November 24, 2020 at 13:24

1. Intro. This program produces a DLX file that corresponds to the problem of packing a given set of polyominoes into a given two-dimensional box. The output file might be input directly to a DLX-solver; however, it often is edited manually, to customize a particular problem (for example, to avoid producing solutions that are equivalent to each other).

Cells of the box have two coordinates xy, in the range $0 \le x, y < 62$, specified by means of the extended hexadecimal "digits" `0`, `1`, ..., `9`, `a`, `b`, ..., `z`, `A`, `B`, ..., `Z`.

As in DLX format, any line of *stdin* that begins with '`|`' is considered to be a comment.

The first noncomment line specifies the cells of the box. It's a list of pairs xy, where each coordinate is either a single digit or a set of digits enclosed in square brackets. For example, '`[02]b`' specifies two cells, `0b`, `2b`. Brackets may also contain a range of items, with UNIX-like conventions; for instance, '`[0-2][b-b]`' specifies three cells, `0b`, `1b`, `2b`, and '`[1-3][1-4]`' specifies a 3×4 rectangle.

Individual cells may be specified more than once, but they appear just once in the box. For example,

```
[123]2    2[123]
```

specifies a cross of five cells, namely `22` and its four neighbors. The cells of a box needn't be connected.

Cell specifications can optionally be followed by a suffix. For example, '`[12]7suf`' specifies two items named '`17suf`' and '`27suf`'. Such items will be *secondary*.

The other noncomment lines consist of a piece name followed by typical cells of that piece. These typical cells are specified in the same way as the cells of a box.

The typical cells lead to up to 8 "base placements" for a given piece, corresponding to rotations and/or reflections in two-dimensional space. The piece can then be placed by choosing one of its base placements and shifting it by an arbitrary amount, provided that all such cells fit in the box. The base placements themselves need not fit in the box.

All suffixes associated with a cell will be appended to the items generated by that cell. For example, a piece that has typical cells '`00`, `01`, `00!`, `01!`' will generate options for every pair of adjacent cells in the box: When `34` and `44` are present, there will be an option '`34 44 34! 44!`'. If `01!` hadn't been specified, there would have been *two* options, '`34 44 34!`' and '`44 34 44!`'.

Each piece name should be distinguishable from the coordinates of the cells in the box. (For example, a piece should not be named `00` unless cell `00` isn't in the box.) This condition is not fully checked by the program.

A piece that is supposed to occur more than once can be preceded by its multiplicity and a vertical line; for example, one can give its name as '`12|Z`'. (This feature will produce a file that can be handled only by DLX solvers that allow multiplicity.)

Several lines may refer to the same piece. In such cases the placements from each line are combined.

2. OK, here we go.

```
#define bufsize 1024 /* input lines shouldn't be longer than this */
#define maxpieces 100 /* at most this many pieces */
#define maxnodes 100000 /* at most this many elements of lists */
#define maxbases 1000 /* at most this many base placements */
#define maxsuffixes 10 /* at most this many suffixes */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  char buf[bufsize];
  ⟨ Type definitions 9 ⟩;
  ⟨ Global variables 8 ⟩;
  ⟨ Subroutines 3 ⟩;
  main()
  {
    register int i, j, k, p, q, r, t, x, y, dx, dy, xy0, suf;
    register long long xa, ya;
    ⟨ Read the box spec 17 ⟩;
    ⟨ Read the piece specs 24 ⟩;
    ⟨ Output the DLX item-name line 34 ⟩;
    ⟨ Output the DLX options 35 ⟩;
    ⟨ Bid farewell 36 ⟩;
  }
```

3. Low-level operations. I'd like to begin by building up some primitive subroutines that will help to parse the input and to publish the output.

