

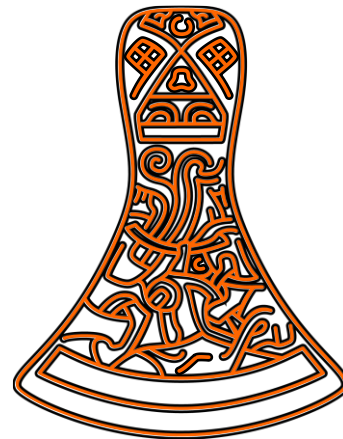
# DYALOC

Glasgow 2024

## Performance Basics



*Josh David*

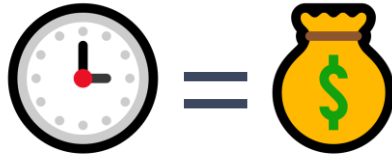


# Introductions

- Clone this repo to follow along with some examples

<https://github.com/dyalog-training/2024-SA2>

# Why does it matter?



# Performance

- ◆ A balance between memory and time of program execution

# Programming Languages

Compiled



Interpreted



# Interpreted, dynamically typed

- Overhead converting code at runtime
- + Immediate feedback
- + Interactive development
- + Specialised algorithms depending on data

[Dyalog '18: The Interpretive Advantage](#)

# The Interpretive Advantage

- ◆ APL can dynamically decide which algorithm to use based on the shape and type of your data
  - ◆ Important for benchmarking: test your algorithms against varying types and sizes

# Big O Notation

- How Computer Scientists discuss algorithm efficiency
- “Big O notation (with a capital letter O, not a zero), also called Landau's symbol, is a symbolism used in complexity theory, computer science, and mathematics to describe the asymptotic behavior of function.”

[https://web.mit.edu/16.070/www/lecture/big\\_o.pdf](https://web.mit.edu/16.070/www/lecture/big_o.pdf)



# Big O Notation

- How Computer Scientists discuss algorithm efficiency
- ~~“Big O notation (with a capital letter O, not a zero), also called Landau's symbol, is a symbolism used in complexity theory, computer science, and mathematics to describe the asymptotic behavior of function. Basically, it tells you how fast a function grows or declines”~~

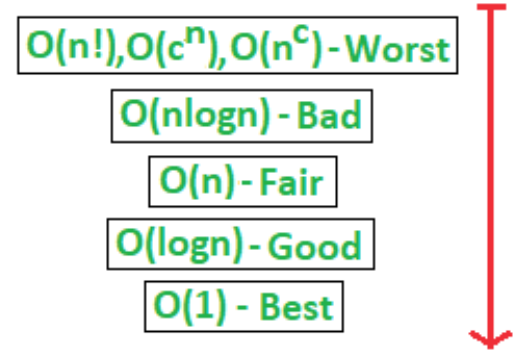
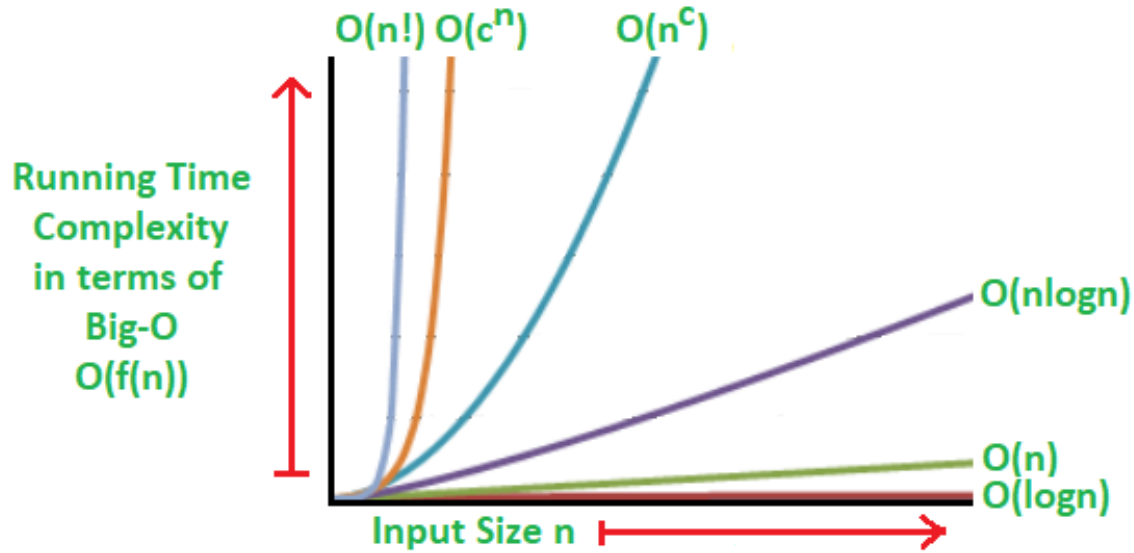
[https://web.mit.edu/16.070/www/lecture/big\\_o.pdf](https://web.mit.edu/16.070/www/lecture/big_o.pdf)

