Output the results.

Problem F: Musical Chairs (10 points)

Read in the player names. Store them (list/vector) in their initial order for later.

One approach is to store the players in an array and then move them in a circular fashion around the array by the required number of places. After the move, the player at the chair to be removed is taken out of the array/vector and the players in higher indexes shuffled down. When a player is removed, they should also be removed from the list in initial order created at the beginning of the process.

After the moves, output the result. Use the list from the beginning of the process to output the names that remain if required.

Problem G: Letter Count (Character counting - 10 points)

Work through all characters. If the character is alphabetic, update its frequency - there is no difference between upper case and lower case letters.

For each letter in the alphabet - output the upper case letter, a space, a vertical bar, a space, and then the frequency of asterisks.

Problem H: League Tables (Sorting - 10 points)

Read the current value of each team's status for each of the values. Store in an object adding each team to a sortable collection (array or vector).

Read and process each game updating both teams values where necessary.

After all processing sort the teams in the specified order:

- highest total points from wins and draws
- highest goal difference (for - against)
- highest goals for - alphabetical

The sorting is best implemented in a comparator on the object/struct the state is stored in.

Problem I: Planes (Geometry - 30 points)

Planes arrive at the boundary of an airport's air traffic control circle (radius is a problem parameter) at given arrival headings (flying directly towards the centre of the circle) and given times. There is a flight path with 'slots' every 30 seconds. The problem asks you to determine which slots can be reached by an aircraft (inside the control circle) and to allocate the earliest unused slots to aircraft in the order of their presentation in the data set.

The geometry looks intimidating, but symmetry reduces it to finding the perpendicular bisector of the line between the arriving aircraft and a potential slot, then determining the intersection point of that line with the flight path - all linear. The number of potential slots is small for each aircraft, so all can be checked.

Best to work in double precision floats, and the problem guarantees that answers are not close to rounding boundaries.

Problem J: Trees (30 points)

Ascii art.

You are given a binary tree in prefix notation, nodes holding single upper case letters and null links showing as @ signs. The task is to draw the tree using | \ and / characters for links.

Exact layout is described with examples. The best approach is to read the binary tree into a tree structure. Have fields on each node to record subtree heights and widths. Do two passes over the tree. The first pass should calculate space required for each subtree. After the first pass allocate a 2D array of characters of the correct size to hold the tree. A second pass over the tree can place the characters. Then all that remains is to output the array, not forgetting to trim blanks off the end of each line.

Problem K: Packing (Dynamic Programming - 30 points)

This is close to a knapsack problem-except you have two knapsacks!

At any point, you need to know three things: which item you are currently working on, the capacity left in knapsack 1 and the capacity left in knapsack 2. Then for each item, you have 3 choices: (a) put the item in knapsack 1, (b) put the item in knapsack 2 or (c) put the item in neither knapsack.

You can recursively implement exactly the above statement (with Dynamic Programming/memoisation) and you're done!

Problem L: Domino Killing (Simulation - 30 points)

Note that at any point in time, there is at most one domino that may fall over. Simply simulate the process by keeping track of the current domino and which direction the push is coming from.

Problem M: Infinite Trees (Dynamic Programming, Trees, DFS - 100 points)

This problem asks us to check whether two self-similar (possibly infinite) trees have the same structure.

Let's traverse the two trees simultaneously starting at their roots, and at each pair of nodes, make sure that they look the same. Suppose we are at vertices u and v, then their two subtrees look the same (as far as we can tell so far) if they have the same number of children. If they have a different number of children, we may immediately stop and answer "NO". Otherwise, we need to then check whether their children also pairwise look the same. We can do this iteratively (maintaining a queue of pairs to check) or recursively. Essentially we are either DFSing or BFSing the two trees simultaneously. There are two issues that we need to deal with:

1. How do we avoid infinite loops, ie. what happens when we visit a node that we've seen before?
2. How do we make it fast enough given that there are O(NM) possible pairs of nodes?

The first issue is easy to handle. Whenever we process a pair (u, v)(u,v)(□, □), we'll simply remember that said pair has been processed. If we ever encounter the same pair again, we can stop processing this branch since it will have already been checked before.

So, maintaining a dynamic programming table processed[u][v] for each pair of vertices $u \in T_1$ and $v \in T_2$ seems tempting, but this would take O(NM) memory, which is far too much when N, M Ò100, 000. Instead, we can use the fact that if u = v and v = w then u = w (this is the transitive property of equality.) Now, we just need to maintain disjoint sets consisting of each vertex of the two trees, ie. we can use a Disjoint-set data structure instead of a two-dimensional dynamic programming table.

Problem N: Competition Day (Graphs, Shortest Paths - 100 points)

There are two cases that we have to distinguish:

- There is a unique path from F to C. In this case, the answer is "Matt wins."
- The shortest path from F to C is unique. In this case, we need to find the second-shortest path.

Since the graph is relatively small (|V|, |E| Ò8, 000|V|,|E|Ò8,000|▯|, |▯| $\leq 8,000$), we can run Dijkstra many times to determine the answer. On the first run of Dijkstra, we'll find a shortest path from FF▯ to CC▯ and assume that Matt takes this path. We now wish to determine the second-shortest path. Suppose the shortest path that we found contains the edges $e_1, e_2, \ldots, e_k$e1,e2,...,ek▯▯, ▯▯, . . . , ▯▯. We can simply run Dijkstra again kk▯ times, once with each edge in $e_1, \ldots e_k$e1,...ek▯▯, . . . ▯▯ missing. If in every single case CC▯ is unreachable from FF▯, then there is a unique path, and so the answer is "Matt wins." Otherwise, we take the shortest path found over each of these, since it is guaranteed that the paths found differ to the original shortest path (since we removed an edge in it.)

Problem O: School Pairing (100 points)

With problems that require lots of output based on queries, there are a few techniques that come up over and over again. This time, it is Mo's algorithm. I will give a very quick overview of Mo's algorithm here, but to get the full details, you should look it up online (for example, here).

First, sort the queries (intervals) into $\sqrt{n}$ buckets based on their **left** end-point:

- $0 - \sqrt{n}$ in the first bucket
- $\sqrt{n} - 2\sqrt{n}$ in the second bucket
- ...
- $(n - \sqrt{n}) - \sqrt{n}$ in the last bucket

Then we are going to answer the queries in each bucket (we do each bucket independent of the other buckets). Within each bucket, sort the intervals by their **right** end-point. Now process the queries in that order by either increasing or decreasing the size of the interval by one (possibly multiple times) until you arrive at the next interval.

If you already know the answer for an interval [L,R], then the answer for [L-1,R], [L+1,R], [L,R+1] or [L,R-1] should be easy to compute (i.e., $O(1)$).

Complexity-wise (for each bucket):

1. Computing the first interval takes $O(n)$.
2. Adjusting the interval may move the left endpoint by at most $O(\sqrt{n})$ places.
3. Over **all** queries within this bucket, the right end-point only moves to the right. So **overall**, this takes $O(n)$.

Thus, 1 and 3 are $O(n)$ per bucket, and there are $O(\sqrt{n})$ buckets, so in total, $O(n\sqrt{n})$. For 2, there are only M queries and each one takes $O(\sqrt{n})$ time, so that takes $O(M\sqrt{n})$ time.

In total, this is $O(n\sqrt{n} + M\sqrt{n})$.