For example, I know that I'll need basic routines for the input and output of radix-62 digits.

```
⟨ Subroutines 3 ⟩ ≡
  int decode(char c)
  {
    if (c ≤ '9') {
      if (c ≥ '0') return c − '0';
    } else if (c ≥ 'a') {
      if (c ≤ 'z') return c + 10 − 'a';
    } else if (c ≥ 'A' ∧ c ≤ 'Z') return c + 36 − 'A';
    if (c ≠ '\n') return −1;
    fprintf(stderr, "Incomplete␣input␣line:␣%s", buf);
    exit(−888);
  }
  char encode(int x)
  {
    if (x < 0) return '-';
    if (x < 10) return '0' + x;
    if (x < 36) return 'a' − 10 + x;
    if (x < 62) return 'A' − 36 + x;
    return '?';
  }
```

See also sections 4, 12, 13, 14, 15, and 30.

This code is used in section 2.

4. I'll also want to decode the specification of a given *set* of digits, starting at position p in *buf*. Subroutine *pdecode* sets the global variable *acc* to a 64-bit number that represents the digit or digits mentioned there. Then it returns the next buffer position, so that I can continue scanning.

```
⟨ Subroutines 3 ⟩ +≡
  int pdecode(register int p)
  {
    register int x;
    if (buf[p] ≠ '[') {
      x = decode(buf[p]);
      if (x ≥ 0) {
        acc = 1_LL ≪ x;
        return p + 1;
      }
      fprintf(stderr, "Illegal␣digit␣at␣position␣%d␣of␣%s", p, buf);
      exit(−2);
    } else ⟨ Decode a bracketed specification 5 ⟩;
  }
```

5. We want to catch illegal syntax such as '`[-5]`', '`[1-]`', '`[3-2]`', '`[1-2-3]`', '`[3--5]`', while allowing '`[7-z32-4A5-5]`', etc. (The latter is equivalent to '`[2-57-A]`'.)

Notice that the empty specification '`[]`' is legal, but useless.

```
⟨ Decode a bracketed specification 5 ⟩ ≡
  {
    register int t, y;
    for (acc = 0, t = x = −1, p++; buf[p] ≠ ']'; p++) {
      if (buf[p] ≡ '\n') {
        fprintf(stderr, "No␣closing␣bracket␣in␣%s", buf);
        exit(−4);
      }
      if (buf[p] ≡ '-') ⟨ Get ready for a range 6 ⟩
      else {
        x = decode(buf[p]);
        if (x < 0) {
          fprintf(stderr, "Illegal␣bracketed␣digit␣at␣position␣%d␣of␣%s", p, buf);
          exit(−3);
        }
        if (t < 0) acc |= 1LL ≪ x;
        else ⟨ Complete the range from t to x 7 ⟩;
      }
    }
    return p + 1;
  }
```

This code is used in section 4.

6.

```
⟨ Get ready for a range 6 ⟩ ≡
  {
    if (x < 0 ∨ buf[p + 1] ≡ ']') {
      fprintf(stderr, "Illegal␣range␣at␣position␣%d␣of␣%s", p, buf);
      exit(−5);
    }
    t = x, x = −1;
  }
```

This code is used in section 5.

7.

```
⟨ Complete the range from t to x 7 ⟩ ≡
  {
    if (x < t) {
      fprintf(stderr, "Decreasing␣range␣at␣position␣%d␣of␣%s", p, buf);
      exit(−6);
    }
    acc |= (1LL ≪ (x + 1)) − (1LL ≪ t);
    t = x = −1;
  }
```

This code is used in section 5.

8.

```
⟨ Global variables 8 ⟩ ≡
  long long acc;      /* accumulated bits representing coordinate numbers */
  long long accx, accy;      /* the bits for each dimension of a partial spec */
```

See also sections 11, 22, and 23.

This code is used in section 2.

9. Data structures. The given box is remembered as a sorted list of cells xy, represented as a linked list of packed integers $(x \ll 8) + y$. The base placements of each piece are also remembered in the same way.

All of the relevant information appears in a structure of type **box**.

```
⟨ Type definitions 9 ⟩ ≡
  typedef struct {
    int list;      /* link to the first of the packed triples xy */
    int size;      /* the number of items in that list */
    int xmin, xmax, ymin, ymax;      /* extreme coordinates */
    int pieceno;      /* the piece, if any, for which this is a base placement */
  } box;
```

See also section 10.