# Big O Notation

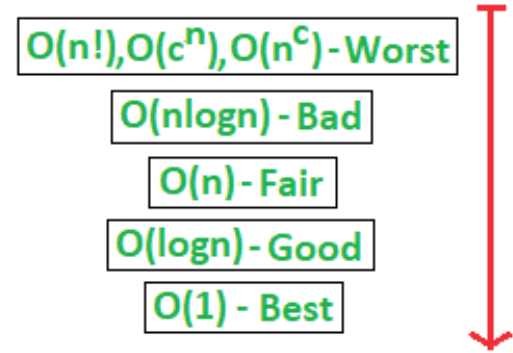
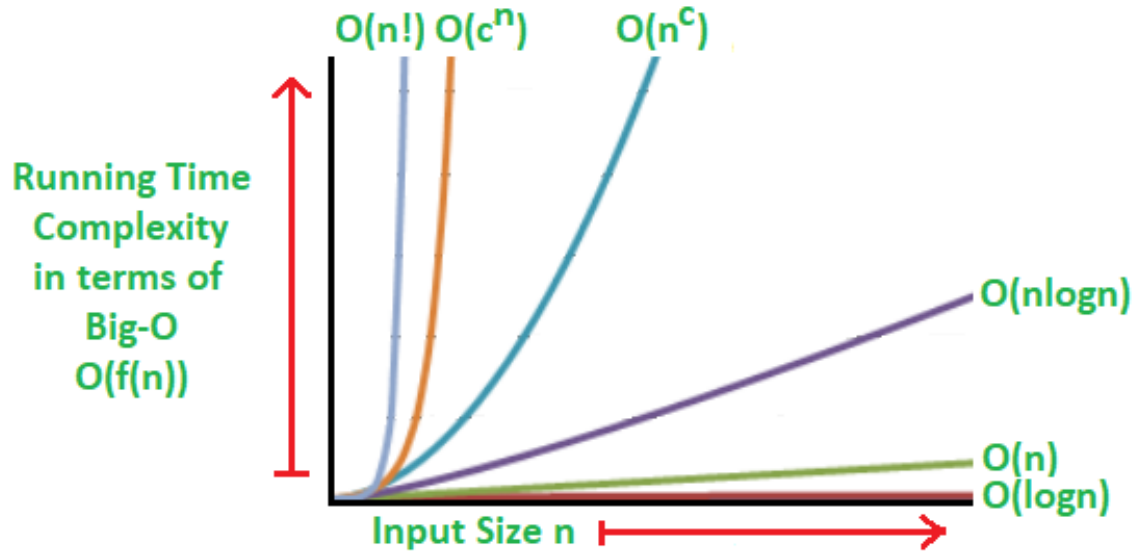
- How Computer Scientists discuss algorithm efficiency
- How does this function scale with input size?

# Big O Notation

- ◆ Constants are ignored
  - ◆  $10n \rightarrow n$
- ◆ “Worst” or largest factor dominates
  - ◆ An algorithm with  $O(n^2)$  and  $O(n)$  is described as  $O(n^2)$



<https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/>



\* Useful to describe Space complexity as well!

# Big O Notation

- ◆ Useful for thinking about space complexity as well (memory)
- ◆ Different results for
  - ◆ Best case
  - ◆ Worst case
  - ◆ Average case

# Array Sorting Algorithms

| Algorithm             | Time Complexity     |                        |                   | Space Complexity |
|-----------------------|---------------------|------------------------|-------------------|------------------|
|                       | Best                | Average                | Worst             | Worst            |
| <u>Quicksort</u>      | $\Omega(n \log(n))$ | $\Theta(n \log(n))$    | $O(n^2)$          | $O(\log(n))$     |
| <u>Mergesort</u>      | $\Omega(n \log(n))$ | $\Theta(n \log(n))$    | $O(n \log(n))$    | $O(n)$           |
| <u>Timsort</u>        | $\Omega(n)$         | $\Theta(n \log(n))$    | $O(n \log(n))$    | $O(n)$           |
| <u>Heapsort</u>       | $\Omega(n \log(n))$ | $\Theta(n \log(n))$    | $O(n \log(n))$    | $O(1)$           |
| <u>Bubble Sort</u>    | $\Omega(n)$         | $\Theta(n^2)$          | $O(n^2)$          | $O(1)$           |
| <u>Insertion Sort</u> | $\Omega(n)$         | $\Theta(n^2)$          | $O(n^2)$          | $O(1)$           |
| <u>Selection Sort</u> | $\Omega(n^2)$       | $\Theta(n^2)$          | $O(n^2)$          | $O(1)$           |
| <u>Tree Sort</u>      | $\Omega(n \log(n))$ | $\Theta(n \log(n))$    | $O(n^2)$          | $O(n)$           |
| <u>Shell Sort</u>     | $\Omega(n \log(n))$ | $\Theta(n(\log(n))^2)$ | $O(n(\log(n))^2)$ | $O(1)$           |
| <u>Bucket Sort</u>    | $\Omega(n+k)$       | $\Theta(n+k)$          | $O(n^2)$          | $O(n)$           |
| <u>Radix Sort</u>     | $\Omega(nk)$        | $\Theta(nk)$           | $O(nk)$           | $O(n+k)$         |
| <u>Counting Sort</u>  | $\Omega(n+k)$       | $\Theta(n+k)$          | $O(n+k)$          | $O(k)$           |
| <u>Cubesort</u>       | $\Omega(n)$         | $\Theta(n \log(n))$    | $O(n \log(n))$    | $O(n)$           |

<https://www.bigocheatsheet.com/>

# Big O Notation for APL

- Can be tricky due to different underlying algorithms used in primitives
- Helpful to reason about it by investigating individual primitives (caveat: idioms)
- Actually profiling your algorithms against your known datatype/size is the best way to see how it scales



