*Dyalog '20*

# How I Won the APL Problem Solving Competition 2020

**Andrii Makukha**,
University of Hong Kong,
Hong Kong S.A.R.

# About Me

# About Me

- **Programming since the age of 9**
  - QBasic and FoxPro – first programming languages
- **Occasional participant of programming contests**
- **10+ years of professional software development experience**
- **Main languages: C, (modern) C++, Python**
  - Newest favourite: APL!

# My Encounters With APL

- **2007 – first heard about APL during my undergraduate studies**
  - APL was used by SimCorp Ukraine
  - SimCorp Ukraine employed many graduates of my school (Kyiv University)
- **2019 – noticed APL solutions on Code Golf Stack Exchange**
- **June 2020 – started to learn APL specifically for this competition**
  - Still a newbie! I cannot read tacit functions without tools.
  - But APL has already influenced the way I write code in C++ and Python.

# APL Problem Solving Competition 2020

# What's so special about this competition?

- **Unlike IOI and ICPC-style programming contests:**
  - Long-term
  - Style of your code influences your chances to win
- **Two phases** – two independent contests:
  - **Phase I** – 10 one-liner tasks (very fun and quickly got me hooked on APL)
  - **Phase II** – 9 problems of varying difficulty

# Phase II – Problems Overview

- **One-liner problems:**
  - Dive score (P.1), Steps (P.2)
- **Scripting-style problems:**
  - Past tasks (P.3), Merge (P.6), UPC barcode (P.7)
- **Performance-style problems:**
  - Bioinformatics (P.4), Cashflows (P.5), Partitioning (P.8), Mobiles (P.9)

# How I won the contest?

**1) I was concentrated on the performance**

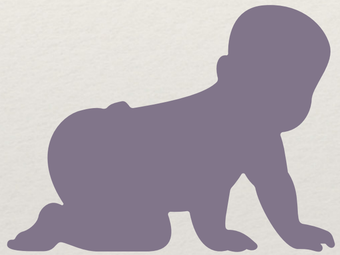
- I used function `cmpx` from the `dfns` workspace for profiling
- Always used direct functions (dfns) instead of traditional functions (tradfns)

**2) I tried to imitate good coding style**

- Comments were written in “literate programming” style
- I would find idioms at APLcart.info, APL Wiki, in books, papers, etc.

**3) I implemented unit tests and performance tests**

# Phase I*



** For the sake of completeness. I **didn't** win Phase I*

# Phase I: My Solution

```
 P1 ← {(⌽⍣(⍺<0))(⍺↑⍵)(⍺↓⍵)}
 P2 ← (128∘≤⍲≤∘191)⊂⊢
 P3 ← 26⊥⎕A⍳⊢
 P4 ← {2 0∊⍨+⌿0<4 100 400∘.|⍵}
 P5 ← {(⊃⍵)+0,(×d)×⍳|d←--/⍵}
 P6 ← {(⍵~t),t←⍵~⍺}
 P7 ← {f←⌽(2∘⊥⍣¯1)⋄A≡A∧(≢A←f⍺)↑f⍵}
 P8 ← {(∧/2≠/b)∧(2</z)≡+\b←2>/z←(10∘⊥⍣¯1)⍵}
 P9 ← {m←(,⍵)⍳⌈/⍵ ⋄ ∧/(⍳∘≢≡⍋)⍤(m↑⍵)(⌽m↓⍵)}
P10 ← {⍉↑⊃,/↓∘⍉⍤⍵}
```

# Phase I: My Solution

```
P1 ← {(⌽⍣(⍺<0))(⍺↑⍵)(⍺↓⍵)}
P2 ← (128∘≤⍲≤∘191)⊂⊢
P3 ← 26⊥⎕A⍳⊢
P4 ← {2 0∊⍨+⌿0<4 100 400∘
P5 ← {(⊃⍵)+0,(×d)×⍳|d←
P6 ← {(⍵~t),t←⍵~⍺}
P7 ← {f←⌽(2∘⊥⍣¯1)⋄A≡A∧(≢A←
P8 ← {(∧/2≠/b)∧(2</z)≡+b←2>/z←(10∘⊥⍣¯1)⍵}
P9 ← {m←(,⍵)⍳⌈/⍵ ⋄ ∧/(⍳∘≢≡⍋)⍤(m↑⍵)(⌽m↓⍵)}
P10 ← {⍉↑⊃,/↓∘⍉⍤⍵}
```

In Phase I my main goal was conciseness, not performance.

# Phase I: Better Solution to Problem 8

```
P1  ← {(⍎⍣(⍺<0))(⍺↑⍵)(⍺↓⍵)}
P2  ← (128∘≤⍲≤∘191)⊂⊢
P3  ← 26⊥⎕A⍳⊢
P4  ← {2 0∊⍨+⌿0<4 100 400∘.|⍵}
P5  ← {(⊃⍵)+0,(×d)×⍳|d←--/⍵}
P6  ← {(⍵~t),t←⍵~⍺}
P7  ← {f←⍎(2∘⊥⍣¯1)⋄A≡A∧(≢A←fα)↑f
P8  ← {×/0>2×/2-/10(⊥⍣¯1)⍵}
P9  ← {m←(,⍵)⍳⌈/⍵ ⋄ ∧/(⍳∘≢≡⍋)⍨(m↑⍵)(
P10 ← {⍕↑⊃,/↓∘⍕⍨⍵}
```

A more elegant solution.
Due to @rak1507

# Phase II

# Problem 1. Take a Dive

# Problem 1: Shortened statement

Write a function named **`DiveScore`** that has the following syntax: **`score ← dd DiveScore scores`**, where:

- the left argument `dd` is a single number representing the degree of difficulty.
- the right argument `scores` is a numeric vector of length 3, 5, or 7 representing judges' scores from 0 to 10;
- the result is the score computed by removing 0, 2 or 4 outliers (respectively), totaling the rest and multiplying by degree of difficulty. The result must rounded to at most 2 decimal places.