This code is used in section 2.

10. Elements of the linked lists appear in structures of type **node**.

All of the lists will be rather short. So I make no effort to devise methods that are asymptotically efficient as things get infinitely large. My main goal is to have a program that's simple and correct. (And I hope that it will also be easy and fun to read, when I need to refer to it or modify it.)

```
⟨ Type definitions 9 ⟩ +≡
  typedef struct {
    int xy;      /* position data stored in this node */
    int suf;      /* suffix data for this node */
    int link;      /* the next node of the list, if any */
  } node;
```

11. All of the nodes appear in the array *elt*. I allocate it statically, because it doesn't need to be very big.

```
⟨ Global variables 8 ⟩ +≡
  node elt[maxnodes];      /* the nodes */
  int curnode;      /* the last node that has been allocated so far */
  int avail;      /* the stack of recycled nodes */
```

12. Subroutine *getavail* allocates a new node when needed.

```
⟨ Subroutines 3 ⟩ +≡
  int getavail(void)
  {
    register int p = avail;
    if (p) {
      avail = elt[avail].link;
      return p;
    }
    p = ++curnode;
    if (p < maxnodes) return p;
    fprintf(stderr, "Overflow! Recompile me by making maxnodes bigger than %d.\n", maxnodes);
    exit(-666);
  }
```

13. Conversely, *putavail* recycles a list of nodes that are no longer needed.

```
⟨ Subroutines 3 ⟩ +≡
  void putavail(int p)
  {
    register int q;
    if (p) {
      for (q = p; elt[q].link; q = elt[q].link) ;
      elt[q].link = avail;
      avail = p;
    }
  }
```

14. The *insert* routine puts new (x, y) data into the list of *newbox*, unless (x, y) is already present.

```
⟨ Subroutines 3 ⟩ +≡
  void insert(int x, int y, int s)
  {
    register int p, q, r, xy;
    xy = (x ≪ 8) + y;
    for (q = 0, p = newbox.list; p; q = p, p = elt[p].link) {
      if (elt[p].xy ≡ xy) {
        if (elt[p].suf ≡ s) return;      /* nothing to be done */
        if (elt[p].suf > s) break;       /* we've found the insertion point */
      } else if (elt[p].xy > xy) break;      /* we've found the insertion point */
    }
    r = getavail( );
    elt[r].xy = xy, elt[r].suf = s, elt[r].link = p;
    if (q) elt[q].link = r;
    else newbox.list = r;
    newbox.size++;
    if (x < newbox.xmin) newbox.xmin = x;
    if (y < newbox.ymin) newbox.ymin = y;
    if (x > newbox.xmax) newbox.xmax = x;
    if (y > newbox.ymax) newbox.ymax = y;
  }
```

15. Although this program is pretty simple, I do want to watch it in operation before I consider it to be reasonably well debugged. So here's a subroutine that's useful for diagnostic purposes.

```
⟨ Subroutines 3 ⟩ +≡
  void printbox(box *b)
  {
    register int p, x, y;
    fprintf(stderr, "Piece␣%d,␣size␣%d,␣%d..%d␣%d..%d:\n", b→pieceno, b→size, b→xmin, b→xmax, b→ymin,
        b→ymax);
    for (p = b→list; p; p = elt[p].link) {
      x = elt[p].xy ≫ 8, y = elt[p].xy & #ff;
      fprintf(stderr, "␣%c%c%s", encode(x), encode(y), elt[p].suf ? suffix[elt[p].suf − 1] : "");
    }
    fprintf(stderr, "\n");
  }
```

16. Inputting the given box. Now we're ready to look at the *xy* specifications of the box to be filled. As we read them, we remember the cells in the box called *newbox*. Then, for later convenience, we also record them in a three-dimensional array called *occupied*.

