# Dynamic Programming - with grammars, algebras, products

Stefanie Schirmer
@linse

# combinatorial optimization problems

**combinatorial** counting/enumerating all possible solutions of a recursive problem

**optimization** finding the desired solution

# example problems

boggle,
money changing problem,
text / sequence alignment,
RNA structure prediction

# classic dynamic programming

# classic dynamic programming

1) characterize structure of optimal solution
2) recursively define value of optimal solution
3) compute value of optimal solution
4) construct opt. solution from computed info

# classic dynamic programming

1) characterize **structure** of optimal solution

$n = 5$

count decompositions?

# classic dynamic programming

2) recursively define value of optimal solution

$$D[n] = D[n-1] + D[n-3] + D[n-4]$$

$$D[0] = 0$$

1

3

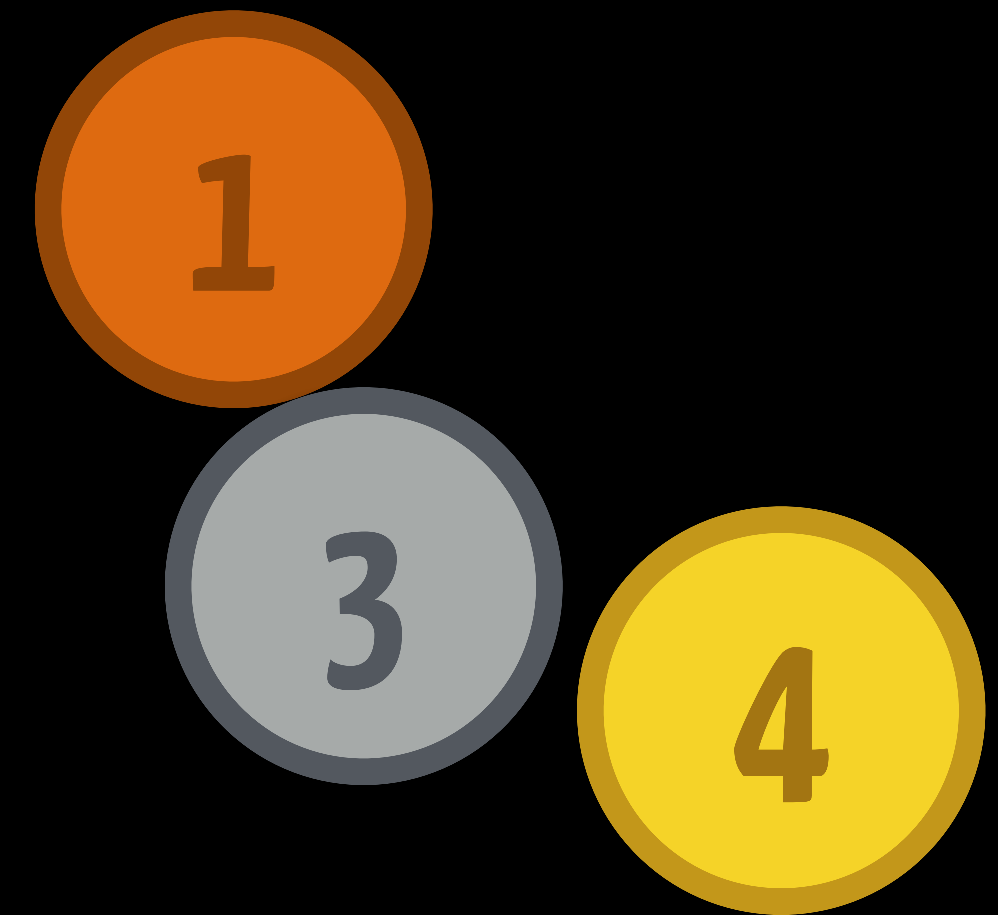
4

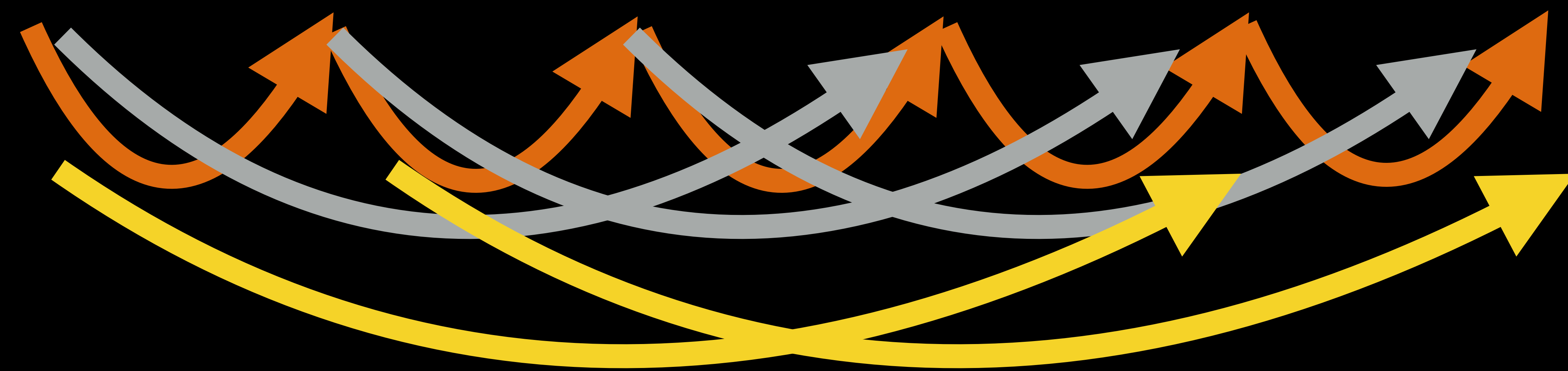
# classic dynamic programming

3) compute value of optimal solution

| n | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| D | 0 | 1 | 1 | 2 | 4 | 6 |

