1. Intro. Make DLX data to pack a given set of words into a 'Torto' puzzle. In other words, we want to create a 6 × 3 array of characters, where each of the given words can be found by tracing a noncrossing king path.

I've tried to allow arrays of sizes that differ from the 6 × 3 default. But I haven't really tested that.

I learned the basic idea of this program from Ricardo Bittencourt in January 2019. (My first attempt was much, much slower.) (My second attempt was better but still not close.) (So I've pretty much adopted his ideas, lock, stock, and barrel. They have an appealing symmetry.)

In order to save a factor of two, I make the middle transition of the first word start in the top three rows. Furthermore, in order to save another factor of (nearly) two, I don't allow it to start in the rightmost column; and if it starts in the middle column, I don't allow it to move right.

It seems likely that best results will be obtained if the first word is the longest, and if it has lots of characters that aren't shared with other words. The reason is that the middle of this word will tend to be placed first, and many other possibilities will be blocked early on.

```
#define rows 6
#define cols 3    /* you must change encode if rows * cols > 26 */
#define encode(k) ((k) < 10 ? '0' + (k) : (k) < 36 ? 'a' + (k) − 10 : (k) < 62 ? 'A' + (k) − 36 : '?')
#define encodeij(i, j) encode(10 + (i) * cols + (j))

#include <stdio.h>
#include <stdlib.h>
  main(int argc, char *argv[])
  {
    register int i, j, k, l, ll, flag;
    ⟨Process the command line 2⟩;
    ⟨Print the item-name line 3⟩;
    for (k = 1; k < argc; k++) ⟨Print the options for word k − 1 4⟩;
  }
```

2. ⟨Process the command line 2⟩ ≡

```
  if (argc ≡ 1) {
    fprintf(stderr, "Usage:␣%s␣word0␣word1␣...\n", argv[0]);
    exit(−1);
  }
  printf("|␣%s", argv[0]);
  for (k = 1; k < argc; k++) printf("␣%s", argv[k]);
  printf("\n");
```

This code is used in section 1.

3. The primary items are `k!j` for j from 1 to $l-1$, where l is the length of word k.

There are secondary items `a` thru `r`, representing the cells of the array. Their colors will be the characters in those cells.

There also are secondary items `k>j` and `k<x`, which "map" the path for word k. This path is a one-to-one correspondence between the indices 0 thru $l-1$ and the cells where the word is found. The color of `k>j` is x if and only if the color of `k<x` is j.

And there also are secondary items `k/y`, where y is a cell at the southeast of a 2×2 subarray, to prevent diagonal moves within path k from crossing.

Finally, there's a secondary item `flag`, whose color is set to '`*`' if this solution is possibly not canonical under reflections. (It happens if the middle step of the first word is vertical.)

```
⟨Print the item-name line 3⟩ ≡
  for (k = 1; k < argc; k++) {
    for (l = 1; argv[k][l]; l++) printf("␣%c!%c", encode(k − 1), encode(l));
  }
  printf("␣|");
  for (i = 0; i < rows; i++)
    for (j = 0; j < cols; j++) printf("␣%c", encodeij(i, j));
  for (k = 1; k < argc; k++) {
    for (i = 0; i < rows; i++)
      for (j = 0; j < cols; j++) {
        printf("␣%c<%c", encode(k − 1), encodeij(i, j));
        if (i ∧ j) printf("␣%c/%c", encode(k − 1), encodeij(i, j));
      }
    for (l = 0; argv[k][l]; l++) printf("␣%c>%c", encode(k − 1), encode(l));
  }
  printf("␣flag\n");
```

This code is used in section 1.

4. ⟨Print the options for word $k-1$ 4⟩ ≡

```
  {
    for (i = 0; i < rows; i++)
      for (j = 0; j < cols; j++) ⟨Print the options for king moves of word k − 1 that start in cell (i, j) 6⟩;
  }
```

This code is used in section 1.