17. ⟨ Read the box spec 17 ⟩ ≡

```
while (1) {
  if (¬fgets(buf, bufsize, stdin)) {
    fprintf(stderr, "Input␣file␣ended␣before␣the␣box␣specification!\n");
    exit(−9);
  }
  if (buf[strlen(buf) − 1] ≠ '\n') {
    fprintf(stderr, "Overflow!␣Recompile␣me␣by␣making␣bufsize␣bigger␣than␣%d.\n", bufsize);
    exit(−667);
  }
  printf("|␣%s", buf);      /* all input lines are echoed as DLX comments */
  if (buf[0] ≠ '|') break;
}
p = 0;
⟨ Put the specified cells into newbox, starting at buf[p] 18 ⟩;
givenbox = newbox;
⟨ Set up the occupied table 21 ⟩;
```

This code is used in section 2.

18. This spec-reading code will also be useful later when I'm inputting the typical cells of a piece.

⟨ Put the specified cells into *newbox*, starting at *buf*[*p*] 18 ⟩ ≡

```
newbox.list = newbox.size = 0;
newbox.xmin = newbox.ymin = 62;
newbox.xmax = newbox.ymax = −1;
for ( ; buf[p] ≠ '\n'; p++) {
  if (buf[p] ≠ '␣') ⟨ Scan an xy spec 19 ⟩;
}
```

This code is used in sections 17 and 25.

19. I could make this faster by using bitwise trickery. But what the heck.

⟨ Scan an *xy* spec 19 ⟩ ≡

```
{
  p = pdecode(p), accx = acc;
  p = pdecode(p), accy = acc;
  ⟨ Digest the optional suffix, suf 20 ⟩;
  if (buf[p] ≡ '\n') p−−;      /* we'll reread the newline character */
  for (x = 0, xa = accx; xa; x++, xa ≫= 1)
    if (xa & 1) {
      for (y = 0, ya = accy; ya; y++, ya ≫= 1)
        if (ya & 1) insert(x, y, suf);
    }
}
```

This code is used in section 18.

20. ⟨ Digest the optional suffix, *suf* 20 ⟩ ≡

```
for (q = 0; buf[p + q] ≠ ' ' ∧ buf[p + q] ≠ '\n'; q++) {
  if (q ≡ 6) {
    fprintf(stderr, "Suffix too long, starting at position %d of %s", p, buf);
    exit(−11);
  }
  suffix[scount + 1][q] = buf[p + q];
}
if (q) {
  suffix[scount + 1][q] = 0;
  p += q;
  for (i = 0; ; i++)
    if (strcmp(suffix[i], suffix[scount + 1]) ≡ 0) break;
  if (i > scount) {
    scount++;
    if (scount > maxsuffixes) {
      fprintf(stderr, "Overflow! Recompile me by making maxsuffixes>%d.\n", maxsuffixes);
      exit(−7);
    }
  }
  suf = i + 1;
} else suf = 0;
```

This code is used in section 19.

21. ⟨ Set up the *occupied* table 21 ⟩ ≡

```
for (p = givenbox.list; p; p = elt[p].link) {
  x = elt[p].xy ≫ 8, y = elt[p].xy & #ff;
  occupied[elt[p].suf][x][y] = 1;
}
```

This code is used in section 17.

22. ⟨ Global variables 8 ⟩ +≡

```
box newbox;      /* the current specifications are placed here */
char suffix[maxsuffixes + 1][8];      /* table of suffixes seen */
int scount;      /* this many nonempty suffixes seen */
char occupied[maxsuffixes + 1][64][64];      /* does the box occupy a given cell? */
box givenbox;
int sfxpresent;      /* this many items in givenbox have suffixes */
```

23. Inputting the given pieces. After I've seen the box, the remaining noncomment lines of the input file are similar to the box line, except that they begin with a piece name.

This name can be any string of one to eight nonspace characters allowed by DLX format, followed by a space. It should also not be the same as a position of the box.

I keep a table of the distinct piece names that appear, and their multiplicities.