n = 5



# classic dynamic programming

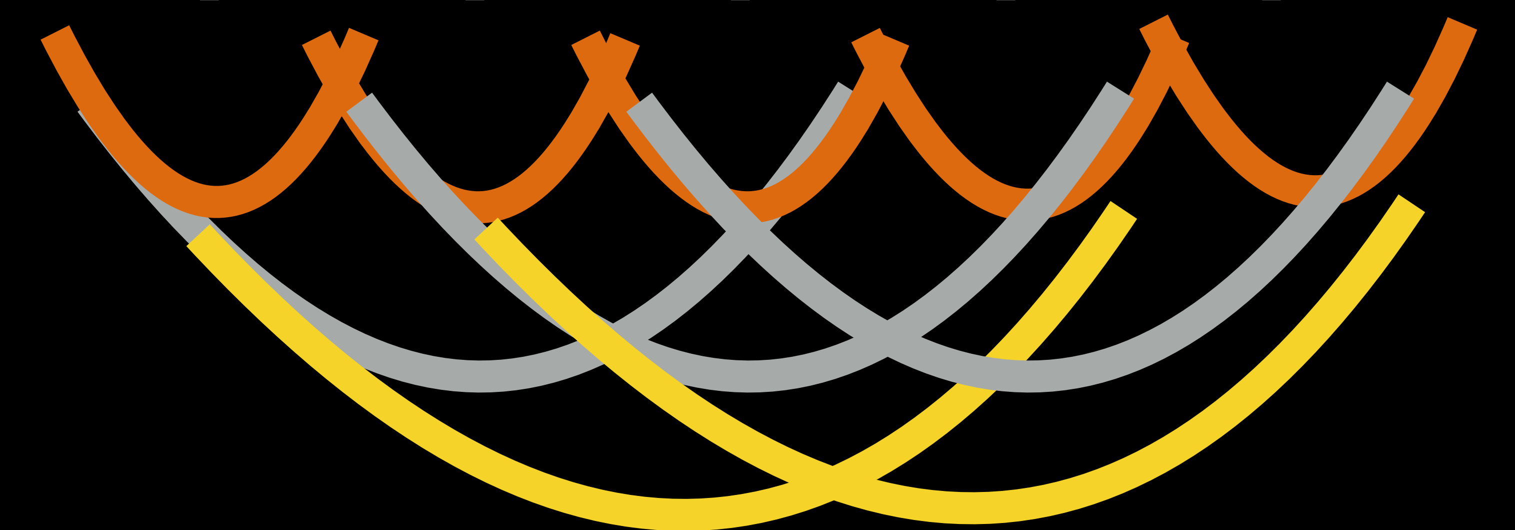
4) construct opt. solution from computed info

1+1+1+1+1,

1+1+3, 1+3+1, 3+1+1,

1+4, 4+1

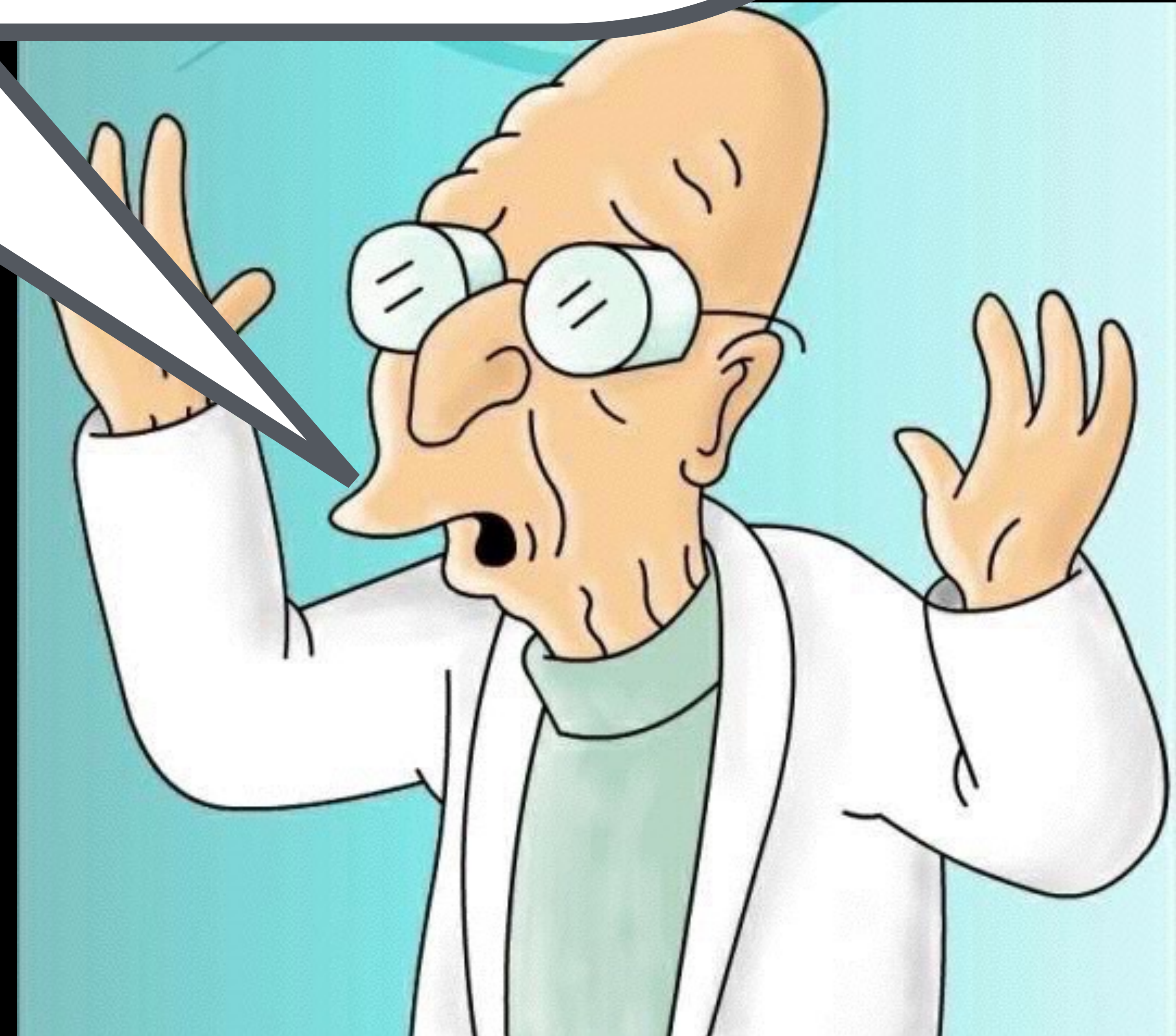
| n | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| D | 0 | 1 | 1 | 2 | 4 | 6 |