{ [ / + \ ( + - ~ ) ' ( ' = ( ω ∈ ' ( ) ' ) / ω }

$\alpha \in \omega$   $O(n \times m)$

$\alpha = \omega$   $O(n)$

$\alpha / \omega$   $O(n)$

$\sim \omega$   $O(n)$

$\alpha - \omega$   $O(n)$

$+ \backslash \omega$   $O(n^2)$

$[ / \omega$   $O(n)$

{ [ / + \ ( + - ~ ) ' ( ' = ( ω ∈ ' ( ) ' ) / ω }

$\alpha \in \omega$   $O(n)$

$\alpha = \omega$   $O(n)$

$\alpha / \omega$   $O(n)$

$\sim \omega$   $O(n)$

$\alpha - \omega$   $O(n)$

$+ \backslash \omega$   $O(n^2)$

$[ / \omega$   $O(n)$

{ [ / + \ ( + - ~ ) ' ( ' = ( ω ∈ ' ( ) ' ) / ω }

$\alpha \in \omega$   $O(n)$

$\alpha = \omega$   $O(n)$

$\alpha / \omega$   $O(n)$

$\sim \omega$   $O(n)$

$\alpha - \omega$   $O(n)$

$+ \backslash \omega$   $O(n^2)$

$[ / \omega$   $O(n)$

{ [ / + \ ( + - ~ ) ' ( ' = ( ω ∈ ' ( ) ' ) / ω }

$\alpha \in \omega$   $O(n)$

$\alpha = \omega$   $O(n)$

$\alpha / \omega$   $O(n)$

$\sim \omega$   $O(n)$

$\alpha - \omega$   $O(n)$

$+ \backslash \omega$   $O(n)$

$[ / \omega$   $O(n)$

# Exercise: PDepth

- Write your own Pdepth function, and let's compare times

# Timing

Aim for 2x faster

```
]runtime -c "expr1" "expr2"
```

-50%      2x faster

+100%     2x slower

# Timing

Try this now

```
]runtime -c "[DL 0.3" "[DL 0.6"
```

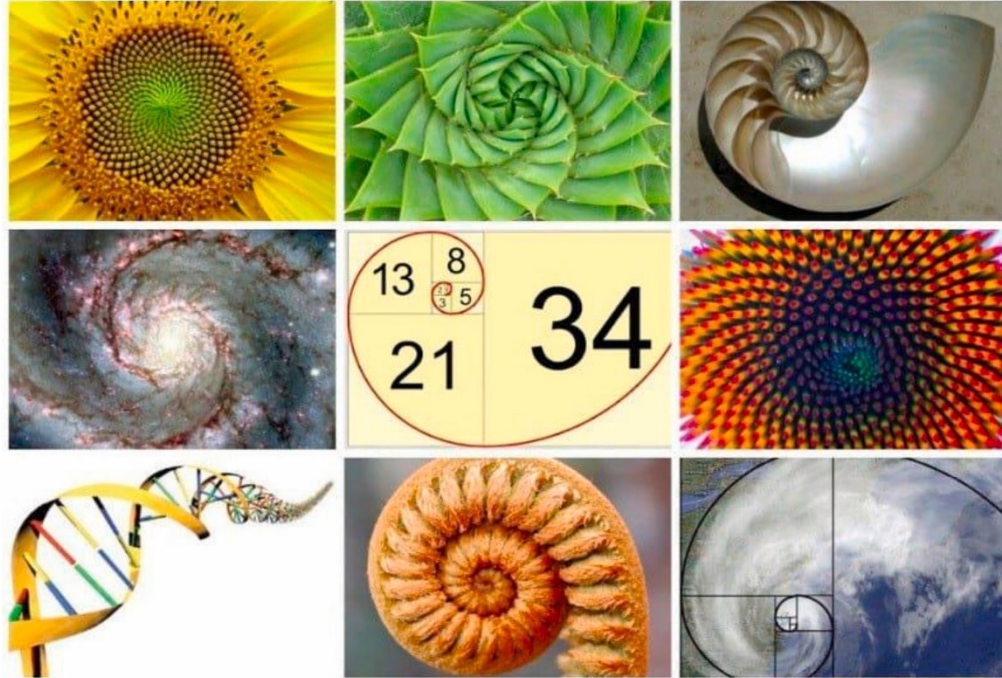
```
[dl 0.3 → 3.1E-1 | 0% ████████████████████████████████████
```

```
* [dl 0.6 → 6.1E-1 | +96% ████████████████████████████████████████████████████████████████████████████████
```

# Exercise: Fibonacci

$$F_0 = 0, \quad F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$



<https://www.sciencedirect.com/science/article/pii/S240584402303387X>



```
fibRec←{
    α←0 1
    ω=0:θρα
    (1↓α,+/α)∇ ω-1
}
```

The following function illustrates the relationship between the Fibonacci sequence and rational approximations to the "golden mean" (Phi).

```
fib←{1^+o÷/0,ω/1}
      |    |    |
      |    |    └── continued fraction: 0 1 1 1 ...
      |    └── approximation to Phi-1: 0 1 0.5 0.666 ...
      └── numerator of rational: 0 1 1 2 3 5 8 13 21 34 ...
```

```
fib ← (+.!oφ~ι)        A Sum of binomial coefficients (Jay Foad)
```

```
]runtime -c "(+.!oφ~ι)10" "{1^+o÷/0,ω/1}10" "fibRec 10"
(+.!oφ~ι)10      → 6.7E-6 | 0% ████████████████████████████████████████████████████████████████████████████
{1^+o÷/0,ω/1}10 → 1.8E-6 | -73% ████████████
fibRec 10       → 5.8E-6 | -14% ████████████████████████████████████████████████████████████████████████████
```

# The Array Model

In simple terms, simple arrays in memory:

[shape...], [elements in ravel order...]

# The Array Model

In simple terms, simple arrays in memory:

2 3 4 ABCDEFGHIJKLMNOPQRSTUVWXYZ

# The Array Model

In simple terms, simple arrays in memory:

2 3 4 ABCDEFGHIJKLMNOPQRSTUVWXYZ

ABCD

EFGH

IJKL

MNOP

QRST

UVWX

# The Array Model

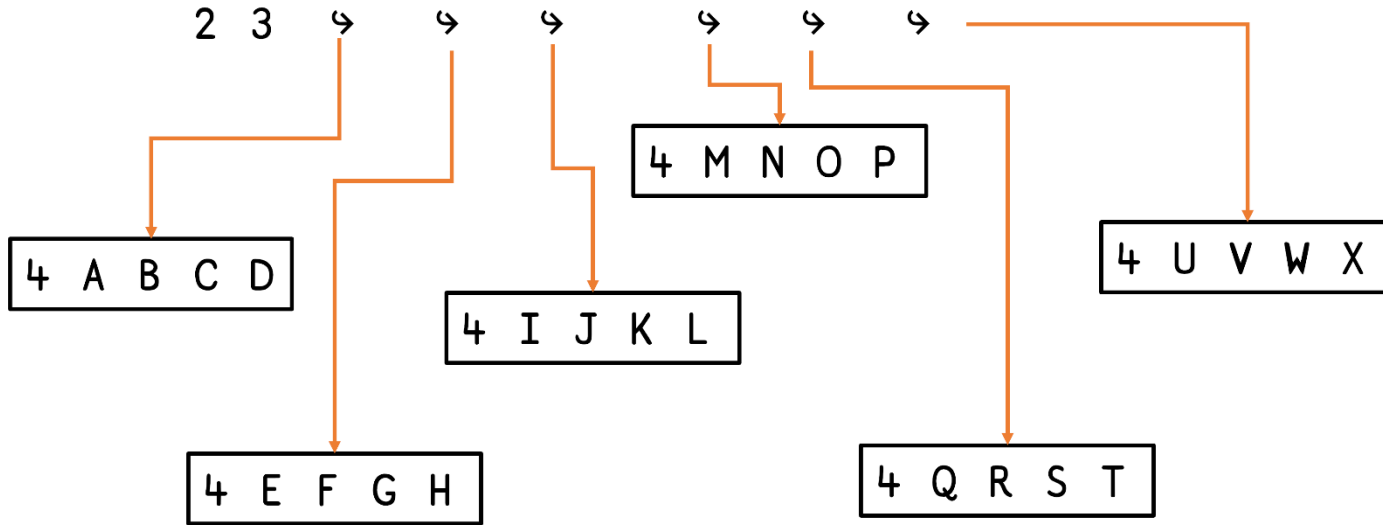
In simple terms, **nested arrays** in memory:

|      |      |      |
|------|------|------|
| ABCD | EFGH | IJKL |
| MNOP | QRST | UVWX |

# The Array Model

|      |      |      |
|------|------|------|
| ABCD | EFGH | IJKL |
| MNOP | QRST | UVWX |

In simple terms, **nested arrays** in memory:



# Techniques for Array Performance

- ◆ Use flat arrays where possible
- ◆ Vectors, unless utilising shape as part of computation
- ◆ Fewer, larger nests are better than many small nests
- ◆ Work on rows rather than columns (last axis principle)

# Move Loops Inside

Sometimes easier to reason over individual units

(scalar-scalar, row-row, scalar-list, matrix-matrix etc.)

then loop over whole data set.

In APL, it is faster to apply fewer primitives to larger sets of data.



# Exercise

Write equivalent functions using flat array techniques.

1.  $\{\epsilon \omega \uparrow \cdot \cdot 1\}$

2.  $\{+ / \cdot \cdot \alpha \subset \omega\}$

3.  $\{\supset (\neg, ' ', \vdash) / \alpha \uparrow ' ' (\neq \subseteq \vdash) \omega\}$

4.  $\{\supset (\neg, ' ', \vdash) / \phi \cdot \cdot ' ' (\neq \subseteq \vdash) \omega\}$

# Exercise

~~Write equivalent functions using flat array techniques.~~

Use APLCart to find flat array equivalents.

1.  $\{\epsilon \omega \uparrow \cdot \cdot 1\}$

2.  $\{+ / \cdot \cdot \alpha \subset \omega\}$

3.  $\{\supset (\neg, ' ', \vdash) / \alpha \uparrow ' ' (\neq \subseteq \vdash) \omega\}$

4.  $\{\supset (\neg, ' ', \vdash) / \phi \cdot \cdot ' ' (\neq \subseteq \vdash) \omega\}$

# Mix nested vectors (sometimes)

7↑w



≠w

10000

```
]runtime -c "w[⚡w]" "w[⚡↑w]"
```

```
w[⚡w]    → 3.5E-3 | 0% ████████████████████████████████████████████████████████████████████████████████████
w[⚡↑w]   → 1.5E-3 | -56% ██████████████████████████████████████████████████████████████████████████████████
```

# Mix nested vectors (sometimes)

```
        wl 'here' 'words' 'are' 'easy'  
20 4 6 8
```

```
        wlö† 'here' 'words' 'are' 'easy'  
20 4 6 8
```

```
]runtime -c "wl 'here' 'words' 'are' 'easy'" "wlö† 'here' 'words' 'are' 'easy'"
```

```
wl 'here' 'words' 'are' 'easy' → 3.0E-4 | 0%   
wlö† 'here' 'words' 'are' 'easy' → 1.3E-4 | -57%   
          
```

# Tip for searching

- Set  $\square CT \leftarrow 0$  before doing lookups on floats

# Check the size of your data

| Value   | Data Type                                 |
|---------|---|
| 111     | Boolean                                   |
| 808     | bits character                            |
| 838     | bits signed integer                       |
| 16016   | bits character                            |
| 16316   | bits signed integer                       |
| 32032   | bits character                            |
| 32332   | bits signed integer                       |
| 326     | Pointer (32-bit or 64-bit as appropriate) |
| 64564   | bits Floating                             |
| 1287128 | bits Decimal                              |
| 1289128 | bits Complex                              |

DR

\*On classic, characters are different

82 for 8 bit char

# Exercise

- Come up with an expression to determine if a vector of items is a Character Array or not
- First, create a random vector of random characters and numbers

# Exercise: A billion bools

- Create a variable that has a billion numbers between 0-1
  - Hint: test your code on a smaller size first!
- What's the size of your variable, and how much memory does each element take?



