Double-Checking the Lists – Solution

By: Alex Walker

This puzzle bears a passing resemblance to an earlier puzzle, *The Jousting Lists*. Actually, the only relevant difference between this puzzle and the former is that the square once marked START now reads RESTART. Since we like following directions, let's RESTART, now following a different knight's tour through the grid of cells:

Given	RESTART	*Singularize*	GEMSTONE
First 4 letters	REST	*Anagram without a T*	GENOMES
Anagram with W	WREST	*Extract an interior letter*	GNOMES
Anagram odd-indexed letters	WET	*Last 4 letters, any order*	SOME
Homophone	WHET	*Change penult. letter to R*	SORE
Change letter #3 to I	WHIT	*Insert a compass point abbr.*	SNORE
Caesar shift letter #2 by 10	WRIT	*Anagram with C*	CRONES
Append a letter	WRITE	*Move letter #1 to home row*	DRONES
Homophone (same length)	RIGHT	*Caesar shift letter #4 by 8*	DROVES
Change letter #5 to a bigram	RIGHTS	*Remove a letter other than #1*	DOVES
5-letter homophone	RITES	*Extract a ≥2-pt. Scrabble letter*	DOES
Anagram with D	DRIEST	*Prepend 2 top row letters*	REDOES
Remove letter #6	DRIES	*Remove letters #2 and #3*	ROES
Anagram without S	DIRE	*Anagram with A*	AROSE
Extract a letter from N–Z	DIE	*Extract the second vowel*	ARSE
Anagram with B	BIDE	*Remove letter #3*	ARE
Change letter #2 to 2 letters[a]	BRIDE	*Append a period 3 elem. sym.*	ARENA
[Previous] & ...	GROOM	*Caesar shift*	RIVER
Pluralize	GROOMS	*Surround with 2 STEED letters*	DRIVERS
3-letter internal substring	ROO	*Make penult. a home row key*	DRIVELS
Unshortened form	KANGAROO	*Extract the letter after E*	DRIVES
Reverse 2-letter substring	OR	*Extract last letter, A-Z order*	DRIES
Insert a vowel after vowel #1	OUR	*Anagram the 4 non-S letters*	RIDE
Make letter #2 a diff. vowel	OAR	*Extract vowel #2*	RID
Remove odd-indexed letters	A	*Change center to EN*	REND
Advance 2 in flavortext	GEMSTONES	*Extract letter #1*	END

[a] ... separated by a shift of 9 letters

This time, the eight 'extracted' letters spell the word REVOLVER.

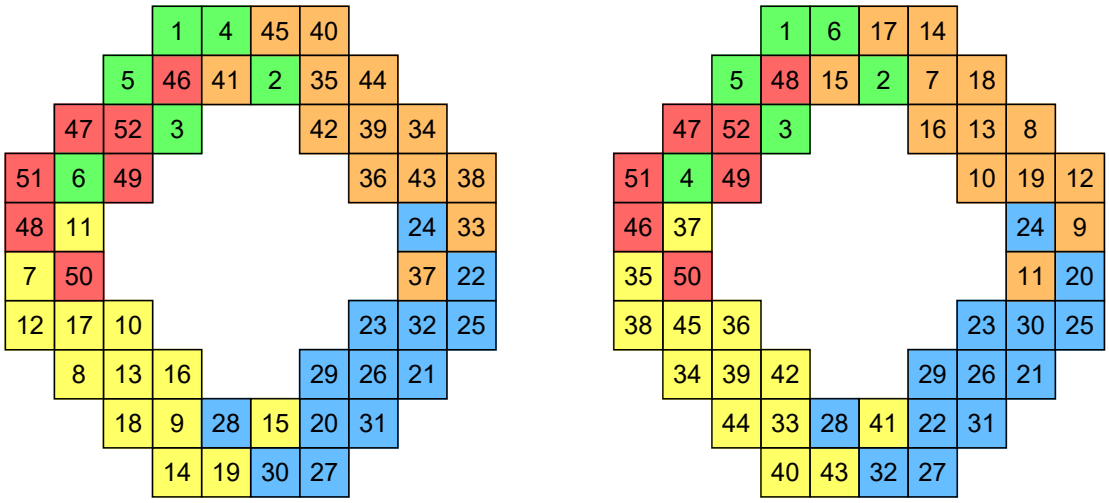
Construction Notes: The idea to create a double feature in *The Jousting Lists* and *Double-Checking the Lists* was inspired by the observation that the feeder answers CAROUSEL and REVOLVER both had length 8 and a semantic connection to rotating things. The answer CAROUSEL evokes horses moving around in a circle, which inspired the knight's tour mechanic and the annular shape of the grid.[1]