DP solves optimization problems

- over a large (exponential) search space
- in a reasonable (polynomial) time

The development of successful DP recurrences is a matter of "experience, talent and luck".



Life, the universe and all the rest

Candidates are trees
Questions are algebras
Programs are grammars

Products are fun !!!

# life, the universe and all the rest

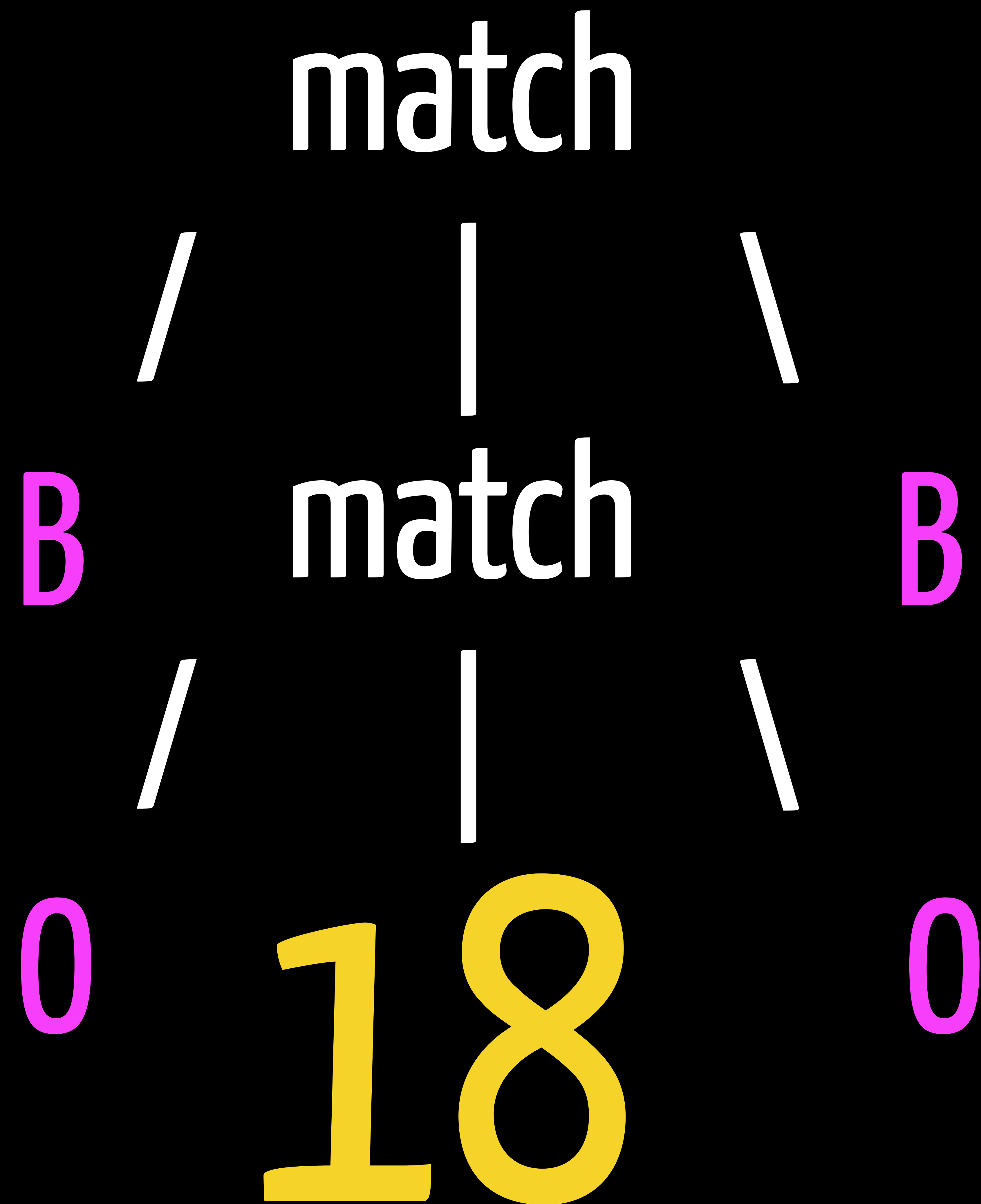
# reverse engineering of DP problems

If this is the answer..