# Squeezing

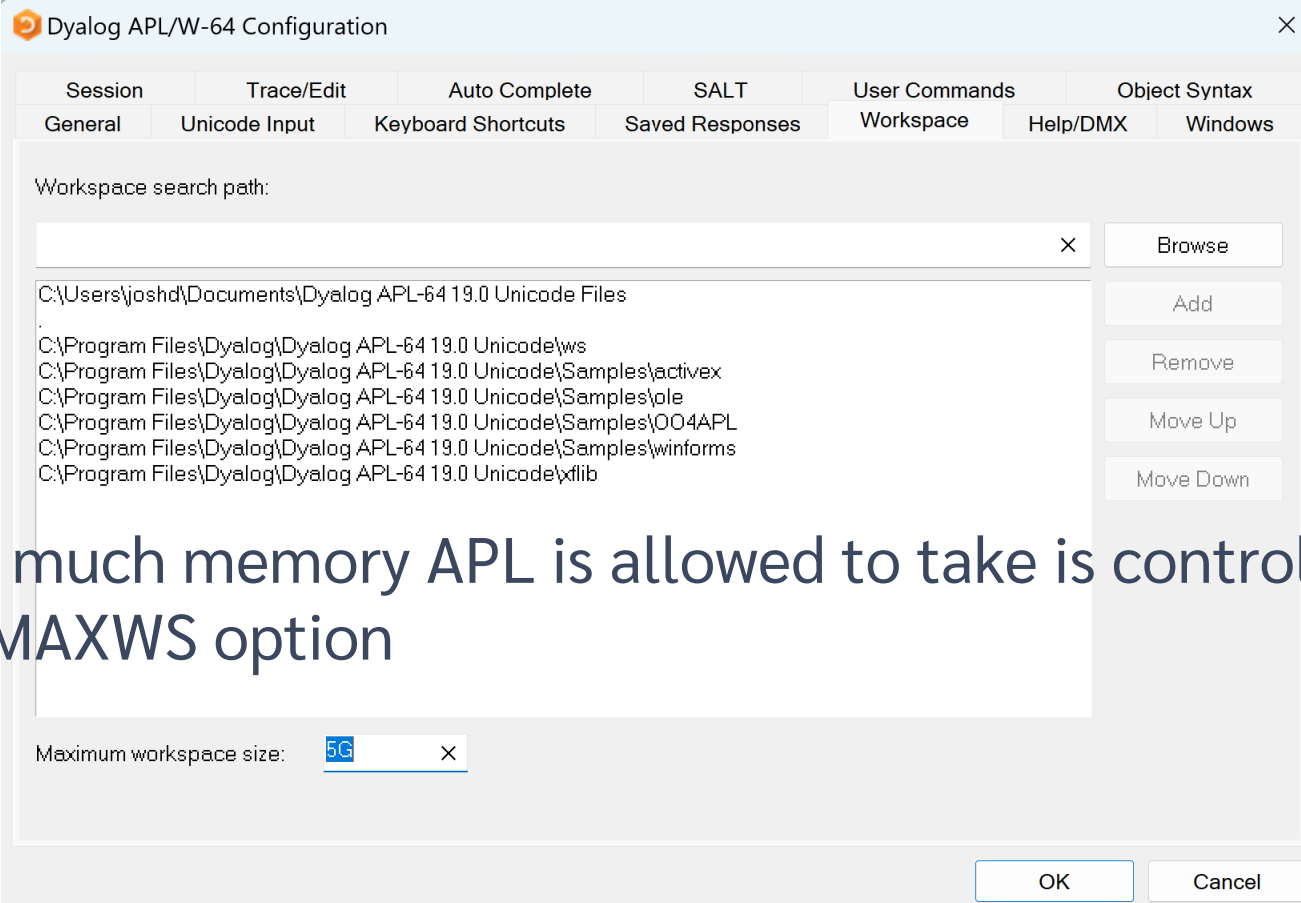
- □SIZE, □DR, □WA will squeeze your data
- Use (181 □) to check the “unsqueezed” type

# Exercise: Modify a boolean list

- Applying a Boolean mask to a table is a common operation. Sometimes we want to leave the first bit (header row) included. Write an APL function to convert the first bit to a 1



# MAXWS



The screenshot shows the "Dyalog APL/W-64 Configuration" dialog box. The "Workspace" tab is selected. The "Workspace search path:" section contains a list of paths: C:\Users\joshd\Documents\Dyalog APL-64 19.0 Unicode Files, C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\ws, C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\activex, C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\ole, C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\OO4APL, C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\winforms, and C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\xflib. The "Maximum workspace size:" is set to 5G. The dialog has tabs for Session, Trace/Edit, Auto Complete, SALT, User Commands, and Object Syntax. The "Workspace" tab is active, showing sub-tabs for General, Unicode Input, Keyboard Shortcuts, Saved Responses, Workspace, Help/DMX, and Windows.

Workspace search path:

C:\Users\joshd\Documents\Dyalog APL-64 19.0 Unicode Files  
.  
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\ws  
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\activex  
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\ole  
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\OO4APL  
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\winforms  
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\xflib

Maximum workspace size: 5G

- How much memory APL is allowed to take is controlled by the MAXWS option

# Garbage Collection

- □ WA will tell you how much Workspace is available
- An expensive operation, use judiciously
  - “Quick WA” i-beam better if needed at runtime

# WS FULL

- ◆ Dyalog reserves a special WS Full Buffer for handling WS FULL errors.
- ◆ The default size of this buffer is  $(1\text{MB}) \lfloor (0.01 \times \square \text{WA})$

# Memory vs Time consumption

- $\nabla$  hwm  $\leftarrow$  mem expr  
{}  $\square$  WA  
{0(2000  $\perp$ )14  
hwm  $\leftarrow$  2000  $\perp$  14  
{  $\perp$  expr  
hwm- $\sim$   $\leftarrow$  2000  $\perp$  14  
 $\nabla$

)copy dfns wsreq

# Exercise: Lasagna Problem

- DefineVars
- What's the size of each variable?
  - $\{\omega, \tau, \text{SIZE } \omega\} \square \text{THIS} . \square \text{NL} - 2$
- What is the Data Representation of n. How can that help you deduce the size?



# Profiling an Application

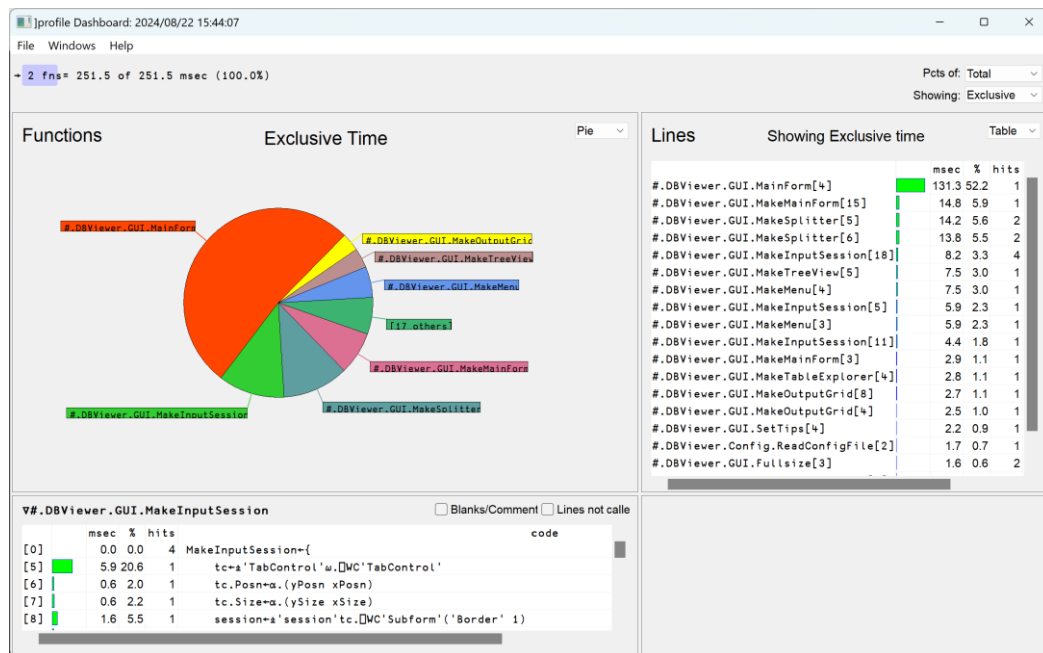
profile 'clear'

profile 'start'