5. ⟨Print the options for the start position of word $k-1$ 5⟩ ≡

```
  for (i = 0; i < rows; i++)
    for (j = 0; j < cols; j++) printf("#%c␣%c:%c␣%c>0:%c␣%c<%c:0\n", encode(k − 1), encodeij(i, j),
          argv[k][0], encode(k − 1), encodeij(i, j), encode(k − 1), encodeij(i, j));
```

6. ⟨Print the options for king moves of word $k-1$ that start in cell (i,j) 6⟩ ≡

```
for (l = 1; argv[k][l]; l++) {
  flag = 0;
  if (k ≡ 1) ⟨Set flag if we might need to flag this move; continue if (i, j, l) is bad 15⟩;
  if (i) {
    if (j) ⟨Print an option for step l moving northwest 11⟩;
    ⟨Print an option for step l moving straight north 9⟩;
    if (j + 1 < cols ∧ ¬flag) ⟨Print an option for step l moving northeast 14⟩;
  }
  if (j) ⟨Print an option for step l moving straight west 7⟩;
  if (j + 1 < cols ∧ ¬flag) ⟨Print an option for step l moving straight east 8⟩;
  if (i + 1 < rows) {
    if (j) ⟨Print an option for step l moving southwest 13⟩;
    ⟨Print an option for step l moving straight south 10⟩;
    if (j + 1 < cols ∧ ¬flag) ⟨Print an option for step l moving southeast 12⟩;
  }
}
```

This code is used in section 4.

7. ⟨Print an option for step l moving straight west 7⟩ ≡

```
printf("%c!%c␣%c:%c␣%c:%c␣%c>%c:%c␣%c<%c:%c␣%c>%c:%c␣%c<%c:%c\n", encode(k − 1), encode(l),
    encodeij(i, j), argv[k][l − 1], encodeij(i, j − 1), argv[k][l], encode(k − 1), encode(l − 1), encodeij(i, j),
    encode(k − 1), encodeij(i, j), encode(l − 1), encode(k − 1), encode(l), encodeij(i, j − 1), encode(k − 1),
    encodeij(i, j − 1), encode(l));
```

This code is used in section 6.

8. ⟨Print an option for step l moving straight east 8⟩ ≡

```
printf("%c!%c␣%c:%c␣%c:%c␣%c>%c:%c␣%c<%c:%c␣%c>%c:%c␣%c<%c:%c\n", encode(k − 1), encode(l),
    encodeij(i, j), argv[k][l − 1], encodeij(i, j + 1), argv[k][l], encode(k − 1), encode(l − 1), encodeij(i, j),
    encode(k − 1), encodeij(i, j), encode(l − 1), encode(k − 1), encode(l), encodeij(i, j + 1), encode(k − 1),
    encodeij(i, j + 1), encode(l));
```

This code is used in section 6.

9. ⟨Print an option for step l moving straight north 9⟩ ≡

```
printf("%c!%c␣%c:%c␣%c:%c␣%c>%c:%c␣%c<%c:%c␣%c>%c:%c␣%c<%c:%c%s\n", encode(k − 1), encode(l),
    encodeij(i, j), argv[k][l − 1], encodeij(i − 1, j), argv[k][l], encode(k − 1), encode(l − 1), encodeij(i, j),
    encode(k − 1), encodeij(i, j), encode(l − 1), encode(k − 1), encode(l), encodeij(i − 1, j), encode(k − 1),
    encodeij(i − 1, j), encode(l), flag ? "␣flag:*" : "");
```

This code is used in section 6.

10. ⟨Print an option for step l moving straight south 10⟩ ≡

```
printf("%c!%c␣%c:%c␣%c:%c␣%c>%c:%c␣%c<%c:%c␣%c>%c:%c␣%c<%c:%c%s\n", encode(k − 1), encode(l),
    encodeij(i, j), argv[k][l − 1], encodeij(i + 1, j), argv[k][l], encode(k − 1), encode(l − 1), encodeij(i, j),
    encode(k − 1), encodeij(i, j), encode(l − 1), encode(k − 1), encode(l), encodeij(i + 1, j), encode(k − 1),
    encodeij(i + 1, j), encode(l), flag ? "␣flag:*" : "");
```

This code is used in section 6.

11. ⟨ Print an option for step *l* moving northwest 11 ⟩ ≡

```
printf("%c!%c␣%c:%c␣%c:%c␣%c>%c:%c␣%c<%c:%c␣%c>%c:%c␣%c<%c:%c␣%c/%c\n", encode(k − 1),
    encode(l), encodeij(i, j), argv[k][l − 1], encodeij(i − 1, j − 1), argv[k][l], encode(k − 1), encode(l − 1),
    encodeij(i, j), encode(k − 1), encodeij(i, j), encode(l − 1), encode(k − 1), encode(l), encodeij(i − 1,
    j − 1), encode(k − 1), encodeij(i − 1, j − 1), encode(l), encode(k − 1), encodeij(i, j));
```

This code is used in section 6.