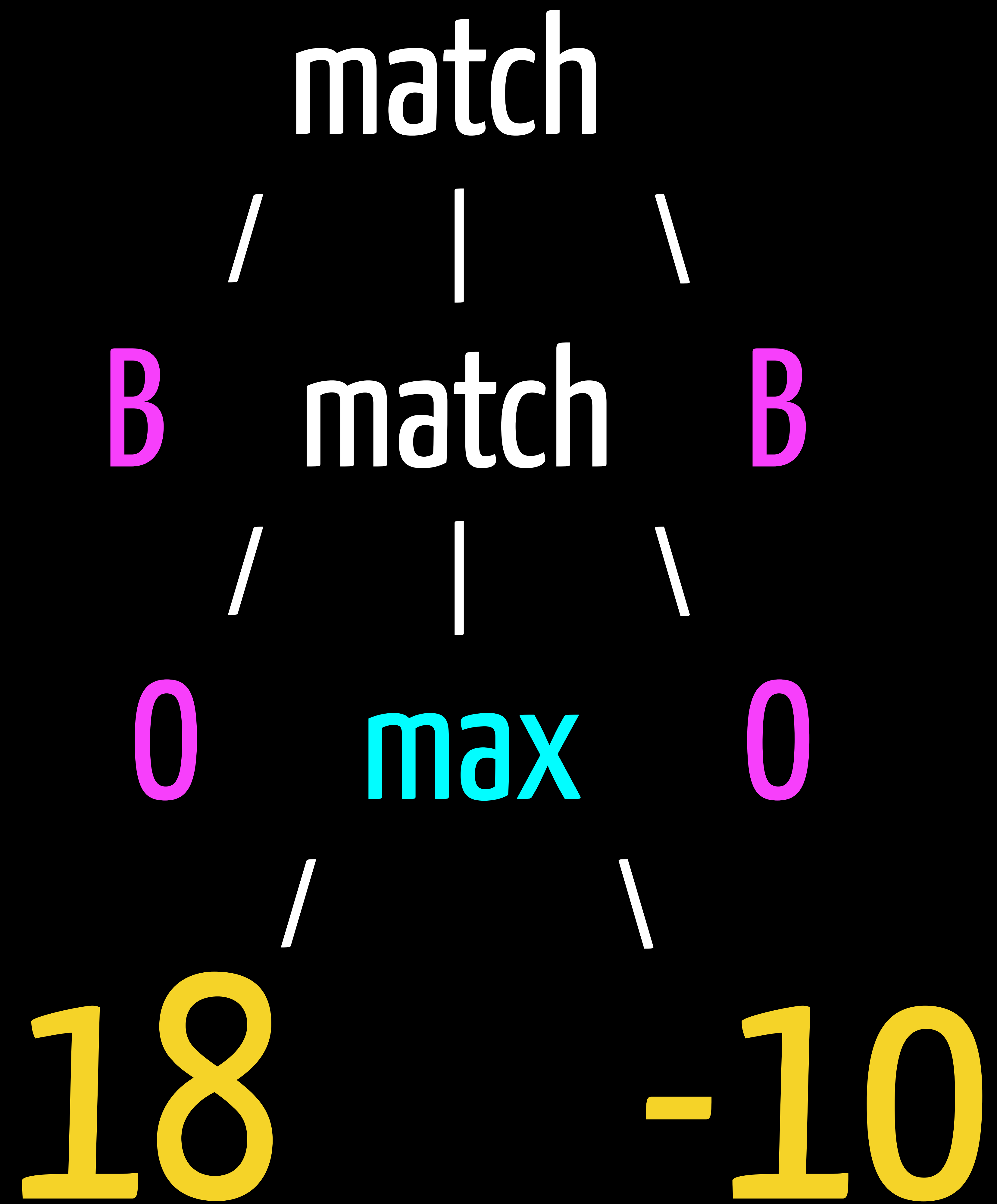
42

match

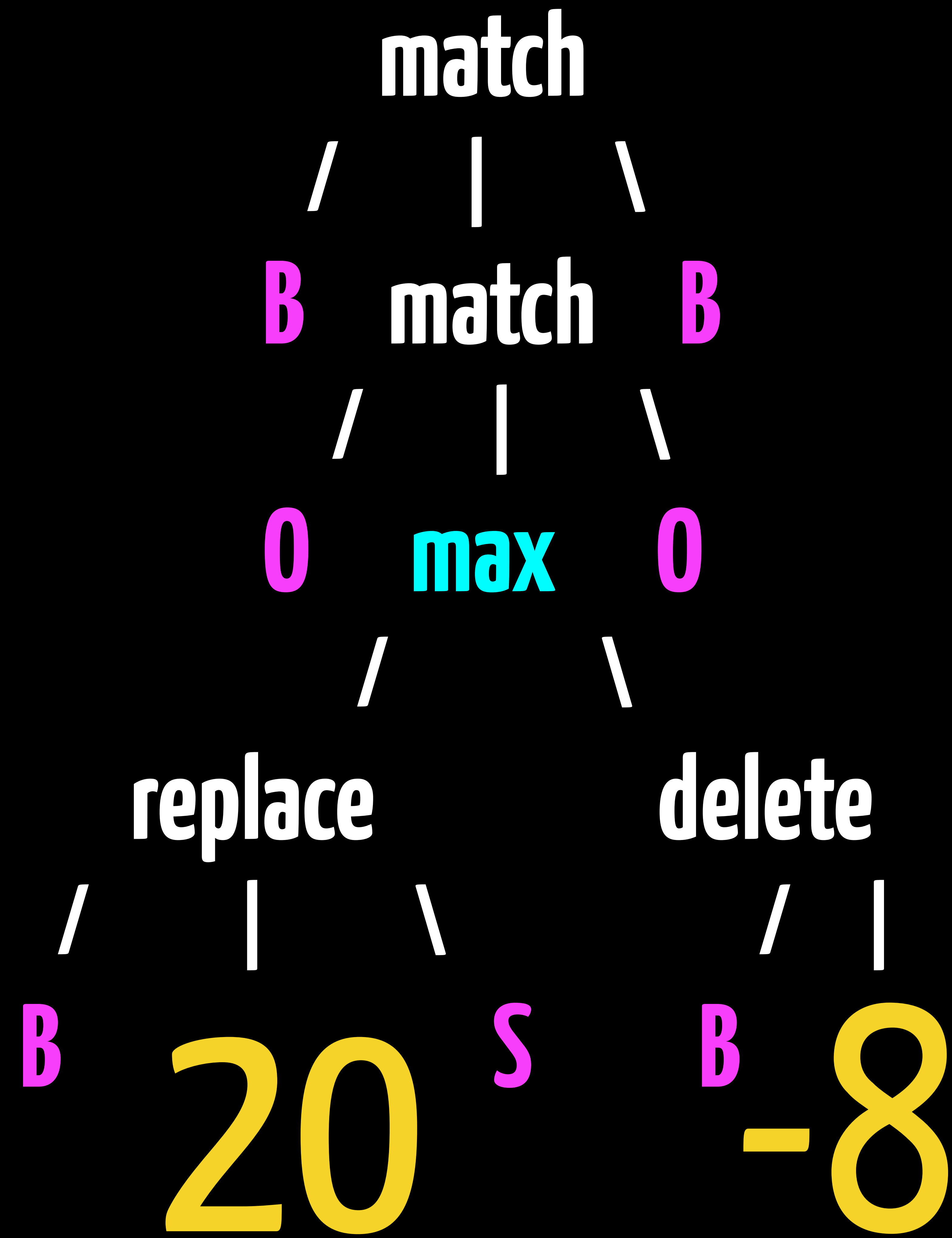
match(a, x, a) = x+12



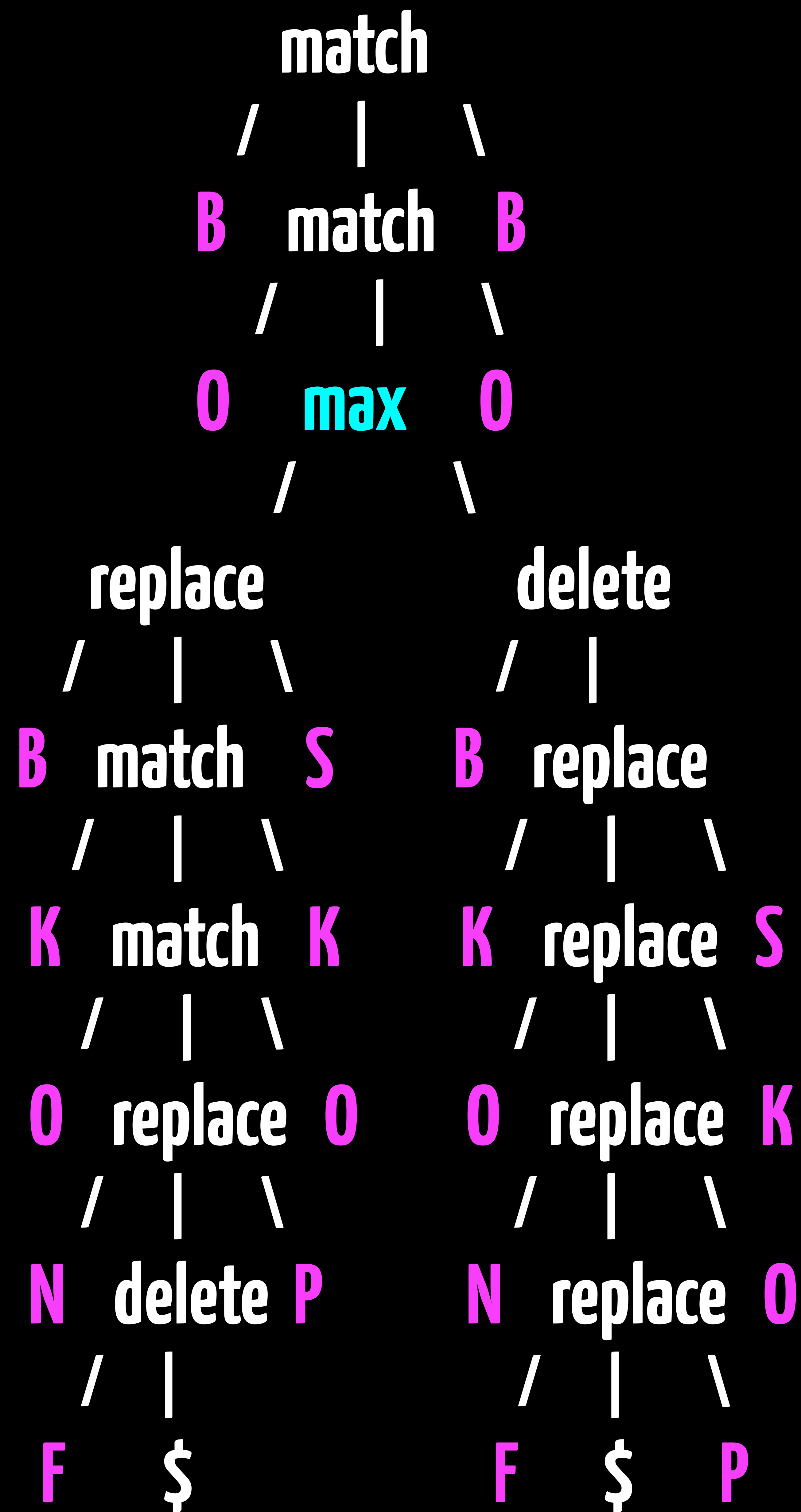

match(a, x, a) = x+12




choice of the maximum score



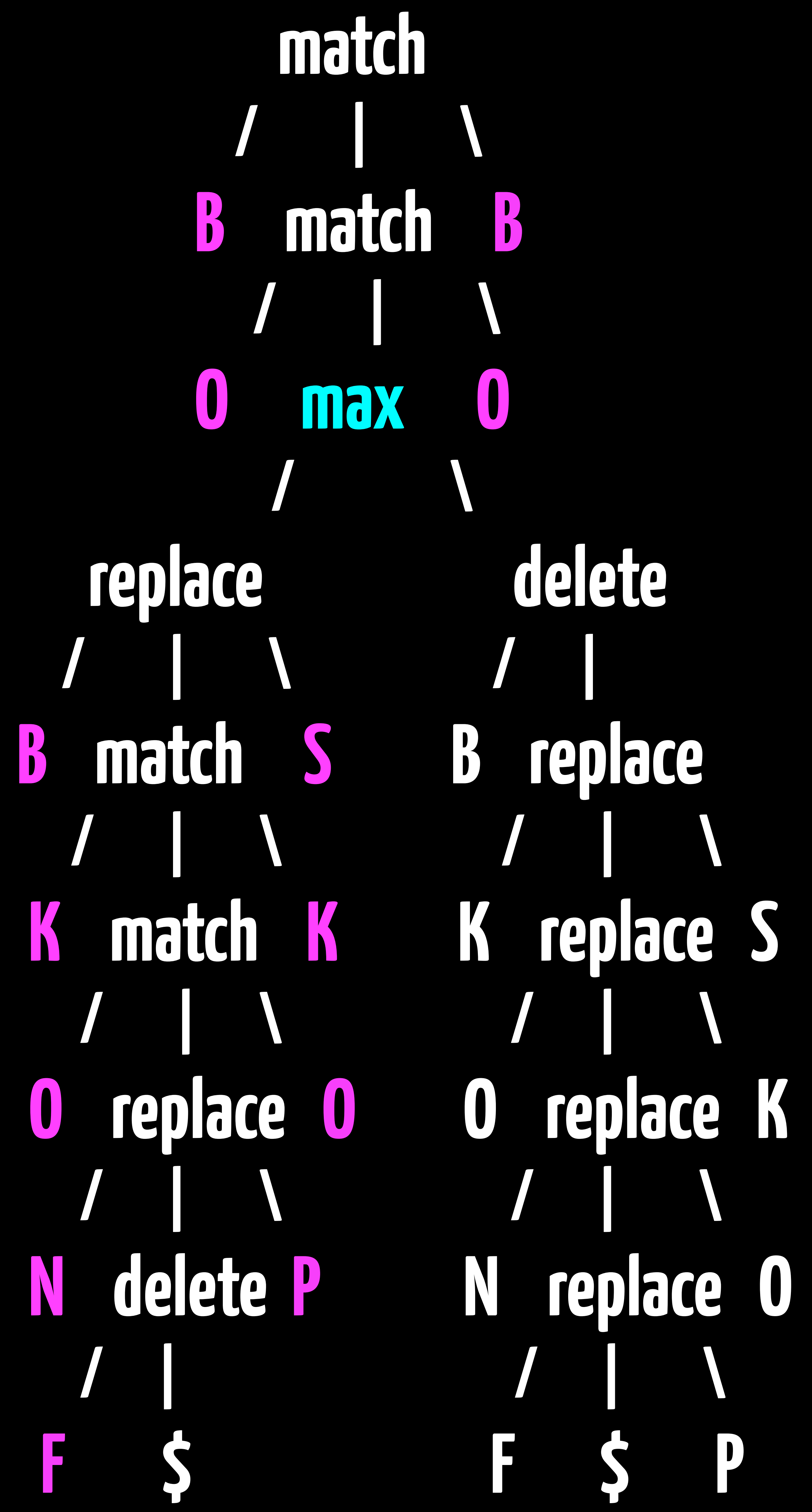

replace(l, x, r) = x-2    delete(a,x) = x-2