...

profile 'stop'

]profile



# Demo/Exercise

# Idiom Recognition

- ◆ Allows the interpreter to apply a combination of primitives in a more performant way

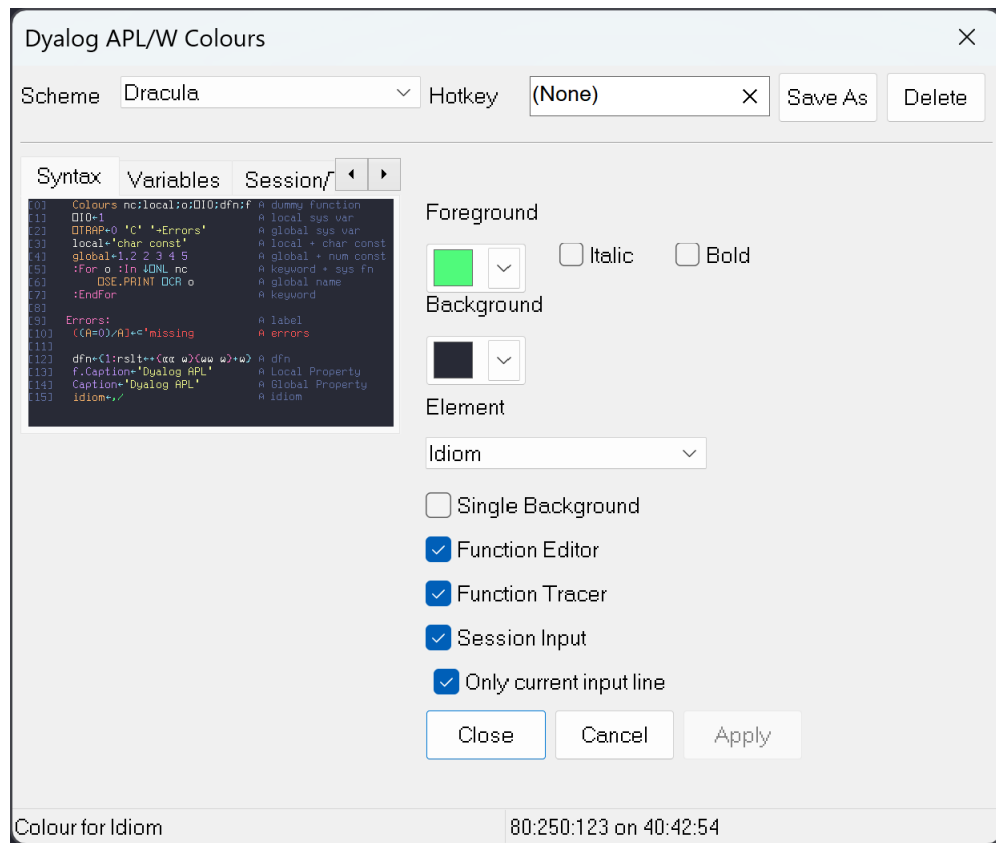
# Idiom List

- See [Programming Reference Guide](#) for full list
- Use syntax highlighting to help spot

| Idiom                                | Description   |
|--------------------------------------|---|
| $XA1 \uparrow + XA2$                 | $XA1$ redefined to be $XA1$ with $XA2$ catenated along its first axis                                     |
| $\{(\uparrow \downarrow \omega)\}XA$ | $XA$ sorted into ascending order  |
| $\{(\uparrow \downarrow \omega)\}XA$ | $XA$ sorted into descending order   |
| $\{\omega[\downarrow \omega]\}XV$    | $XV$ sorted into ascending order  |
| $\{\omega[\downarrow \omega]\}XV$    | $XV$ sorted into descending order   |
| $\{\omega[\downarrow \omega;]\}XM$   | $XM$ with the rows sorted into ascending  |
| $\{\omega[\downarrow \omega;]\}XM$   | $XM$ with the rows sorted into descending order   |
| $1 = \#XA$                           | 1 if $XA$ has a depth of 1 (simple array), 0 otherwise  |
| $1 = \# , XA$                        | 1 if $XA$ has a depth of 0 or 1 (simple scalar, vector, etc.), 0 otherwise                                |
| $0 \in \rho XA$                      | 1 if $XA$ is empty, 0 otherwise   |
| $\sim 0 \in \rho XA$                 | 1 if $XA$ is not empty, 0 otherwise   |
| $\uparrow \# XA$                     | The first sub-array along the first axis of $XA$  |
| $\sim \uparrow XA$                   | The first sub-array along the last axis of $XA$   |
| $\uparrow \# XA$                     | The last sub-array along the first axis of $XA$   |
| $\sim \uparrow XA$                   | The last sub-array along the last axis of $XA$  |
| $\ast \circ NA$                      | Euler's idiom (accurate when $NA$ is a multiple of $0.5$ )  |
| $0 = \rho XA$                        | 1 if $XA$ has an empty first dimension, 0 otherwise ( $\square ML < 2$ )                                  |
| $0 \neq \rho XA$                     | 1 if $XA$ does not have an empty first dimension, 0 otherwise ( $\square ML < 2$ )                        |
| $\square AV \uparrow CA$             | Classic version only: The character numbers (atomic vector index) corresponding to the characters in $CA$ |
| $\lfloor 0.5 + NA$                   | Round to nearest integer  |