Example:

```
   2.8 DiveScore 6.5 7 7 7.5 7.5 8 8
61.6
```

# Problem 1: My Solution

- Simple one-liner problem, similar to Phase I:

```
DiveScore←{
      2(⍦⍤⍧)⍺×+/⍵[3↑(2÷⍨¯3+≠⍵)↓⍋⍵]
}
```

# Problem 1: My Solution

- Simple one-liner problem, similar to Phase I:

```
DiveScore←{
      2(⍦⍤⍧)⍺×+/⍵[3↑(2÷⍨¯3+≢⍵)↓⍋⍵]
}
```

Calculate number of outliers to drop

# Problem 1: My Solution

- Simple one-liner problem, similar to Phase I:

```
DiveScore←{
        2(⍦⍤⍧)⍺×+/⍵[3↑(2÷⍨¯3+≠⍵)↓⍋⍵]
}
```

Take three indices in the middle

# Problem 1: My Solution

- Simple one-liner problem, similar to Phase I:

```
DiveScore←{
        2(⍢⊖)α×+/⍵[3↑⍒(2÷⍨¯3+≠⍵)↓⍋⍵]
}
```

2-train idiom for rounding;
same as just: ⍢2⊖α...

# Problem 2. Another Step in the Proper Direction

# Problem 2: Shortened statement

Write a function named **`Steps`** which has the following syntax: **`steps ← {p} Steps fromTo`**, where:

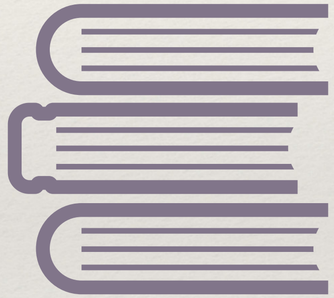
- the right argument `fromTo` is a 2-element integer vector representing the start and end points of the result.
- the optional left argument `p` is a single number as follows:
  - If `p` is **negative**, its absolute value represents the number of equally sized steps to take.
    - If `p` is not an integer, treat it like `⌊p`. (e.g., `¯4.7` is treated as `¯5`)
  - If `p` is **positive**, it represents the step size.
    - If `p` does not evenly divide `fromTo`, the last step will be `<p`.
  - If `p` is `0`, return the first element of `fromTo` (as if you took zero steps)
  - If `p` is **not provided**, return a vector of the integers from the first element of the right argument to the second, inclusively.
- the result is a numeric vector such that `fromTo≡(⊣/,⊢/)steps` and whose interior elements, if any, represent intermediate points.

# Problem 2: My Solution

- My solution is just four separate one-line solutions for each respective case:

```
Steps←{
      α←1
      α=0:ω[1]
      α=1:(⊃ω)+0,(×d)×⍳|d←--/ω
      α<0:(ω[2]@(n+1))(⊃ω)+(n÷⍨--/ω)×0,⍳n←|⌊α
      (ω[2]@(n+1))(⊃ω)+α×(×d)×0,⍳n←⌈α÷⍨|d←--/ω
 }
```

# Problem 3. Past Tasks Blast



# Problem 3: Shortened statement

Write a function named **`PastTasks`** which has the following syntax: **`urls ← PastTasks url`**, where:

- the right argument, url, is the character vector `'https://www.dyalog.com/student-competition.htm'`.
- the result is a vector of character vectors each representing a fully qualified URL for a PDF file linked on the web page. The order of the URLs is not significant.

# Problem 3: Beware! Don't just use regex

- Challenges & traps:
  - The problem says that the right argument is a *specific* URL of a web page with relatively stable content.

    → So there is a temptation to make assumptions about the future content of the web page and just use regular expressions;

    → Beware! This is a potential trap!
  - The content of the web page could *potentially* change arbitrarily (maybe even to intentionally break your solution?)

# Problem 3: Use XML parser!

- Therefore, it's important to use XML parser:
  - For example, what if the web page is changed and there is a new tag:

    <area shape="rect" coords="100,100,200,200" href="faq.pdf">

  - ...or a comment like this:

    <!-- <a href="booby_trap.pdf">Ha-ha! Gotcha!</a> -->

  - Using regex to find href="*.pdf" is **not sufficient**

# Problem 3: My Solution

```
PastTasks←{
      dat←⍎XML (#.HttpCommand.Get ⍵).Data
      getUris←{
          attrs←⊃⍪/dat[⍸((⊂,⍵)∘{⍵[2]≡⍺}⍤1)dat;4]
          0=≢attrs:⍬          ⍝ sanity check
          attrs[⍸({⍵[1]≡⊂'href'}⍤1)attrs;2]
      }
      0=≢uris←getUris 'a':⍬
      pdfs←{({⊃⍴('\.pdf$'⎕S 0⍠1)⍵}¨⍵)/⍵}uris
      base←{0≠≢⍵:⊃⍵ ⋄ ''} getUris 'base'
      (base,⊢)¨pdfs
 }
```

# Problem 3: My Solution

```
PastTasks←{
      dat←⎕XML (#.HttpCommand.Get ⍵).Data
      getUris←{
            attrs←⊃⍪/dat[⍸((⊂,⍵)∘{⍵[2]≡⍺}⍤1)dat;4]
            0=≢attrs:⍬
            attrs[⍸({⍵[1]≡⊂'hr
      }
      0=≢uris←getUris 'a':⍬
      pdfs←{({⊃⍴('\.pdf$'⎕S 0⍠1)⍵}¨⍵)/⍵}uris
      base←{0≠≢⍵:⊃⍵ ⋄ ''} getUris 'base'
      (base,⊢)¨pdfs
}
```

Make the solution work for an empty web page

# Problem 4. Bioinformatics

# Problem 4, Task 1: Shortened statement

Write a function named **`revp`** which has the following syntax: **`r ← revp dna`**, where:

- the right argument `dna` is a character vector representing a DNA string;
- the result is a 2-column numeric matrix of the [;1] positions [;2] lengths of all the *reverse palindromes* of length between 4 and 12 in the input DNA string.