And of course I also compute and store all of the base placements that correspond to the typical cells that are specified.

```
⟨ Global variables 8 ⟩ +≡
  char names[maxpieces][8];      /* the piece names seen so far */
  int piececount;      /* how many of them are there? */
  char mult[maxpieces][8];      /* what is the multiplicity? */
  char multip[8];      /* current multiplicity */
  box base[maxbases];      /* the base placements seen so far */
  int basecount;      /* how many of them are there? */
```

24. ⟨ Read the piece specs 24 ⟩ ≡

```
  while (1) {
    if (¬fgets(buf, bufsize, stdin)) break;
    if (buf[strlen(buf) − 1] ≠ '\n') {
      fprintf(stderr, "Overflow! Recompile me by making bufsize bigger than %d.\n", bufsize);
      exit(−777);
    }
    printf("| %s", buf);      /* all input lines are echoed as DLX comments */
    if (buf[0] ≡ '|') continue;
    ⟨ Read a piece spec 25 ⟩;
  }
```

This code is used in section 2.

25. ⟨ Read a piece spec 25 ⟩ ≡

```
  ⟨ Read the piece name, and find it in the names table at position k 27 ⟩;
  newbox.pieceno = k;      /* now buf[p] is the space following the name */
  ⟨ Put the specified cells into newbox, starting at buf[p] 18 ⟩;
  ⟨ Normalize the cells of newbox 29 ⟩;
  base[basecount] = newbox;
  ⟨ Create the other base placements equivalent to newbox 31 ⟩;
```

This code is used in section 24.

26. We accept any string of characters followed by '|' as a multiplicity.

27. ⟨ Read the piece name, and find it in the *names* table at position k 27 ⟩ ≡

```
for (p = 0; buf[p] ≠ '\n'; p++)
  if (buf[p] ≡ '|') break;
  else multip[p] = buf[p];
if (buf[p] ≡ '|') multip[p] = '\0', p++;
else p = 0, multip[0] = '1', multip[1] = '\0';
for (q = p; buf[p] ≠ '\n'; p++) {
  if (buf[p] ≡ ' ') break;
  if (buf[p] ≡ '|' ∨ buf[p] ≡ ':') {
    fprintf(stderr, "Illegal character in piece name: %s", buf);
    exit(−8);
  }
}
if (buf[p] ≡ '\n') {
  fprintf(stderr, "(Empty %s is being ignored)\n", p ≡ 0 ? "line" : "piece");
  continue;
}
⟨ Store the name in names[piececount] and check its validity 28 ⟩;
for (k = 0; ; k++)
  if (strncmp(names[k], names[piececount], 8) ≡ 0) break;
if (k ≡ piececount) {      /* it's a new name */
  if (++piececount > maxpieces) {
    fprintf(stderr, "Overflow! Recompile me by making maxpieces bigger than %d.\n", maxpieces);
    exit(−668);
  }
}
if (¬mult[k][0]) strcpy(mult[k], multip);
else if (strcmp(mult[k], multip)) {
  fprintf(stderr, "Inconsistent multiplicities for piece %.8s, %s vs %s!\n", names[k], mult[k],
      multip);
  exit(−6);
}
```

This code is used in section 25.

28. ⟨ Store the name in *names*[*piececount*] and check its validity 28 ⟩ ≡

```
if (p ≡ q ∨ p > q + 8) {
  fprintf(stderr, "Piece name is nonexistent or too long: %s", buf);
  exit(−7);
}
for (j = q; j < p; j++) names[piececount][j − q] = buf[j];
if (p ≡ q + 2) {
  x = decode(names[piececount][0]);
  y = decode(names[piececount][1]);
  if (x ≥ 0 ∧ y ≥ 0 ∧ occupied[0][x][y]) {
    fprintf(stderr, "Piece name conflicts with board position: %s", buf);
    exit(−333);
  }
}
```

This code is used in section 27.

29. It's a good idea to "normalize" the typical cells of a piece, by making the *xmin* and *ymin* fields of *newbox* both zero.

```
⟨ Normalize the cells of newbox 29 ⟩ ≡
  xy0 = (newbox.xmin ≪ 8) + newbox.ymin;
  if (xy0) {
    for (p = newbox.list; p; p = elt[p].link) elt[p].xy −= xy0;
    newbox.xmax −= newbox.xmin, newbox.ymax −= newbox.ymin;
    newbox.xmin = newbox.ymin = 0;
  }
```

This code is used in section 25.