There are a few options for grids which approximate the shape of an annulus, but most of the right size (e.g. between 30 and 60 cells) have no knight's tours or have millions of them. The selected grid has **exactly two** closed knight's tours (i.e. knight's tours in which the ending cell is a knight's move away from the starting cell). These tours can be found by looking for Hamiltonian paths on the graph of legal knight moves on the ring. Empirically, the number of knight's tours is roughly dictated by the 'thickness' of the annular ring.

Puzzle construction took approximately ten weeks. One of the challenges with using different knight's tours for the two puzzles was that transformations used consecutively in one solution

[1] Meanwhile, REVOLVER is particularly cute as a callback answer – *revolve* traces back to Latin in the sense of *roll back / return / happen again.*

could occur non-consecutively in the other path. This meant that large sections of the puzzle had to be constructed simultaneously; in the end, construction proceeded in five phases:



Colors indicate the five stages of construction (in rainbow order), while numbers give the knight's tour ordering in *The Jousting Lists* (left) and *Double-Checking the Lists* (right). Each color represents a series of transformations which form a consecutive block in *both* puzzles. For example, the orange section represents transformation steps 33-45 in *The Jousting Lists* and steps 7-19 in the sequel.

As one imagines, creating a fill for this puzzle was incredibly challenging. After some ill-fated attempts at designing fills by hand, I enlisted computer assistance to explore the search space. In essence, one fixes a wordlist W (mine had around 86000 words) and defines a large family of transformations T which map words $w \in W$ to *subsets* $T(w) \subset W$; for example

$$\texttt{anagrams}(\textit{silent}) = \{\textit{enlist}, \textit{inlets}, \textit{silent}, \textit{tinsel}\};$$
$$\texttt{anagrams}(\textit{orange}) = \varnothing.$$

To limit explosive branching during construction, transformations which return too many hits can be overwritten to return no results instead. The final codebase included 273 transformation rules. Each function was batch-computed and cached for efficiency.[2]

Given a sequence of transformations, we can compute (as a breadth-first search) every chain of dictionary words which transforms according to the sequence. We do this for both puzzles at once, often with additional constraints on the departure or destination wordlist. When this fails, we test a different sequence of transformations. (Transformation sequences of length four or five were typical.) We then test every possible sequence of transformations (of some fixed length), and cross our fingers that at least one possibility gives success.

In practice, many sections were constructed by running this process in reverse, i.e. by fixing a destination and applying 'inverse' operations in reverse order. For example, this technique was used in the red section above to efficiently backtrack from the word `END`. To facilitate this process, we essentially compute the inverse image of each $w \in W$ under each transformation T; specifically, we compute $T^{-1}(w) = \{v \in W \mid w \in T(v)\}$ for each T. Like the original transforms, these 'inverse transforms' were batch-computed and cached.

Once enough transformations were added, a different problem emerged – there were too many successes, and it was impossible to review them all manually. To fix this, all transforms were assigned scores based on novelty. For example, the homophone transform was given a score of 10, while the transform 'omit a substring of length 15' was given a score of 1. Sequences of transforms could then be scored and sorted based on their total novelty scores.[3]

[2]For example, to compute anagrams in bulk, one can sort the characters in each dictionary word alphabetically as a hash, then gather on the hashes.

[3]To include options for more 'semantic' transforms, I also needed to create several databases of word transformation rules, including a database of 2500 homophones, 550 X-and-Y phrases, 750 shortened/unshortened word pairs, 8000 antonym pairs, and a large collection of words lying in well-known sequences, like {one, two, three,...} or {snap, crackle, pop}.