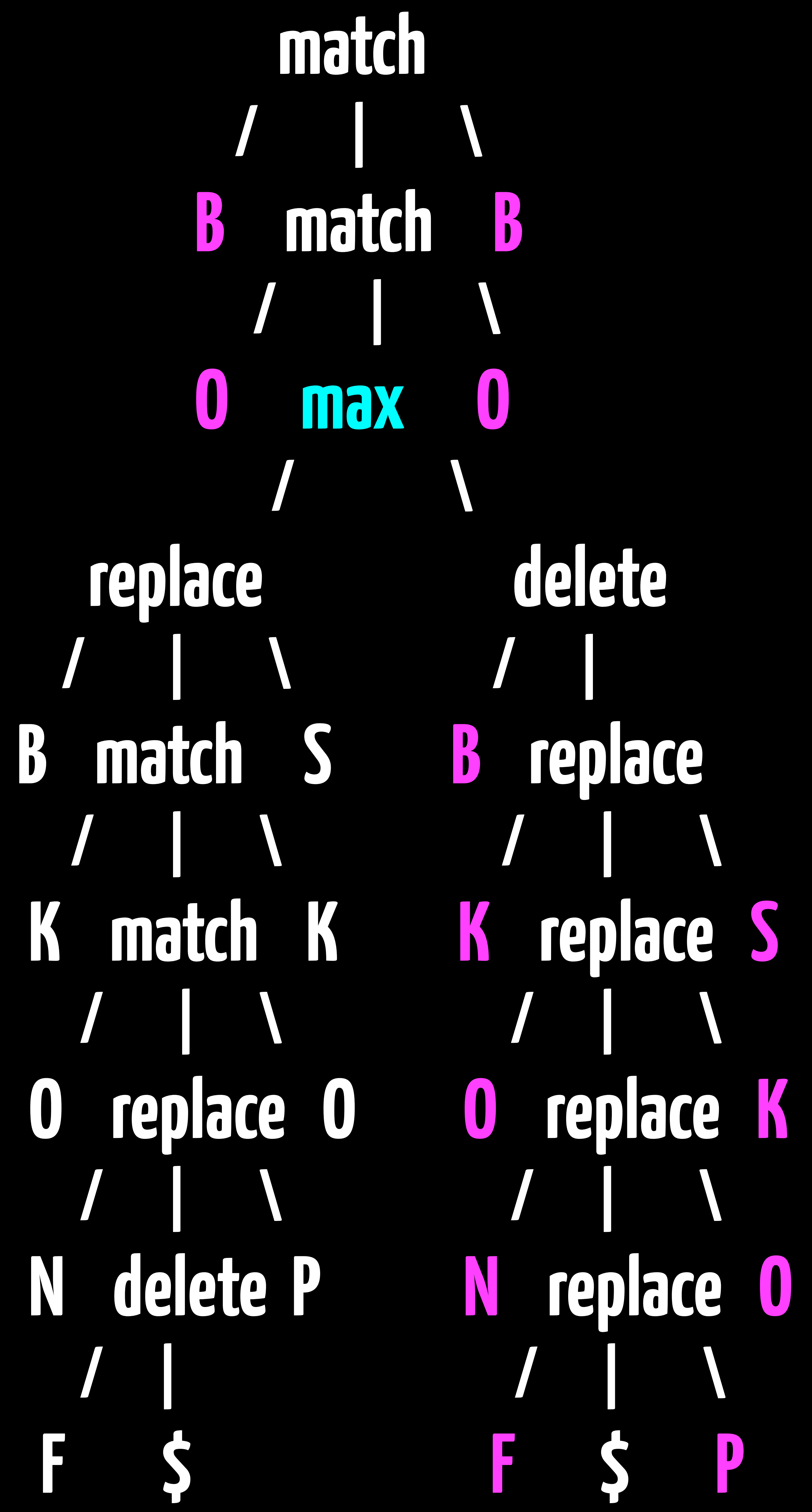



$$\text{match}(a, x, a) = x+12$$

$$\text{replace}(l, x, r) = x-2$$

$$\text{delete}(a,x) = x-2$$


match
B match B
O max O
replace
delete
B match S
B replace
K match K
K replace S
O replace O
O replace K
N delete P
N replace O
F $
F $ P



match
B match B
O max O
replace
delete
B match S
B replace
K match K
K replace S
O replace O
O replace K
N delete P
N replace O
F $
F $ P

# We found two alignments of

```
BOBKONF      BOBKONF
MMRMMRD      MMDRRRR
BOSKOP-      BO-SKOP
```

Score: 42 Score: 14

# Each alignment + score is represented by the same formulas

```
BOBKONF        BOBKONF
MMRMMRD        MMDRRRR
BOSKOP-        BO-SKOP
```

Score: 42        Score: 14

# reverse engineering summary

1) result of a DP algorithm: value of a formula

- built from evaluation functions,
- interleaved with applications of choice function

# reverse engineering summary

2) all applications of the choice function move to the top.

