DYALOG

Belfast 2018

SA2: Performance Tuning

Jay Foad, Roger Hui,
Marshall Lochbaum

Introduction

- Measuring performance
- Data types
- Contiguous data

Measuring performance

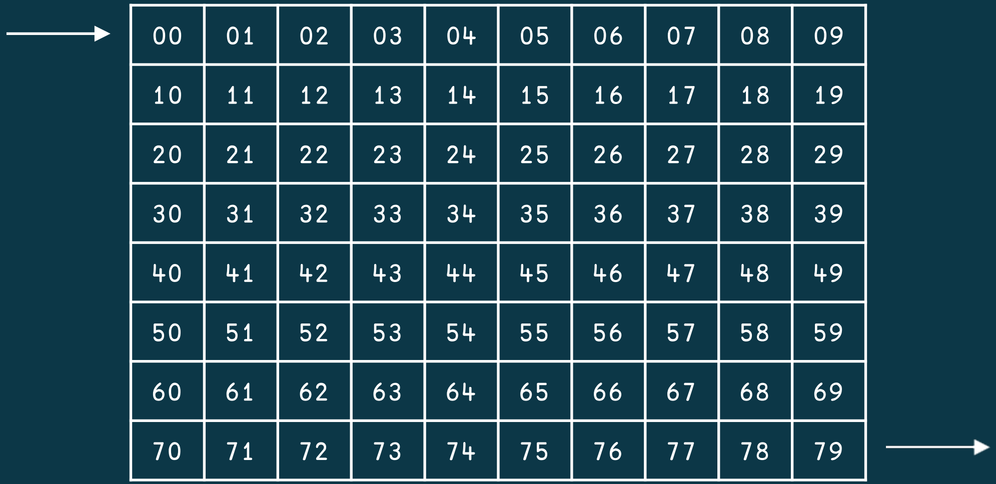
- `cmpx`
- `⎕PROFILE` and `]PROFILE`
- `APLMON`

Data types

- Dyalog has many data types
- Smaller ones are (almost) always better!
- ⎕DR is a useful diagnostic tool…
- … but can be misleading

Contiguous data

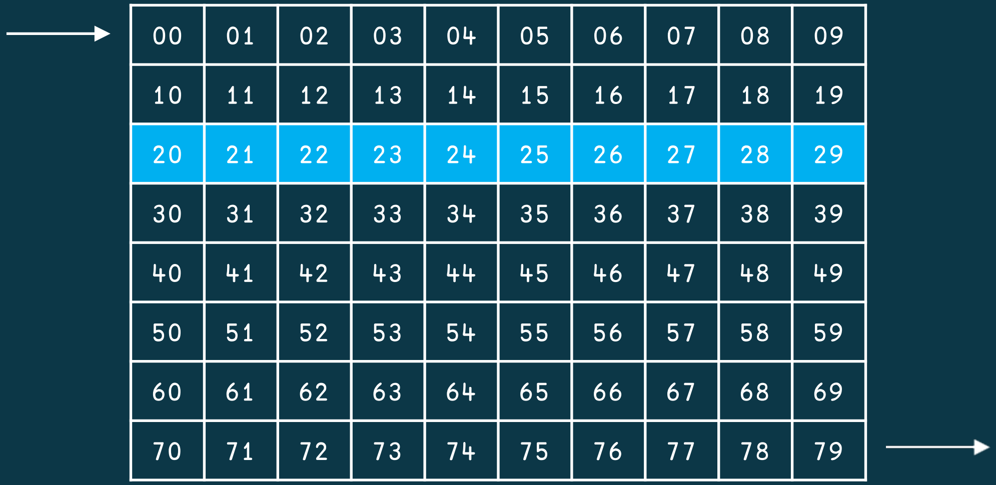
- Array data is stored in memory in *ravel order*



