## [3.26] Find indices of specific elements in lists and matrices

The built-in functions *min()* and *max()* return the minimum and maximum elements in lists and matrices, but there are no built-in functions that find the indices at which these values occur. This tip shows how to accomplish the more general task, of finding the locations of any target element, in TI Basic and in C.

*TI Basic version*

Considering lists first, the general problem is to find the indices at which a particular element is located, where that element is the minimum or maximum. This problem can be solved in a straight-forward way by comparing each element to the target value, as shown in this function:

```
indexlst(l,v)
func
©(l,v) return list of k|l[k]=v
©3nov01/dburkett@infinet.com

local k,r,j

{}→r                  ©Initialize result list ...
1→j                   ©... and result list index

for k,1,dim(l)        ©Loop to test each list element
 if l[k]=v then       ©If list element matches target ...
  k→r[j]              ©... save index in result list
  j+1→j               ©... and update result list index
 endif
endfor

return r              ©Return result list

Endfunc
```

Since more than one list element may match the target value, the matching indices are returned as a list. For example, if the list *mylist* is {5,4,7,3,2,1,1}, then *indexlst(mylist,5)* returns {1}, and *indexlst(mylist,1)* returns {6,7}.

*indexlst()* returns an empty list {} if the target element is not in the list. So, this function could also be used to determine if a list includes a particular element, by testing the returned result for *dim()* = 0.

This function can find the indices of the minimum and maximum list values, with the built-in *min()* and *max()* functions. To find the indices of the minimum list element, use

```
indexlst(mylist,min(mylist))
```

which, for the example above, returns {6,7}. To find the indices of the maximum list element, use

```
indexlst(mylist,max(mylist))
```

which returns {3}.

Finding the indices of a particular element in a matrix is only slightly more complicated, and we can use *indexlst()* to do the work, by converting the matrix to a list with `mat▸list()`. Since *indexlst()* returns a list, we must then convert the list indices to the equivalent matrix indices. This function implements these ideas.

```
indexmat(m,v)
```

```
Func
©(mat,v) return matrix of [k,l]|mat[k,l]=v
©calls util\indexlst()
©4nov01/dburkett@infinet.com

local r,n,j,i,s,c

mat▸list(m)→r              ©Convert the matrix to a list
util\indexlst(r,v)→r       ©Find list indices of v

if r={} then               ©Return empty string if no matching indices
 return ""
else                       ©Convert list indices to matrix indices
 dim(r)→c                  ©... get number of matching indicex
 coldim(m)→n               ©... get number of matrix columns
 newmat(c,2)→s             ©... make result matrix
 for i,1,c                 ©... loop to convert indices
  intdiv(r[i]-1,n)+1→j     ©... find row index
  j→s[i,1]                 ©... save row index
  r[i]-n*(j-1)→s[i,2]      ©... find and save column index
 endfor
endif

return s                   ©Return matrix of indices

EndFunc
```

The function returns an empty string ("") if the target value is not found. The calling program or function can test the result with *getType()* to determine if there are any matching indices.

If the matrix *m* is $\begin{bmatrix} -2 & -2 & 8 & -7 \\ -5 & 9 & -7 & -5 \\ -2 & 0 & -3 & 9 \\ -8 & 0 & -5 & 9 \end{bmatrix}$

then *indexmat(m,-2)* returns the result $\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 3 & 1 \end{bmatrix}$

indicating that the element -2 is at indices [1,1], [1,2] and [3,1]. In the index matrix returned by *indexmat()*, the first column is the row indices of the original matrix, and the second column is the column indices. You can use *rowDim()* on the result matrix to determine the number of matching elements.

As with the list index function, we can use *indexmat()* to find the indices of the minimum and maximum matrix elements, but we must first use `mat▸list()` in the argument of *min()* and *max()*, because those functions return a row vector of column extrema for matrix arguments. So, to find the indices of the minimum elements of matrix *m*, we use

```
indexmat(m,min(mat▸list(m)))
```

which returns [4,1], since there is only one element which is the minimum value of -8.