Definitions:

- A **reverse palindrome** is a section of DNA that matches the reverse of its *complement*.
- The **complement** of a DNA string swaps 'A' for 'T', 'T' for 'A', 'C' for 'G', and 'G' for 'C'.
  - Example: the complement of 'ATCG' is 'TAGC', so its reverse complement is 'CGAT'.

*Source:* http://rosalind.info/problems/revp/

# Problem 4, Task 1: Beware!

- What if the input string is too short?
  - Less than 4 characters?
  - Less than 12 characters?
- Optimization:
  - No need to check lengths 5, 7, 9, 11, because reverse palindromes of odd lengths do not exist

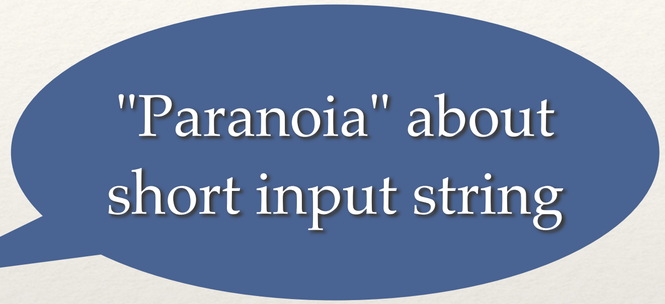
# Problem 4, Task 1: My Solution

- A straightforward solution:

```
revp←{
      4>≢⍵:0 2⍴⍬
      pal←{⍵≡⌽'ATCG'['TAGC'⍳⍵]}
      k←{⍵(⊢,⊣)¨⍳pal¨⍵,/⍺}
      ↑⊃,/⍵∘k¨(2+2×⍳5⌊2÷⍨¯2+≢⍵)
 }
```

# Problem 4, Task 1: My Solution

- A straightforward solution:




```
revp←{
      4>≢⍵:0 2⍴⍬
      pal←{⍵≡⌽'ATCG'['TAGC'⍳⍵]}
      k←{⍵(⊢,⊣)¨⍸pal¨⍵,⍵/⍺}
      ↑⊃,/⍵∘k¨(2+2×⍳5)[⌊2÷⍨¯2+≢⍵]
 }
```

# Problem 4, Task 1: My Solution

- A straightforward solution:

```
revp←{
      4>≢⍵:0 2⍴⍬
      pal←{⍵≡⌽'ATCG'['TAGC'⍳⍵]}
      k←{⍵(⊢,⊣)⍨⍳⍨pal⍨⍵,/⍺}
      ↑⊃,/⍵∘k⍨(2+2×⍳5)|2÷⍨¯2+≢⍵)
 }
```

Checks if the argument is a reverse palindrome

# Problem 4, Task 1: My Solution

- A straightforward solution:

```
revp←{
      4>≢⍵:0 2⍴⍬
      pal←{⍵≡⌽'ATCG'['TAGC'⍳⍵]}
      k←{⍵(⊢,⊣)¨⍳pal¨⍵,/⍺}
      ↑⊃,/⍵∘k¨(2+2×⍳5⌊⌊2÷⍨¯2+≢⍵)
 }
```

Find all substrings of α of length ω which are reverse palindromes

# Problem 4, Task 1: My Solution

- A straightforward solution:

```
revp←{
      4>≢⍵:0 2⍴⍬
      pal←{⍵≡⌽'ATCG'['TAGC'⍳⍵]}
      k←{⍵(⊢,⊣)¨⍳pal¨⍵,/⍺}
      ↑⊃,/⍵∘k¨(2+2×⍳5⌊2÷⍨¯2+≢⍵)
 }
```

Calculate a number of lengths to check

# Problem 4, Task 1: My Solution

- A straightforward solution:

```
revp←{
      4>≢⍵:0 2⍴⍬
      pal←{⍵≡⌽'ATCG'['TAGC'⍳⍵]}
      k←{⍵(⊢,⊣)¨⍳pal¨⍵,/⍺}
      ↑⊃,/⍵∘k¨(2+2×⍳5⌊2÷⍨¯2+≢⍵)
 }
```

Typically:
4 6 8 10 12
(if 12≤≢⍵)

# Problem 4, Task 1: Fast solution by Sam Weiss

- Solution by Sam Weiss. It runs **10 times faster** than my solution:

```
revp←{
      d←'AC GT'
      e←¯3+d⍳⍵
      i←{↑⍵,/⍳⍺}
      m←(≢⍵)i⍤2+2×⍳5
      n←e∘(⊣[]≍∘⊂)⍤m
      ↑↑,/(((⍸(∧/0∘=)),⍤¯1∘↑∘⍴)(⊢-+⌽))⍤n
 }
```

# Problem 4, Task 2: Shortened statement

Write a function named **`sset`** which has the following syntax: **`r ← sset n`**, where:

- the right argument `n` is a positive integer number ≤1000.
- the result is an integer of the total number of subsets that can be made from a set of `n` elements modulo 1,000,000.