30. Transformations. Now we get to the interesting part of this program, as we try to find all of the base placements that are obtainable from a given set of typical cells.

The method is a simple application of breadth-first search: Starting at the newly created base, we make sure that every elementary transformation of every known placement is also known.

This procedure requires a simple subroutine to check whether or not two placements are equal. We can assume that both placements are normalized, and that both have the same size. Equality testing is easy because the lists have been sorted.

```
⟨ Subroutines 3 ⟩ +≡
  int equality(int b)
  {    /* return 1 if base[b] matches newbox */
    register int p, q;
    for (p = base[b].list, q = newbox.list; p; p = elt[p].link, q = elt[q].link)
      if (elt[p].xy ≠ elt[q].xy ∨ elt[p].suf ≠ elt[q].suf) return 0;
    return 1;
  }
```

31. Just two elementary transformations suffice to generate them all.

```
⟨ Create the other base placements equivalent to newbox 31 ⟩ ≡
  j = basecount, k = basecount + 1;    /* bases j thru k − 1 have been checked */
  while (j < k) {
    ⟨ Set newbox to base[j] transformed by 90° rotation 32 ⟩;
    for (i = basecount; i < k; i++)
      if (equality(i)) break;
    if (i < k) putavail(newbox.list);    /* already known */
    else base[k++] = newbox;    /* we've found a new one */
    ⟨ Set newbox to base[j] transformed by xy transposition 33 ⟩;
    for (i = basecount; i < k; i++)
      if (equality(i)) break;
    if (i < k) putavail(newbox.list);    /* already known */
    else base[k++] = newbox;    /* we've found a new one */
    j++;
  }
  basecount = k;
  if (basecount + 8 > maxbases) {
    fprintf(stderr, "Overflow! Recompile me by making maxbases bigger than %d.\n",
        basecount + 23);
    exit(−669);
  }
```

This code is used in section 25.

32. The first elementary transformation replaces (x, y) by $(y, -x)$. It corresponds to 90-degree rotation about the origin.

```
⟨ Set newbox to base[j] transformed by 90° rotation 32 ⟩ ≡
  newbox.size = newbox.list = 0;
  newbox.xmax = base[j].ymax, t = newbox.ymax = base[j].xmax;
  for (p = base[j].list; p; p = elt[p].link) {
    x = elt[p].xy ≫ 8, y = elt[p].xy & #ff;
    insert(y, t − x, elt[p].suf);
  }
```

This code is used in section 31.

33. The other elementary transformation replaces (x, y) by (y, x). It corresponds to reflection about a diagonal.

```
⟨ Set newbox to base[j] transformed by xy transposition 33 ⟩ ≡
  newbox.size = newbox.list = 0;
  newbox.xmax = base[j].ymax, newbox.ymax = base[j].xmax;
  for (p = base[j].list; p; p = elt[p].link) {
    x = elt[p].xy ≫ 8, y = elt[p].xy & #ff;
    insert(y, x, elt[p].suf);
  }
```

This code is used in section 31.

34. Finishing up. In previous parts of this program, I've terminated abruptly when finding malformed input.

But when everything on *stdin* passes muster, I'm ready to publish all the information that has been gathered.

```
⟨ Output the DLX item-name line 34 ⟩ ≡
  printf("|␣this␣file␣was␣created␣by␣polyomino-dlx␣from␣that␣data\n");
  for (p = givenbox.list; p; p = elt[p].link)
    if (¬elt[p].suf) {
      x = elt[p].xy ≫ 8, y = elt[p].xy & #ff;
      printf("␣%c%c", encode(x), encode(y));
    }
  for (k = 0; k < piececount; k++) {
    if (mult[k][0] ≡ '1' ∧ mult[k][1] ≡ '\0') printf("␣%.8s", names[k]);
    else printf("␣%s|%.8s", mult[k], names[k]);
  }
  if (scount) {
    printf("␣|");
    for (sfxpresent = 0, p = givenbox.list; p; p = elt[p].link)
      if (elt[p].suf) {
        x = elt[p].xy ≫ 8, y = elt[p].xy & #ff, sfxpresent++;
        printf("␣%c%c%s", encode(x), encode(y), suffix[elt[p].suf − 1]);
      }
  }
  printf("\n");
```

This code is used in section 2.