To find the maximum element indices, use

```
indexmat(m,max(mat▸list(m)))
```

which returns $\begin{bmatrix} 2 & 2 \\ 3 & 4 \\ 4 & 4 \end{bmatrix}$

and these are the indices of the maximum element 9.

*indexmat()* works on matrices of any row and column dimensions, and always returns a 2-column matrix. Note that you can use a single row-index value to extract the individual indices for the result matrix. For the example above, if we want the second index from the result, we use

    [[2,2][3,4][4,4]][2]

which returns [3,4].

*C version*

The TI Basic functions above are functional but slow. The same functionality can be programmed in C, but the trade-off is increased hassle in that ASM functions, including those written in C, cannot be used in expressions. Tip [12.2] shows how to get around this by installing a couple of hacks. Note that you must use the correct version of *inlist()* for your calculator model: .89z for the TI-89, and .9xz for the TI-92 Plus.

The C version, *inlist()*, works the same as *indexlst()* above for lists. However, for matrices, *inlist()* is used to search for a row of elements, and not the individual elements.

An early version of inlist(), which only found a single element occurence, was about 22 times faster than the equivalent TI Basic version.

The source code for *inlist()* follows:

```
/**********************************************************************
This function searches a specified list for a specified item. If the
target item is found, the function returns a list containing the position
of item. If the item is found more than once, the function returns
a list containing all the indices of the target item. If the item is not
found, the function returns an empty list. No restrictions exist on the
types of items in the list.

Error Code Descriptions
90  -- Argument must be a list
930 -- Too few arguments
940 -- Too many arguments

Syntax: inlist(list, item)
        inlist(matrix, row)

Since a matrix is defined as a list of lists, this function will also
work with input such as inlist([1,2;2,3], {1,2})

Thanks to Samuel Stearley for his help in optimizing this program.

Brett Sauerwein
15 March 02
*********************************************************************/

#define USE_TI89
```

```
#define USE_TI92PLUS
#define OPTIMIZE_ROM_CALLS

#define RETURN_VALUE

#include <tigcclib.h>

void _main(void)
{
    unsigned char *listPointer, *itemPointer;

    int argumentNumber = 0;
    int index = 1;

    // get a pointer to the first argument entered by the user

    InitArgPtr (listPointer);

    argumentNumber = remaining_element_count (listPointer);

    // throw the appropriate error if the user enters too many or
    // too few arguments, or doesn't enter a LIST or MATRIX

    if (argumentNumber > 2)
          ER_throwVar (940);

    if (argumentNumber < 2)
          ER_throwVar (930);

    if (*listPointer != LIST_TAG)
          ER_throwVar (90);

    // get a pointer to the second argument

    itemPointer = next_expression_index (listPointer);

    // decrement listPointer so it points to first element in list

    listPointer--;

    // designate the end of the list of indices that will be
    // pushed to the expression stack

    push_quantum (END_TAG);

    // step through the list, examining each element to see
    // if it matches target item; if it does, add its
    // position to the list

    while (*listPointer != END_TAG)
    {
          // compare_expressions returns 0 if the items match

          if (compare_expressions(listPointer, itemPointer) == 0)
                push_longint (index);

          // advance pointer to the next element

          listPointer = next_expression_index (listPointer);
          index++;
    }

    // list on the top of the stack is in descending order, so
    // reverse the list so items are in ascending order

    push_reversed_tail (top_estack);
    push_quantum (LIST_TAG);
    // estack clean up

    itemPointer = next_expression_index (top_estack);

    while (*itemPointer != END_TAG)
    {
          listPointer = itemPointer;
          itemPointer = next_expression_index (itemPointer);
          delete_expression (listPointer);
    }

} // main
```

*(Credit to Brett Sauerwein for the C version)*