# Idiom List

- Use syntax highlighting to help identify





# Set Phrases — key $F \in Y$

$\{c\omega\}$      $\{\alpha\omega\}$      $\{\alpha\}$      $\neg$

$\{\neq \omega\}$      $\{\alpha(\neq \omega)\}$      $\{\alpha, \neq \omega\}$   
 $\{\neq \cup \omega\}$      $\{\alpha(\neq \cup \omega)\}$      $\{\alpha, \neq \cup \omega\}$

$\{F/\omega\}$      $\{\alpha(F/\omega)\}$      $\{\alpha, F/\omega\}$   
 $\{F \neq \omega\}$      $\{\alpha(F \neq \omega)\}$      $\{\alpha, F \neq \omega\}$

for Boolean  $Y$  if  $F$  is one of  $\wedge \vee = \neq +$   
for numeric  $Y$  if  $F$  is one of  $+ [ |$

# Set Phrases — key F

$\{c/w\}$                        $\{\neq/w\}$   
 $\{+/w\}$                        $\{\Gamma/w\}$                        $\{L/w\}$   
 $\{\alpha, +/w\}$                        $\{\alpha(f/w)\}$



# Set Phrases — key F

```

    {<w}           {≠w}
    {+/w}         {|/w}         {[/w}
  
```

```

RL←42 ◊ key←(A[?30p8]),,(?30p5) ◊ (≠,≠∘u)key
  
```

30 20

```

]runtime -c "{≠w}key" "{≠+w}key"
  
```

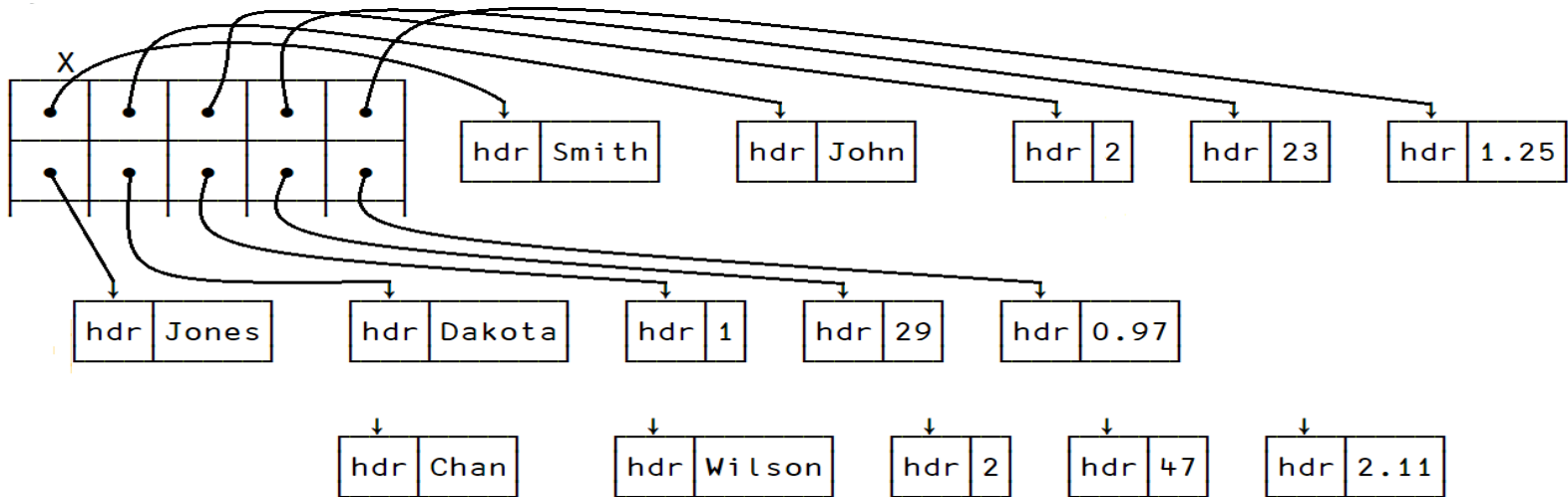
```

{≠w}key → 1.2E-6 | 0% 
{≠+w}key → 3.2E-5 | +2626% ████████████████████████████████████████████████████████████████████████████
  
```

# Inverted Tables

- ◆ In APL, each item of a nested array is a pointer to an array, and every array has a header and data. On a 64-bit system, a pointer takes 64 bits.
- ◆ This can be a rather inefficient way to store data

# Pointer bloat in “verted” table



# Size of a verted table

$$\text{size} \leftarrow \{24 + (8 \times \# \rho \omega) + 8 \times \lceil 64 \div \sim (x / \rho \omega) \times \lfloor 0.1 \times \square \text{dr } \omega \rfloor \}$$