35. ⟨ Output the DLX options 35 ⟩ ≡

```
  for (j = 0; j < basecount; j++) {
    for (dx = givenbox.xmin; dx ≤ givenbox.xmax − base[j].xmax; dx++)
      for (dy = givenbox.ymin; dy ≤ givenbox.ymax − base[j].ymax; dy++) {
        for (p = base[j].list; p; p = elt[p].link) {
          x = elt[p].xy ≫ 8, y = elt[p].xy & #ff;
          if (¬occupied[elt[p].suf][x + dx][y + dy]) break;
        }
        if (¬p) {      /* they're all in the box */
          printf("%.8s", names[base[j].pieceno]);
          for (p = base[j].list; p; p = elt[p].link) {
            x = elt[p].xy ≫ 8, y = elt[p].xy & #ff;
            printf("␣%c%c%s", encode(x + dx), encode(y + dy), elt[p].suf ? suffix[elt[p].suf − 1] : "");
          }
          printf("\n");
        }
      }
  }
```

This code is used in section 2.

36. Finally, when I've finished outputting the desired DLX file, it's time to say goodbye by summarizing what I did.

```
⟨ Bid farewell 36 ⟩ ≡
  if (¬sfxpresent)
    fprintf(stderr, "Altogether␣%d␣cells,␣%d␣pieces,␣%d␣base␣placements,␣%d␣nodes.\n",
        givenbox.size, piececount, basecount, curnode + 1);
  else fprintf(stderr, "Altogether␣%d+%d␣cells,␣%d␣pieces,␣%d␣base␣placements,␣%d␣nodes.\n",
        givenbox.size − sfxpresent, sfxpresent, piececount, basecount, curnode + 1);
```

This code is used in section 2.

37. Index.

⟨ Bid farewell 36 ⟩ Used in section 2.
⟨ Complete the range from t to x 7 ⟩ Used in section 5.
⟨ Create the other base placements equivalent to *newbox* 31 ⟩ Used in section 25.
⟨ Decode a bracketed specification 5 ⟩ Used in section 4.
⟨ Digest the optional suffix, *suf* 20 ⟩ Used in section 19.
⟨ Get ready for a range 6 ⟩ Used in section 5.
⟨ Global variables 8, 11, 22, 23 ⟩ Used in section 2.
⟨ Normalize the cells of *newbox* 29 ⟩ Used in section 25.
⟨ Output the DLX item-name line 34 ⟩ Used in section 2.
⟨ Output the DLX options 35 ⟩ Used in section 2.
⟨ Put the specified cells into *newbox*, starting at *buf*[p] 18 ⟩ Used in sections 17 and 25.
⟨ Read a piece spec 25 ⟩ Used in section 24.
⟨ Read the box spec 17 ⟩ Used in section 2.
⟨ Read the piece name, and find it in the *names* table at position k 27 ⟩ Used in section 25.
⟨ Read the piece specs 24 ⟩ Used in section 2.
⟨ Scan an xy spec 19 ⟩ Used in section 18.
⟨ Set up the *occupied* table 21 ⟩ Used in section 17.
⟨ Set *newbox* to *base*[j] transformed by 90° rotation 32 ⟩ Used in section 31.
⟨ Set *newbox* to *base*[j] transformed by xy transposition 33 ⟩ Used in section 31.
⟨ Store the name in *names*[*piececount*] and check its validity 28 ⟩ Used in section 27.
⟨ Subroutines 3, 4, 12, 13, 14, 15, 30 ⟩ Used in section 2.
⟨ Type definitions 9, 10 ⟩ Used in section 2.

POLYOMINO-DLX