Subject: Re: Array sorting by row
Posted by Jean H. on Thu, 17 Aug 2006 16:06:43 GMT
View Forum Message <> Reply to Message

```
a = [[4,2,0,5],[9,0,1,5],[0,4,2,1],[1,2,3,4]]
b = sort(a) ;sort the whole array

sizeX = 4 ;(use DIM instead)
sizeY = 4
nbElements = n_elements(a)

c = b/sizeX  ;tells you, for each element of b, which line of A the
    ;index correspond to

useless = histogram(c, reverse_indices = ri)
;get the reverse indices of the histogram. It tells you, for each line
of a (print useless, you will get 4 4 4 4), the indices of b that
correspond to it.

;get the sorted indices, resorted by lines. You have, for sure, 4 bin
here, so the first indice of b is located at ri[5].
d = b[ri[sizeX+1:sizeX+nbElements]]

minus = indgen(sizeY) * sizeX
minusBIG = transpose(rebin(minus,sizeX,sizeY))
;you want the indices on each line and not on the whole array (if that's
what you want, d is fine for you then)

result =  reform(d-minusBIG, sizeX,sizeY)


IDL> print, result
       2       1       0       3
       1       2       3       0
       0       3       2       1
       0       1       2       3
```

I suggest you to learn to use the histogram... it's wonderful what you can do with it!

Jean

humphreymurray@gmail.com wrote:
> Hi,
>
> In IDL, is there a way to independly sort the columns of a 2d matrix

> without looping through and sorting each row individually?
>
> Currenly I'm using:
>
> for x = long(0), x_size - 1 do begin
> sorted_indexs[0,x] = sort(matrix[*,x])
> endfor
>
> For example, if the matrix contained the following values:
>
> 4 2 0 5
> 9 0 1 5
> 0 4 2 1
> 1 2 3 4
>
> I want the result matrix to contain index's like:
>
> 2 1 0 3
> 1 2 3 0
> 0 3 2 1
> 0 1 2 3
>
> Here, each column is sorted as if it's an independant vector.
>
> Cheers.
>

Subject: Re: Array sorting by row
Posted by humphreymurray on Fri, 18 Aug 2006 02:52:10 GMT
View Forum Message <> Reply to Message

Cheers Jean, that code of yours runs well.

However, I decided to switch back to my original code with the for loop. This was because the time taken to sort increases significantly with the number of elements to be sorted. In your approach, you are sorting the entire array, and then working out what elements are in each row.

My original code (with the loops) runs faster (for me anyway), because each sort operation is sorting a smaller subset of the numners, and so is quicker.

That is, unless somebody can proof me wrong :-p

Humphrey

Jean H. wrote:
> a = [[4,2,0,5],[9,0,1,5],[0,4,2,1],[1,2,3,4]]
> b = sort(a) ;sort the whole array
>
> sizeX = 4 ;(use DIM instead)
> sizeY = 4
> nbElements = n_elements(a)
>
> c = b/sizeX ;tells you, for each element of b, which line of A the
> ;index correspond to
>
> useless = histogram(c, reverse_indices = ri)
> ;get the reverse indices of the histogram. It tells you, for each line
> of a (print useless, you will get 4 4 4 4), the indices of b that
> correspond to it.
>
> ;get the sorted indices, resorted by lines. You have, for sure, 4 bin
> here, so the first indice of b is located at ri[5].
> d = b[ri[sizeX+1:sizeX+nbElements]]
>
> minus = indgen(sizeY) * sizeX
> minusBIG = transpose(rebin(minus,sizeX,sizeY))
> ;you want the indices on each line and not on the whole array (if that's
> what you want, d is fine for you then)
>
> result = reform(d-minusBIG, sizeX,sizeY)
>
>
> IDL> print, result
> 2 1 0 3
> 1 2 3 0
> 0 3 2 1
> 0 1 2 3
>
>
> I suggest you to learn to use the histogram... it's wonderful what you
> can do with it!
>
> Jean
>
>
> humphreymurray@gmail.com wrote:
>> Hi,
>>
>> In IDL, is there a way to independly sort the columns of a 2d matrix
>> without looping through and sorting each row individually?
>>

>> Currenly I'm using:
>>
>> for x = long(0), x_size - 1 do begin
>> sorted_indexs[0,x] = sort(matrix[*,x])
>> endfor
>>
>> For example, if the matrix contained the following values:
>>
>> 4 2 0 5
>> 9 0 1 5
>> 0 4 2 1
>> 1 2 3 4
>>
>> I want the result matrix to contain index's like:
>>
>> 2 1 0 3
>> 1 2 3 0
>> 0 3 2 1
>> 0 1 2 3
>>
>> Here, each column is sorted as if it's an independant vector.
>>
>> Cheers.
>>

Subject: Re: Array sorting by row
Posted by MarioIncandenza on Fri, 18 Aug 2006 18:12:36 GMT
View Forum Message <> Reply to Message

Try this one out for size (or rather, for speed):
a = [[4,2,0,5],[9,0,1,5],[0,4,2,1],[1,2,3,4]]; or any 2-d array
sz=size(a,/dimensions); get dimensions
cols=rebin(indgen(sz[0]),sz[0],sz[1]); label points by column
 h2=hist_nd(transpose([[a[*]],[cols[*]]]),[1,1],reverse_indic es=ri2)
; syntax is a wee messy, but this is JD Smith's beautiful HIST_ND()
function:
; http://www.dfanning.com/documents/programs.html#HIST_ND
starti=n_elements(h2)+1; beginning of indices in $RI2
sorted=transpose(reform(a[ri2[starti:*]],sz[0],sz[1]))

This one should be blazing fast right up to memory limits.