$\lfloor 0.1 \times \square \text{dr } \omega \rfloor$  # of bits per item

$x / \rho \omega$  # of items

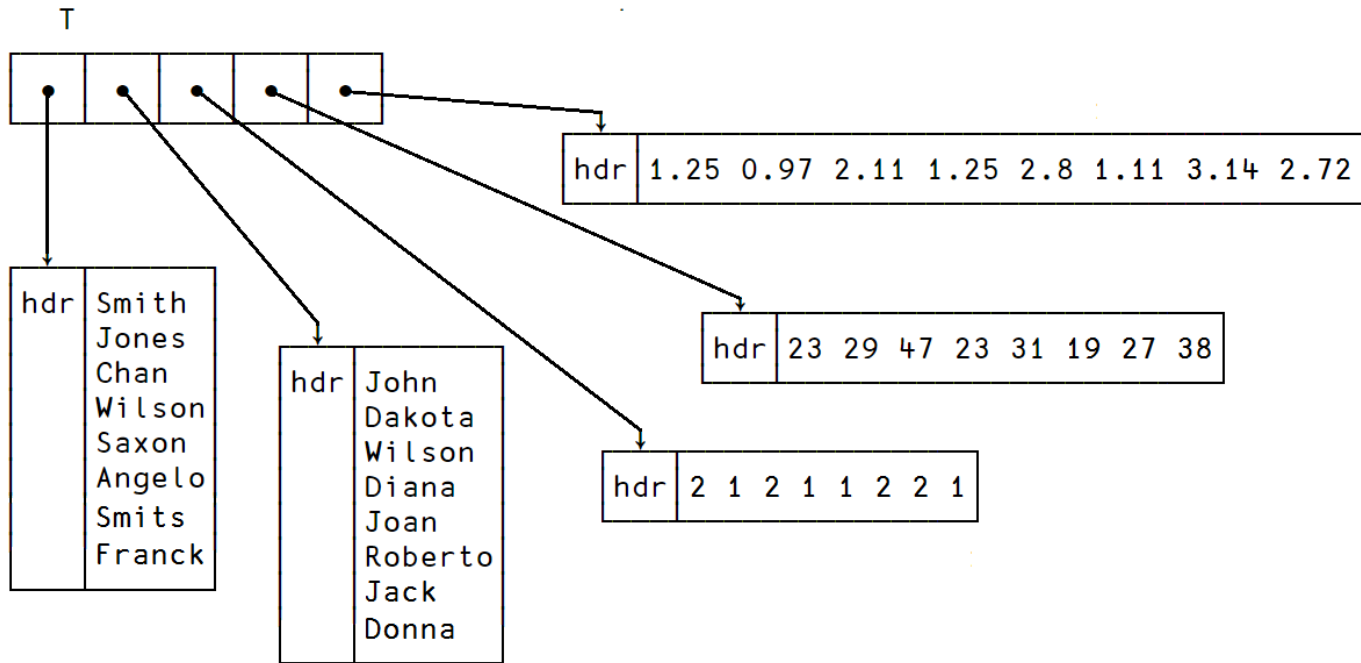
$8 \times \lceil 64 \div \sim$  # of bytes for the items, rounded up to the next multiple of 8

$24 + (8 \times \# \rho \omega)$  # of bytes for the header

# Size of an “Inverted” table

- ◆ In contrast, for an inverted table, each column is a simple array, so
  - ◆ the number of pointers is the same as the number of columns
  - ◆ the number of headers is the same as the number of columns, plus 1 for the table itself.

Moreover, the data for each column are contiguous in memory, resulting in faster access.



# Exercise: Invert a regular table

- Load in the file as a regular table
  - CSV path  $\theta$  4
- Check the size of the table
- Transform the table to an inverted style
- What's the size difference?

# Exercise: Try lookup of IVT vs VT

- 8I



# Data table normalization

- Write a function that replaces infrequently occurring items with a shorter id
  - $(Ux)_ix$

# Parallelization: Automatic

- ◆ APL is implicitly a Parallel language
- ◆ Many of the primitives already automatically parallelized by the interpreter

# Not as simple

- ◆ Overhead in parallelization not worth it if your arrays are small
  - ◆ Most APL arrays are small!
- ◆ Parallel programming requires care – many applications depend on side effects and will have race conditions

# Green Threads

- ◆ & operator
- ◆ F & arg
- ◆ Not a real OS thread, the interpreter handles multitasking
- ◆ Useful for running a program in the background, without waiting (asynchronous)

&

- ◆ Immediately returns the TID
- ◆ Look into `□TPUT/□TGET` (tokens) for communicating between threads
- ◆ `□TSYNC` for synchronization

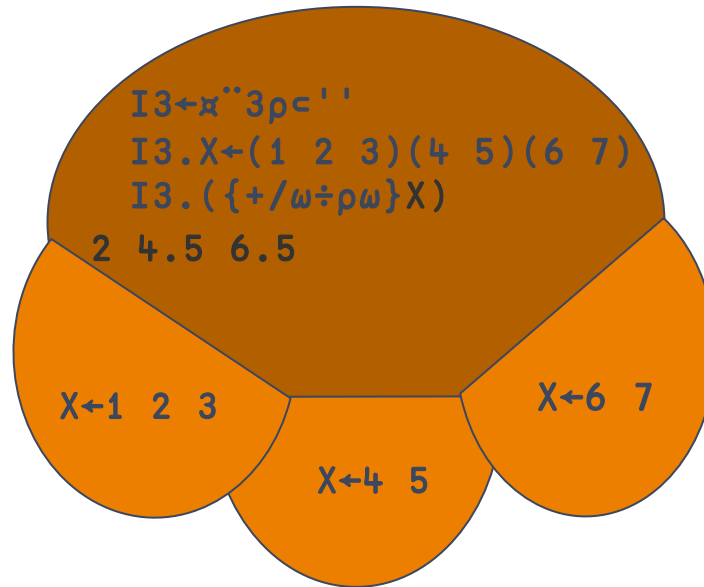
# Quiz

- How long do the following expressions take to return and run?
  - `DL 1`
  - `DL 1 2`
  - `DL& 1 2`
  - `TSYNC DL& 1 2`
  - `DL II 1 2`

# Isolates

- ◆ Isolates are a form of explicit parallelization
  - ◆ The task is left to the programmer to decide when things ought to be run in parallel
- ◆ Different from “Green Threads”(&)
  - ◆ Actual OS processes used

# Isolates





# Exercise

The cosine similarity encodes correlation between two vectors

```
◦.CosineSimilarity~(0 0 1)(0 1 1)(1 0 0)
1          0.7071067812 0
0.7071067812 1          0
0           0           1
```

# Exercise

The cosine similarity encodes correlation between two vectors

```
CosineSimilarity ← {  
  Abs ← { 0.5 * (α + ω) / (α * ω)  
  (α + ω) ÷ α * Abs ω  
}
```

# Exercise

- Run the cosine similarity using isolates, chunking the data into partitions first

# Exercise

- ◆ Is there a faster way to write the earlier function, without isolates?

# Overcomputing

## Example: Maximum Difference

Given a list of numbers, compute the largest difference between any two numbers in the list.

$$\{\lceil /, \circ . - \sim \omega \}$$
$$\{\alpha \leftarrow 0 \ \diamond \ 0 \in \rho \omega : \alpha \ \diamond \ (\alpha \lceil \lceil / \mid \omega - \rhd \omega) \nabla 1 \downarrow \omega \}$$



# Practical notes

- ◆ Don't get lost in micro-optimizations
- ◆ Use judgement on code clarity vs performance