12. ⟨ Print an option for step *l* moving southeast 12 ⟩ ≡

```
printf("%c!%c␣%c:%c␣%c:%c␣%c>%c:%c␣%c<%c:%c␣%c>%c:%c␣%c<%c:%c␣%c/%c\n", encode(k − 1),
    encode(l), encodeij(i, j), argv[k][l − 1], encodeij(i + 1, j + 1), argv[k][l], encode(k − 1), encode(l − 1),
    encodeij(i, j), encode(k − 1), encodeij(i, j), encode(l − 1), encode(k − 1), encode(l), encodeij(i + 1,
    j + 1), encode(k − 1), encodeij(i + 1, j + 1), encode(l), encode(k − 1), encodeij(i + 1, j + 1));
```

This code is used in section 6.

13. ⟨ Print an option for step *l* moving southwest 13 ⟩ ≡

```
printf("%c!%c␣%c:%c␣%c:%c␣%c>%c:%c␣%c<%c:%c␣%c>%c:%c␣%c<%c:%c␣%c/%c\n", encode(k − 1),
    encode(l), encodeij(i, j), argv[k][l − 1], encodeij(i + 1, j − 1), argv[k][l], encode(k − 1), encode(l − 1),
    encodeij(i, j), encode(k − 1), encodeij(i, j), encode(l − 1), encode(k − 1), encode(l), encodeij(i + 1,
    j − 1), encode(k − 1), encodeij(i + 1, j − 1), encode(l), encode(k − 1), encodeij(i + 1, j));
```

This code is used in section 6.

14. ⟨ Print an option for step *l* moving northeast 14 ⟩ ≡

```
printf("%c!%c␣%c:%c␣%c:%c␣%c>%c:%c␣%c<%c:%c␣%c>%c:%c␣%c<%c:%c␣%c/%c\n", encode(k − 1),
    encode(l), encodeij(i, j), argv[k][l − 1], encodeij(i − 1, j + 1), argv[k][l], encode(k − 1), encode(l − 1),
    encodeij(i, j), encode(k − 1), encodeij(i, j), encode(l − 1), encode(k − 1), encode(l), encodeij(i − 1,
    j + 1), encode(k − 1), encodeij(i − 1, j + 1), encode(l), encode(k − 1), encodeij(i, j + 1));
```

This code is used in section 6.

15. I assume here that *rows* is even and *cols* is odd.

⟨ Set *flag* if we might need to flag this move; **continue** if (i, j, l) is bad 15 ⟩ ≡

```
{
  for (ll = 1; argv[k][ll]; ll++) ;
  if (l ≡ (ll ≫ 1)) {
    if (i + i ≥ rows ∨ j + j ≥ cols) continue;
    if (j + j ≡ cols − 1) flag = 1;
  }
}
```

This code is used in section 6.

16. Index.

⟨ Print an option for step l moving northeast 14 ⟩ Used in section 6.
⟨ Print an option for step l moving northwest 11 ⟩ Used in section 6.
⟨ Print an option for step l moving southeast 12 ⟩ Used in section 6.
⟨ Print an option for step l moving southwest 13 ⟩ Used in section 6.
⟨ Print an option for step l moving straight east 8 ⟩ Used in section 6.
⟨ Print an option for step l moving straight north 9 ⟩ Used in section 6.
⟨ Print an option for step l moving straight south 10 ⟩ Used in section 6.
⟨ Print an option for step l moving straight west 7 ⟩ Used in section 6.
⟨ Print the item-name line 3 ⟩ Used in section 1.
⟨ Print the options for king moves of word $k-1$ that start in cell (i,j) 6 ⟩ Used in section 4.
⟨ Print the options for the start position of word $k-1$ 5 ⟩
⟨ Print the options for word $k-1$ 4 ⟩ Used in section 1.
⟨ Process the command line 2 ⟩ Used in section 1.
⟨ Set *flag* if we might need to flag this move; **continue** if (i,j,l) is bad 15 ⟩ Used in section 6.

TORTO-DLX