Example:

```
      sset 857
551872
```

*Source:* http://rosalind.info/problems/sset/

# Problem 4, Task 2: Slow Solution

- Pure mathematics solution (will not work in practice):

```
sset←{1e6|2*⍵}
```

- Straightforward APL solution:

```
sset←{({1e6|2×⍵}*⍵)1}
```

Time complexity: O(N)

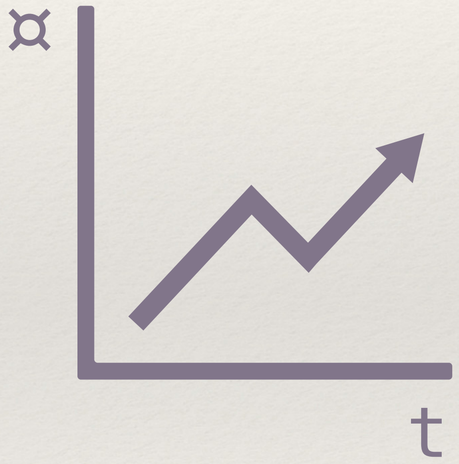
# Problem 4, Task 2: My Solution

- Exponentiation by squaring solution:

```
sset←{⍵>1:1e6|(1+2|⍵)×(∇⌊⍵÷2)*2 ⋄ 2}
```

- Time complexity: O(log N)
- Any intermediate value is only up to 41 bits long → answer is precise
  - Well within 52-bit fraction of double floats which are used in Dyalog APL

# Problem 5. Future and Present Value



# Problem 5, Task 1: Shortened statement

Write a function named **`rr`** which has the following syntax: **`r ← amounts rr rates`**, where:

- the right argument, `rates`, is a numeric vector of interest rates where the first value is 0.
- the left argument, `amounts`, is numeric vector of deposit and withdrawal amounts, the first value is the initial deposit.
- the result is a numeric vector of the cumulative values.

Note: Non-looping solutions will be given higher credit.

Example:

```
      2⍤ 100 0 20 0 ¯10 0 0 rr 0 .03 .04 .06 .02 .02 .04
100.00 103.00 127.12 134.75 127.44 129.99 135.19
```

# Problem 5, Task 1: Shortened statement

Write a function named **`rr`** which has the following syntax: **`r ← amounts rr rates`**, where:

- the right argument, `rates`, is a numeric vector of interest rates where the first value is 0.
- the left argument, `amounts`, is numeric vector of deposit and withdrawal amounts, the first value is the initial deposit.
- the result is a numeric vector of the cumulative values.

**Note: Non-looping solutions will be given higher credit.**

Example:

```
      2⍤ 100 0 20 0 ¯10 0 0 rr 0 .03 .04 .06 .02 .02 .04
100.00 103.00 127.12 134.75 127.44 129.99 135.19
```

# Problem 5, Task 1: My Array-Based Solution

- My solution was meant to illustrate "array-based" thinking:

```
rr←{AR×+\α÷AR←×\1+ω}
```

# Problem 5, Task 1: My Array-Based Solution

- My solution was meant to illustrate "array-based" thinking:

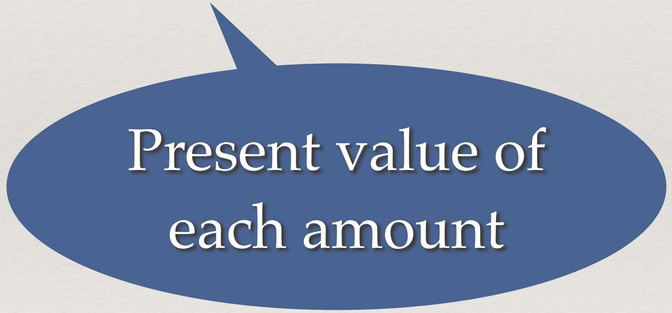
```
rr←{AR×+\α÷AR←×\1+ω}
```

Accumulated interest rate

# Problem 5, Task 1: My Array-Based Solution

- My solution was meant to illustrate "array-based" thinking:

```
rr←{AR×+\α÷AR←×\1+ω}
```



# Problem 5, Task 1: My Array-Based Solution

- My solution was meant to illustrate "array-based" thinking:

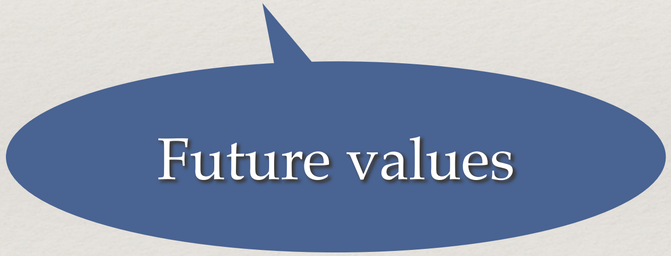
```
rr←{AR×+\α÷AR←×\1+ω}
```

Accumulated present values

# Problem 5, Task 1: My Array-Based Solution

- My solution was meant to illustrate "array-based" thinking:

```
rr←{AR×+\α÷AR←×\1+ω}
```



# Problem 5, Task 1: My Array-Based Solution

- My solution was meant to illustrate "array-based" thinking:

```
rr←{AR×+\⍺÷AR←×\1+⍵}
```

- **Pros:** O(N) time complexity, O(N) space complexity
- **Cons:** Unnecessary divisions

# Problem 5, Task 1: Alternative Solution

```
rr←{
    rec←{
        0=≢∈ω:α
        r M←(∈1↑ω)(1↓ω)
        (α,r[1]+r[2]×⊃⌽α)∇ M
    }
    ⍬ rec⍣↑(α)(1+ω)
}
```

- **Pros:** no division, still O(N) time and space complexity
- **Cons:** Slower and slightly more complex

# Problem 5, Task 2: Shortened statement

Write a function named **pv** which will calculate the present value of a cashflow and has the following syntax: **`r ← cashFlow pv rates`**, where:

- the right argument, `rates`, is a numeric vector of interest rates where the first value is 0.
- the left argument, `cashFlow`, is numeric vector representing a cash flow.
- the result is a single number representing the *present value* of the cash flow.

Note: Non-looping solutions will be given higher credit.