00	01	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

Contiguous data

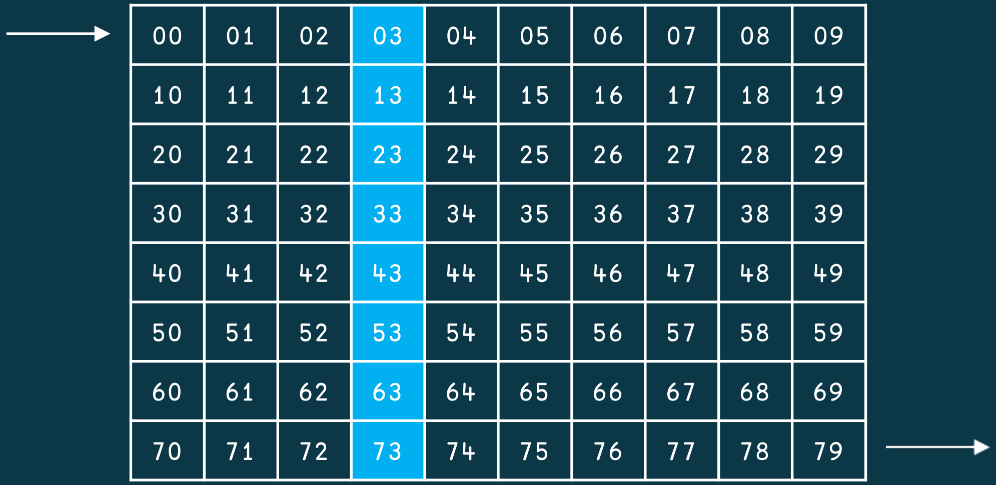
- The data for a single row is *contiguous* in memory



00	01	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

Contiguous data

- The data for a single column is *not* contiguous



00	01	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

Contiguous data

- Efficient operations:
  - move whole rows around, or
  - move data around within a single row

- Inefficent operations:
  - move whole columns around, or
  - move data around within a single column

Contiguous data

- Historical note:
  primitives inspired by Sharp APL / J
  strongly encourage you to work on leading axes

- (Some APL2 primitives do too,
  notably grade up and grade down)

Contiguous data

- What if you really need to work on the last axis?
  1. Transpose your data, then transpose back
     (one day we will implement Under)
  2. Try not to have this problem in the first place
  3. Think about inverted tables

Selective assignment

- Back in version 14.1...

```
(Idxs⌷Cube) +← Fn Idxs⌷Cube ⍝ slow

(P Q R)←Idxs
Cube[P;Q;R] +← Fn Cube[P;Q;R] ⍝ fast
```

- (This was fixed in version 15.0)

Selective assignment

```
      A←'hello'

      (2↑⌽A)←'xi'

      A
helix
```

Selective assignment

```
      A←'hello'

      (2↑⌽A)←'xi'

      A                1 2 3 4 5
helix
```

Selective assignment

```
      A←'hello'

      (2↑⌽A)←'xi'

      A                   5 4 3 2 1
helix
```

Selective assignment

```
      A←'hello'

      (2↑⌽A)←'xi'

      A                  5 4
helix
```

Selective assignment

```
      A←'hello'

      (2↑⌽A)←'xi'

      A                A[5 4]←'xi'
helix
```

Selective assignment

```
      A←'hello'
      (2↑⌽A)←'xi'
      A
helix
```

- Do it yourself!

```
      A←'hello'
      A[2↑⌽⍳⍴A]←'xi'
      A
helix
```

Selective assignment

```
      A←'hello'
      (2↑⌽A)←'xi'
      A
helix
```

- Do it yourself!

```
      A←'hello'
      A[2↑⌽⍳⍴A]←'xi'
      A
helix
```

Selective assignment

- For *vectors* the DIY code is pretty fast
- The DIY code also works for higher ranked arrays

Selective assignment

- For *vectors* the DIY code is pretty fast

- The DIY code also works for higher ranked arrays
- but the index array `⍳⍴A` is nested

Selective assignment

- For *vectors* the DIY code is pretty fast

- The DIY code also works for higher ranked arrays
- but the index array ⍳⍴A is nested
- so performance sucks

Selective assignment

- Always creates the whole index array

```
A←⍳1E9
 (2↑A)←¯1 ⍝ slow
A[1 2]←¯1 ⍝ fast
```

Selective assignment

- Always creates the whole index array
- …except when it doesn't

```
      A←⍳1E9
((⊂1 2)⌷A)←¯1 ⍝ fast(ish)
      A[1 2]←¯1 ⍝ fast
```