Cheers,

Edward H.

Subject: Re: Array sorting by row
Posted by JD Smith on Tue, 22 Aug 2006 20:48:59 GMT
View Forum Message <> Reply to Message

On Fri, 18 Aug 2006 11:12:36 -0700, Ed Hyer wrote:

> Try this one out for size (or rather, for speed):
> a = [[4,2,0,5],[9,0,1,5],[0,4,2,1],[1,2,3,4]]; or any 2-d array
> sz=size(a,/dimensions); get dimensions
> cols=rebin(indgen(sz[0]),sz[0],sz[1]); label points by column
> h2=hist_nd(transpose([[a[*]],[cols[*]]]),[1,1],reverse_indic es=ri2)
> ; syntax is a wee messy, but this is JD Smith's beautiful HIST_ND()
> function:
> ; http://www.dfanning.com/documents/programs.html#HIST_ND
> starti=n_elements(h2)+1; beginning of indices in $RI2
> sorted=transpose(reform(a[ri2[starti:*]],sz[0],sz[1]))
>
> This one should be blazing fast right up to memory limits.

Interesting method. However, a (usual sort of) problem occurs if your input array is sparse, e.g.:

a=[[4,2,0,5],[9,0,1,5],[0,4,2,1],[1,212121212L,3,4]]

In this case, you will quickly fill up memory with countless zeroes in a nearly empty 2D histogram.

A related method which avoids this sparseness issue, and allows arbitrary floating point numbers, is as follows:

```
sz=size(a,/DIMENSIONS)
s=sort(a)
h=histogram(s mod sz[0],REVERSE_INDICES=ri)
sorted_inds=transpose(reform(s[ri[sz[0]+1L:*]],sz))
sorted=a[sorted_inds]
```

Basically, you sort the entire array, and then bin the sorted list by column number using HISTOGRAM. The histogram is guaranteed to contain only as many bins as you have columns in your input array, which is a small number even for very large arrays. This means the algorithm offers a constant runtime depending only on array size.

There's one other variant possible. If you know in advance your values are not sparsely distributed (e.g. at least 1 in 10 number are represented, on average), you can see further speedup (depending on your memory resources) by changing:

```
s=sort(a)
```

in the above prescription to

```
h=histogram(a,REVERSE_INDICES=ri)
s=ri[n_elements(h)+1:*]
```

i.e. just sort directly using HISTOGRAM. This really has no business being this much faster than SORT, but it is. Here are some timings comparing the three methods:

```
; 100x100 array, on average every number represented (1 in 1 sparse)
HIST_ND method:                 0.17189288
SORT+HISTOGRAM method:          0.0094909668
HISTOGRAM+HISTOGRAM method:     0.0018260479

; 500x500 array, 1 in 1 sparse
HIST_ND method:                 10.859160
SORT+HISTOGRAM method:          0.37013221
HISTOGRAM+HISTOGRAM method:     0.17653203

; 500x500 array 10 in 1 sparse (numbers repeated 10 times on average)
HIST_ND method:                 1.3364871
SORT+HISTOGRAM method:          0.37795591
HISTOGRAM+HISTOGRAM method:     0.10262513

; 500x500 array, 1 in 10 sparse (histogram only 10% filled)
HIST_ND method:           <array too large to allocate>
SORT+HISTOGRAM method:          0.35415411
HISTOGRAM+HISTOGRAM method:     0.35962415
```

And, just to show that in some small part of parameter space the first two are similar.

```
; 2000x2000 array, 100 in 1 sparse (numbers repeated 100 times on average)
HIST_ND method:                 11.669180
SORT+HISTOGRAM method:          10.190833
HISTOGRAM+HISTOGRAM method:     2.4765849
```

And to really puts a hurt on the pure HISTOGRAM based methods, by cranking up the sparseness (HIST_ND is already out of its league):

```
; 1000x1000 array, 1 in 5 sparse
SORT+HISTOGRAM method:          2.0481181
HISTOGRAM+HISTOGRAM method:     1.2417412

; 1000x1000 array, 1 in 10 sparse
SORT+HISTOGRAM method:          2.0289030
HISTOGRAM+HISTOGRAM method:     1.6590791
```

; 1000x1000 array, 1 in 50 sparse
SORT+HISTOGRAM method: 2.0890758
HISTOGRAM+HISTOGRAM method: 4.4074631

; 2000x2000 array, 1 in 50 sparse
SORT+HISTOGRAM method: 10.333820
HISTOGRAM+HISTOGRAM method: <array too large to allocate>

So there you have it. These results are highly reminiscent of an old post giving the various trade-offs using SORT and HISTOGRAM for list matching and set operations, see http://www.dfanning.com/tips/set_operations.html. The advice offered is so similar, I'll quote myself:

"So, if you want a generic solution which works in the same n log(n) time using a fixed amount of memory for any type of integer input data, sparse or not, use SORT. If you know your data is non-sparse (better than 1 in 10 say), you can see a speedup of a few with HISTOGRAM."

JD