Example:

```
      ¯6200 ¯2000 3400 3850 4300 4750 pv 0 .03 .04 .06 .02 .02
6156.480816
```

# Problem 5, Task 2: My Solution

- Straightforward calculation of present values:

```
pv←{+/α÷×\1+ω}
```

# Problem 5, Task 2: My Solution

- Straightforward calculation of present values:

```
pv←{+/α÷×\1+ω}
```

Accumulated interest rate

# Problem 5, Task 2: My Solution

- Straightforward calculation of present values:

```
pv←{+/α÷×\1+ω}
```

Present value of each amount

# Problem 5, Task 2: My Solution

- Straightforward calculation of present values:

```
pv←{+/α÷×\1+ω}
```

Present value of the entire cashflow

# Problem 6. Merge

# Problem 6: Shortened statement

Write a function named **`Merge`** which has the following syntax: **`text ← templateFile Merge jsonFile`**, where:

- the right argument, `jsonFile`, is a character vector representing the name of a JSON file containing an associative arrays with field values.
- the left argument, `templateFile`, is a character vector representing the name of a text file in which template fields, which are delimited by @, have to be replaced with values from the JSON file. (@ character itself if escaped as @@.)
- the result is a character vector representing the merged text.

# Problem 6: My Solution

```
Merge←{
      ns←⎕JSON⊃⎕NGET ⍵
      names←ns.⎕NL ¯2
      parts←1++\ats←'@'=template←⊃⎕NGET ⍺
      parts[⍸(≠parts)∧ats∧(2|parts)]-←1
      replace←{
          ('@'≠¯1↑⍵)∨(1≥≢⍵)∨'@'≠⊃⍵:⍵
          0=≢name←¯1↓1↓⍵:'@'
          (⊂name)∈names:⍎ns⍎name
          '???'
      }
      ⊃,/replace¨parts⊆template
 }
```

# Problem 6: My Solution

```
Merge←{
      ns←⎕JSON⊃⎕NGET ⍵
      names←ns.⎕NL ¯2
      parts←1++\ats←'@'=template←⊃⎕NGET ⍺
      parts[⍸(≠parts)∧ats∧(2|parts)] -← 1
      replace←{
          ('@'≠¯1↑⍵)∨(1≥≢⍵)∨'@'≠⊃⍵:⍵
          0=≢name←¯1↓1↓⍵:'@'
          (⊂name)∈names:⍎ns⍎name
          '???'
      }
      ⊃,/replace¨parts⊆template
}
```

Partition the template into substrings

# Problem 6: My Solution

```
Merge←{
      ns←⎕JSON⊃⎕NGET ⍵
      names←ns.⎕NL ¯2
      parts←1++\ats←'@'=template←⊃⎕NGET ⍺
      parts[⍸(≠parts)∧ats∧(2|parts)] -← 1
      replace←{
          ('@'≠¯1↑⍵)∨(1≥≢⍵)∨'@'≠⊃⍵:⍵
          0=≢name←¯1↓1↓⍵:'@'
          (⊂name)∈names:⍎ns⍎name
          '???'
      }
      ⊃,/replace¨parts⊆template
}
```

Decrement partition value at the position of every other @ sign

# Problem 6: My Solution

```
Merge←{
      ns←⎕JSON⊃⎕NGET ⍵
      names←ns.⎕NL ¯2
      parts←1++\ats←'@'=template←⊃⎕NGET ⍺
      parts[⍸(≠parts)∧ats∧(2|parts)]-←1
      replace←{
          ('@'≠¯1↑⍵)∨(1≥≢⍵)∨'@'≠⊃⍵:⍵
          0=≢name←¯1↓1↓⍵:'@'
          (⊂name)∈names:⍎ns⍎name
          '???'
      }
      ⊃,/replace¨parts⊆template
}
```

Replaces each template field with a value from namespace

# Problem 6: Elegant solution by Louis de Forcrand

- Due to Louis de Forcrand:

```
Merge←{
      ns←⎕JSON⊃⎕NGET ⍵
      getValue←{
          0=⍴⍵:,'@'      ⍝ '@@'              → ,'@'
          6::'???'       ⍝ ~⍵∊ns.⎕NL ¯2 → '???'
          ⍎ns⍕⍵          ⍝  ⍵∊ns.⎕NL ¯2 → ⍎ns.⍵
      }
      ∊getValue¨@(⍴⍴1 0⍤⍬)'@'(1↓¨=⊂⊢)⊃⎕NGET ⍺
}
```

# Problem 7. UPC

# Problem 7, Task 1: Shortened statement

Write a function named **`CheckDigit`** which has the following syntax: **`digit ← CheckDigit digits`**, where:

- the right argument, `digits`, is an 11-element integer vector representing the first 11 digits of the UPC barcode.
- the result is the integer check digit.

Examples:

```
      CheckDigit 2 3 4 5 2 3 4 5 2 3 4
7
```

# Problem 7, Task 1: My Solution

```
CheckDigit←{
    10|-(11⍴3 1)+.×11↑⍵
}
```

Derived straight from the definition:

1. Find dot product of the first 11 digits and the vector 3 1 3 ... 1 3.
2. Negate it and find the residue modulus 10

# Problem 7, Task 2: Shortened statement

Write a function named **`WriteUPC`** which has the following syntax: **`bits ← WriteUPC digits`** where:

- the right argument, `digits`, is an integer vector of 11 elements representing the digits (in left to right order) to be represented in the barcode.
- the result is a 95-element Boolean vector representing the modules of the UPC barcode in left to right order. If there is an error in digits like incorrect length, or element(s) not in 0-9, return ¯1.

# Problem 7, Task 2: My Solution

```
WriteUPC←{
     (1≠ρρω)∨11≠≢∈ω:¯1
     (∨/(⊢≠⌊)∨0∘≤⍤≤∘9)ω:¯1
     codes←((7ρ2)⊤⊢)¨13 25 19 61 35 49 47 59 55 11
     LR←∈codes[1+ω,CheckDigit ω]
     1 0 1,(42↑LR),0 1 0 1 0,(~42↓LR),1 0 1
}
```

# Problem 7, Task 3: Shortened statement

Write a function named **`ReadUPC`** which has the following syntax: **`digits ← ReadUPC bits`**, where:

- the right argument, `bits`, is a Boolean vector representing the scanned bits of the UPC barcode. `bits` could be the result of scanning from right to left or left to right.
- the result is an integer vector of the digits of the UPC barcode in left to right order. If there is an error in the barcode, like incorrect parity, incorrect number of bits, or the check digit is not correct, return ¯1.

# Problem 7, Task 3: My Solution

```
ReadUPC←{
    (1≠ρρω)∨95≠≢∈ω:¯1
    ~(∧/∈∘0 1)ω:¯1
    2|+/ω:¯1

    B left M right E←((1@1 4 46 51 93)95ρ0)⊂ω
    1365≠2⊥B,M,E:¯1
    LR←left{2|+/7↑α:α,~ω ⋄ (⌽ω),~⌽α}right
    digs←¯1+{13 25 19 61 35 49 47 59 55 11⍳2⊥ω}¨(84ρ1 0 0 0 0 0 0)⊂LR

    10∈digs:¯1
    (12⊃digs)≠CheckDigit 11↑digs:¯1
    digs
}
```

# Problem 8. Balancing the Scales

# Problem 8: Shortened statement

Write a function named **`Balance`** which has the following syntax: **`parts ← Balance nums`**, where:

- the right argument, `nums`, is an integer vector of 2 to 20 elements;
- the result is a 2-element vector of integer vectors where the sums of the elements are equal, and the concatenation of parts has the same elements as in `nums`. In other words, if possible, parts should satisfy the following:

```
=/+/¨parts                    ⍝ both parts have the same total
≡/{ω[⍋ω]}¨nums (∈parts)       ⍝ (∈parts) and nums have the same elements
```

  If `nums` cannot be split into 2 equally summed groups, return `0`.

# Problem 8: Beware!

- A classic problem:
  **number partitioning problem**
  - Special case of:
    **subset sum problem**
- One catch: misleading reference to scales in the title and image!
  - Problem statement **doesn't** say that numbers cannot be negative

# Problem 8: Many existing algorithms

Four algorithms:

- **Brute force**, $O(N \times 2^N)$
- **Dynamic programming**, $O(N \times Sum)$
- **Recursive** (branch & bound, greedy)
- **Horowitz—Sahni optimization**, $O(N \times 2^{N/2})$

# Problem 8: Brute Force – 1

**Solution:**

- Multiply the number vector by a matrix of all boolean masks
- Find if the needed number (half) is in the resulting vector

**Optimization:**

- Due to symmetry, always put the first number in the first half
  - Thus, can consider only 19-number vector  (running time reduced by 50%)

# Problem 8: Brute Force – 2

**Pros:**

- Straightforward
- Works fast enough for 20 numbers

**Cons:**

- Exponential time complexity: $O(N \times 2^N)$

# Problem 8: My Brute Force Solution

```
BalanceBrute←{
    half←2÷⍨+/nums
    sz←¯1+≢⍵                                ⍝ vector size (≤19)
    bin←(szρ2)⊤⊢                            ⍝ convert to bool
    G←bin ¯1+⍳2*sz                          ⍝ generate masks
    si←((1↓nums)+.×G)⍳half−⊃nums            ⍝ locate solution
    si>2*sz:0                               ⍝ no solution
    (mask/⍵)((~mask←1,bin ¯1+si)/⍵)         ⍝ return solution
}
```

# Problem 8: Dynamic Programming – 1

**Solution:**

- Build a matrix M:
  - one row for each possible sum of a part of the vector: 0..T
  - one column for each additional number considered: 0..N

# Problem 8: Dynamic Programming – 2

**Pros:**

- Works fast for vectors of small numbers
- Semi-polynomial time complexity: O(T×N), where T – sum of all numbers

**Cons:**

- Running time and memory use depend on the magnitude of numbers
  - e.g., solving "`Balance 1e12+ı20`" would require 3.2 PB of RAM
- Does't work for negative numbers

# Problem 8: Dynamic Programming Solution

```
BalanceDP←{
    half≠⌊half←2÷⍨+/⍵:0
    M←(1(1+≠⍵))⍴1                ⍝ first row is for weight 0 - all 1s
    ign←(⊂⍵){                    ⍝ calculate other rows
        s←⍵ ⋄ ⊢M⍪←(0,∨\⍺{s≤⍺:0 ⋄ M[s-⍺;⍵]}¨⍳≠⍺)
    }¨1+⍳half
    ~M[1+half;1+≠⍵]:0            ⍝ no solution
    mask←(half+1){               ⍝ build a solution mask
        0=≠⍵:0
        M[⍺;≠⍵]∨⍺=1:(⍺ ∇ ¯1↓⍵),0
        ((⍺-⊃¯1↑⍵)∇ ¯1↓⍵),1           ⍝ include rightmost weight
    }⍵
    (mask/⍵)((~mask)/⍵)
}
```

# Problem 8: Dynamic Programming Solution

```
BalanceDP←{
    half≠⌊half←2÷⍨+/⍵:0
    M←(1(1+≠⍵))⍴1
    ign←(⊂⍵){
        s←⍵ ⋄ ⊢M⍪←(0,∨\⍺{s≤⍺:0 ⋄ M[s-⍺;⍵]}¨⍳≠⍺)
    }¨1+⍳half
    ~M[1+half;1+≠⍵]:0          ⍝ no solution
    mask←(half+1){             ⍝ build a solution mask
        0=≠⍵:0
        M[⍺;≠⍵]∨⍺=1:(⍺ ∇ ¯1↓⍵),0
        ((⍺-⊃¯1↑⍵)∇ ¯1↓⍵),1          ⍝ include rightmost weight
    }⍵
    (mask/⍵)((~mask)/⍵)
}
```

Dynamic programming looks very neat in dfns!

# Problem 8: Recursive

**Pros:**

- Very fast for small numbers
  - In practice, faster than the dynamic programming solution

**Cons:**

- Running time depends on the magnitude of numbers
- Does't work for negative numbers

# Problem 8: My Recursive Solution

```
BalanceRecursive←{
    α←2÷⍨+/⍵ ⋄ α≠⌊half←α:0
    rem←1↓⌽+\⌽nums←⍵[⍒⍵]          ⍝ nums in descending order; sums of remaining
    rec←{
        α=half:⍵                   ⍝ arrived to a solution: return it
        (half<α)∨(≠nums)=≠⍵:0      ⍝ cap bounding or end of search: return 0
        half>α+rem[≠⍵]:0           ⍝ cup bounding (sum is too small): return 0
        r←(α+nums[1+≠⍵])∇ ⍵,1      ⍝ try to include the current number
        0≠r:r                      ⍝ solution found: return it
        α ∇ ⍵,0                    ⍝ try to exclude the current number
    }
    0≡mask←(⊃nums)rec,1:0          ⍝ find a solution; return 0 if not found
    mask←(≠nums)↑mask              ⍝ pad the mask with zeroes if needed
    (mask/nums)((~mask)/nums)      ⍝ split numbers according to mask
}
```

# Problem 8: Horowitz-Sahni optimization

**Pros:**

- Worst-case time complexity: $O(N \times 2^{N/2})$
  - Works fast enough for 40 numbers!

**Cons:**

- Constant running time no matter how "trivial" the case is (e.g., all ones).

# Problem 8: My Horowitz-Sahni Solution

```
BalanceSplits←{
    α←2÷⍨+/⍵ ⋄ α≠⌊half←α:⍬
    h2←s-h1←⌊2÷⍨s←≢⍵              ⍝ sizes of each set
    mask←(1@(h1?s))s⍴0            ⍝ mask for the first set
    s1 s2←(mask/⍵)((~mask)/⍵)     ⍝ the two sets
    G1←(h1⍴2)⊤⍳¯1+⍳2*h1
    G2←h1{α=⍵:G1 ⋄ (⍵⍴2)⊤⍳¯1+⍳2*⍵}h2
    g1←⊂⍋v1←s1+.×G1
    g2←⊂⍋v2←s2+.×G2
    it←{
        (0=≢α)∨0=≢⍵:⍬             ⍝ reached the end unsuccessfully: return ⍬
        s←(⊃⍵)+⊃α                  ⍝ add the two current sums
        s=half:(≢α)(≢⍵)            ⍝ solution found: return the positions
        s>half:α ∇ 1↓⍵             ⍝ current sum is too big: reduce
        (1↓α)∇ ⍵                   ⍝ current sum is too small: increase
    }
    ⍬≡r←(g1⌷v1)it⍤⌷g2⌷v2:⍬
    i1←(⊃g1)[1+(≢v1)-r[1]]
    i2←(⊃g2)[r[2]]
    a1←(G1[;i1]/s1),G2[;i2]/s2
    a2←((~G1[;i1])/s1),(~G2[;i2])/s2
    a1 a2
}
```

# Problem 8: I used three algorithms together

My final approach:

- Divide the input numbers by their GCD
  - e.g., input "100 200 300 400 500" becomes trivial:
    - "1 2 3 4 5", odd sum → no solution
- Use one of **three algorithms**, which performs best for that specific case
- Multiply a solution by the GCD

# Problem 8: My final combined solution

```
Balance←{
    nums←⍵÷gcd←{(-⍣(∧/0∘≥⍵))1⌈∨/⍵}⍵
    half≠⌊half←2÷⍨+/nums:0

    ⍝ Algorithm A: Recursive algorithm
    th←200÷2*(0.3×0⌈20-≠⍵)    ⍝ empiric threshold
    (12≤≠⍵)∧(half<th)∧(∧/0∘≤)nums: (gcd×⊢)BalanceRecursive nums

    ⍝ Algorithm B: Horowitz–Sahni algorithm
    15<≠⍵: (gcd×⊢)BalanceSplits nums

    ⍝ Algorithm C: Brute force algorithm
    (gcd×⊢)BalanceBrute nums
}
```

# Problem 8: My final combined solution

```
Balance←{
    nums←⍵÷gcd←{(-⍣(∧/0∘≥⍵))1⌈∨/⍵}⍵
    half≠⌊half←2÷⍨+/nums:0

    ⍝ Algorithm A: Recursive algorithm
    th←200÷2*(0.3×0⌈20-≠⍵)    ⍝ empiric threshold
    (12≤≠⍵)∧(half<th)∧(∧/0∘≤)nums: (gcd×⊢)BalanceRecursive nums

    ⍝ Algorithm B: Horowitz–Sahni algorithm
    15<≠⍵: (gcd×⊢)BalanceSplits nums

    ⍝ Algorithm C: Brute force algorithm
    (gcd×⊢)BalanceBrute nums
}
```

That's what software engineering is about! 😋

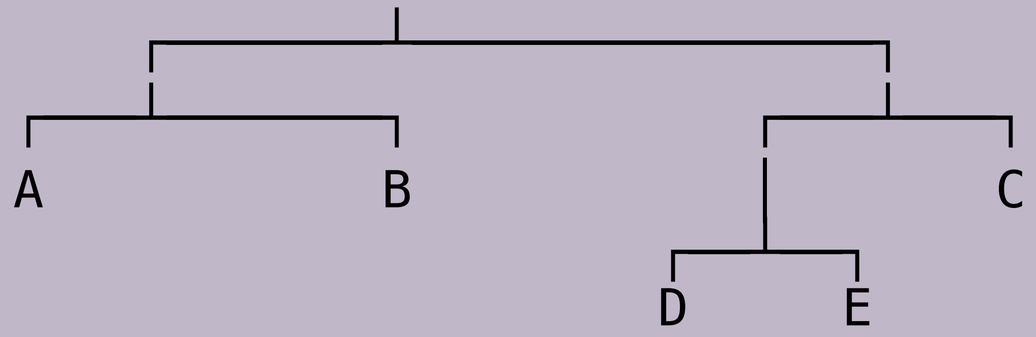
# Problem 9. Upwardly Mobile

# Problem 9: Shortened statement

Write a function named **`Weights`** which has the following syntax: **`weights ← Weights filename`**, where:

- the right argument, `filename`, is a character vector representing the name of a file with a diagram of a mobile (see example below).
- the result, `weights`, is a vector representing the list of smallest integers that will keep the mobile in balance. Weights should be in "alphabetic" order according to the diagram.

Example diagram:



# Problem 9: My Approach

- Read the mobile diagram from file
  - Optimization: remove all repeating lines which have only spaces and "|"
- Format the diagram into a matrix of characters (2D)
- Parse mobile recursively into a matrix
- Find inverse of the matrix and take its first row
- Divide by generalized GCD in order to get the lowest integer solution

# Problem 9: My full solution

```
Weights←{
    q←⍉((∧/∈∘'│ ')¨M)∧0,2≡/M←⊃⎕NGET ⍵ 1   ⍝ read diagram, find repeating lines
    M←⍉↑q/M.                ⍝ remove the repeats, format into a 2D matrix
    N←∪' ┌┴┐│'~⍨⍨∈M         ⍝ find the distinct weight names
    C←(1@1)(1(≢N))⍴0        ⍝ coef matrix; first weight is set to 1 (provisionally)
    descent←{                           ⍝ parses input vertically
        y←⍺+¯1+⌊/⍳'│'≠(⍺-1)↓M[;⍵]       ⍝ find first non-'│' from here down
        '┴'=M[y;⍵]:y explore ⍵          ⍝ explore new lever if fulcrum is found
        (1@(N⍳M[y;⍵]))(≢N)⍴0            ⍝ otherwise: turn weight name into a vector
    }
    explore←{                           ⍝ parses input horizontally
        d←(∈∘'┐┌')M[⍺;]                 ⍝ find all lever edges in the current row
        l r←(⍳∘1)¨(⍋(⍵-1)↑d)(⍵↓d)        ⍝ offsets of the left and right edges
        w1←(⍺+1)descent ⍵-l              ⍝ parse left sub-mobile
        w2←(⍺+1)descent ⍵+r              ⍝ parse right sub-mobile
        cfs←(l×w1)-r×w2                 ⍝ relationship between weights here (coefs)
        C,←cfs÷∨/cfs                    ⍝ divide the coefs by GCD and catenate to C
        w1+w2                           ⍝ return all the weights of this sub-mobile
    }
    x←⌊/M[1;]⍳'┴│'          ⍝ find the topmost link: the starting point for parsing
    ign←1 descent x         ⍝ parse the input to find the matrix C
    ,W÷∨/W←(⌹C)[;1]         ⍝ calculate integer solution
}
```

# Problem 9: First part of the solution

```
Weights←{
    q←~((∧/∈∘'│ ')¨M)∧0,2≡/M←⊃⎕NGET ⍵ 1   ⍝ find the repeats
    M←⍉↑q/M.                  ⍝ final diagram matrix (2D)
    N←∪' ┌─┴┐│'~⍨∈M           ⍝ find the distinct weight names
    C←(1@1)(1(≢N))⍴0          ⍝ coef matrix; first weight is 1
    descent←{                              ⍝ parses input vertically
        y←⍺+¯1+⌊/⍳'│'≠(⍺-1)↓M[;⍵]          ⍝ find first non-'│' below
        '┴'=M[y;⍵]:y explore ⍵             ⍝ explore new lever
        (1@(N⍳M[y;⍵]))(≢N)⍴0               ⍝ turn weights into vector
    }
```

…

# Problem 9: Second part of the solution

```
...
      explore←{                          ⍝ parses input horizontally
          d←(∈∘'┐┌')M[α;]                ⍝ find all lever edges in the current row
          l r←(⍳∘1)¨(⌽(ω-1)↑d)(ω↓d)       ⍝ offsets of the left and right edges
          w1←(α+1)descent ω-l            ⍝ parse left sub-mobile
          w2←(α+1)descent ω+r            ⍝ parse right sub-mobile
          cfs←(l×w1)-r×w2                ⍝ relationship between weights here (coefs)
          C,←cfs÷∨/cfs                   ⍝ divide the coefs by GCD and catenate to C
          w1+w2                          ⍝ return all the weights of this sub-mobile
      }
      x←⌊/M[1;]⍳'┴│'                     ⍝ find the topmost link: the starting point
      ign←1 descent x                    ⍝ parse the input to find the matrix C
      ,W÷∨/W←(⌹C)[;1]                    ⍝ calculate integer solution
 }
```

# Finally

# Advice for future contestants

**1) Write code that is fast**

- Use function `cmpx` from the `dfns` workspace for profiling
- Use direct functions (dfns) instead of traditional functions (tradfns)
- Write automated performance tests
- Think about time complexity and space complexity
  - Study algorithms!

# Advice for future contestants

**2) Ensure that your code has good style**

- Study the existing practices and other people's code (e.g., `dfns.dws`)
- Look for idioms (APLcart.info, APL Wiki, etc.)

**3) Ensure correctness**

- Write unit tests
- Always use: `]Boxing on -trains=tree`

# Links

- My solutions' GitHub repository: https://github.com/amakukha/apl-contest-2020
- Solutions by other people cited in this talk:
  - Sam Weiss: https://gist.github.com/saweiss/2e134bff9539e17ad4d6dbde58ada219
  - Louis de Forcrand: https://github.com/Olius/Contest2020/
  - rak1507: https://github.com/rak1507/Various-APL-Stuff/

# Thank you!

# Q & A