# reverse engineering summary

3) formulas are candidate solutions

# reverse engineering summary

4) input sequences are part of each formula

# reverse engineering - reversed :-D

# reverse engineering, reversed :-D

4. Read the input sequence
3. Construct candidate solutions (= formulas)
2. Move choice function down/inside formulas
1. Evaluate formulas to get desired result

candidates are trees

Steps
- read input,
- apply choice and
- evaluate

are always the same and can be automated.

Talents and experience go into constructing candidates:

- which candidates arise for a given input?
- what does a desired candidate look like?

⇒ a language of formulas (trees)

With this language, constructing candidates can also be automated!

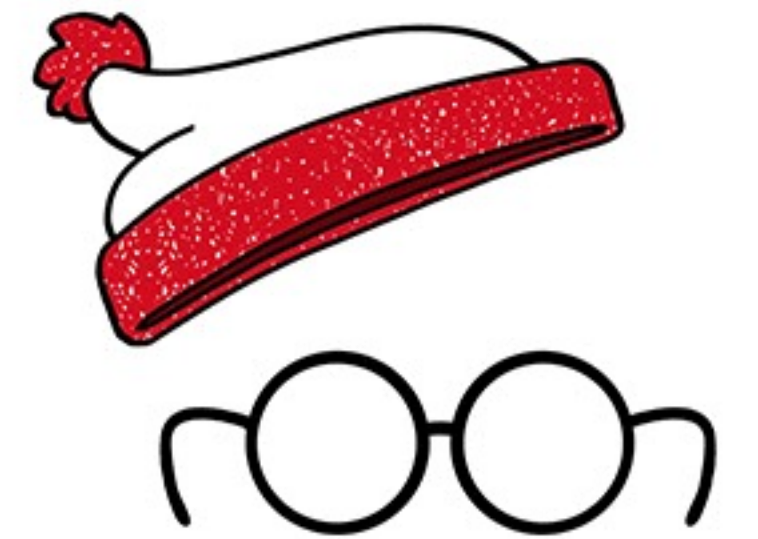
We get everything for free except for the creative part! <3

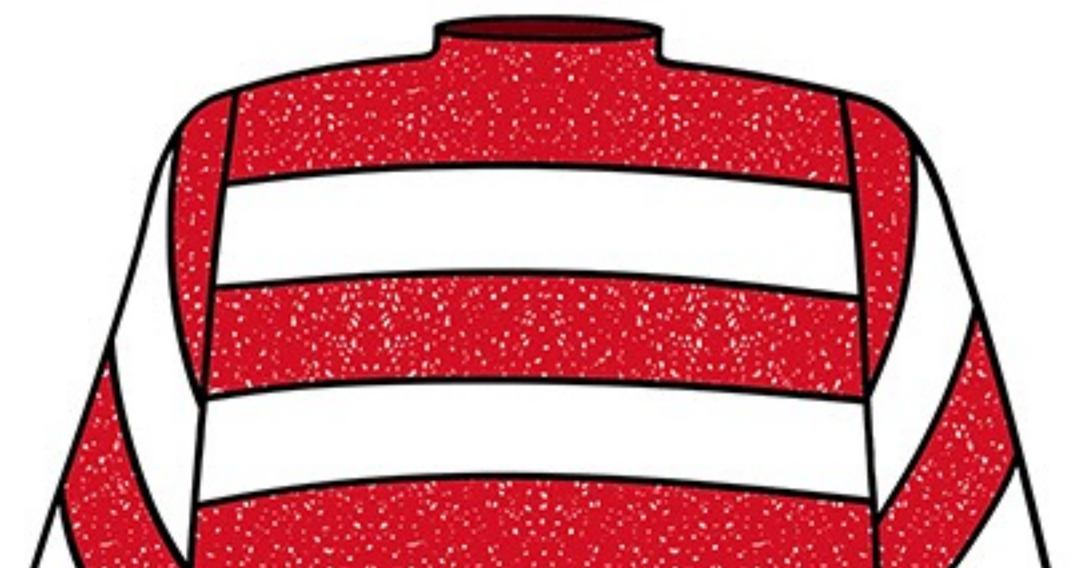
# the signature

```
1 signature Align(alphabet, answer) {
2   answer replace(<alphabet, alphabet>, answer);
3   answer delete(<alphabet, void>, answer);
4   answer insert(<void, alphabet>, answer);
5   answer empty(<void, void>);
6   choice [answer] h([answer]);
7 }
```

# the signature

signature = datatype hiding in every DP program. BUT in classical style it's invisible, since candidates are never represented.





# questions are algebras

# evaluation algebras

evaluation = scoring candidates
+ making choices

evaluation algebra = scoring functions
+ choice function

# choice functions

most popular: h = max, h = min

also popular: h = $max_k$, h = $min_k$

enumeration: h = id (keep all)

combinatorics: h = sum

sampling: h = random choice

```
h: [values] -> [values]
```

# scoring alignments: algebra score

```
1 algebra score implements
2   Align(alphabet = char, answer = int) {
3     int replace(<char a, char b>, int x) {
4     if (a == b) return x + 12; else return x - 2; }
5     int delete(<char g, void>, int x) { return x - 2; }
6     int insert(<void, char g>, int x) { return x - 2; }
7     int empty(<void, void>) { return 0; }
8     choice [int] h([int] l) { return list(maximum(l)); }
9   }
```

we evaluate

m(B, m(O, r(B, m(K, m(O, r(N, d(F, $) P) O) K) S) O) B) = 42

# scoring schemes

distance / similarity between substructures

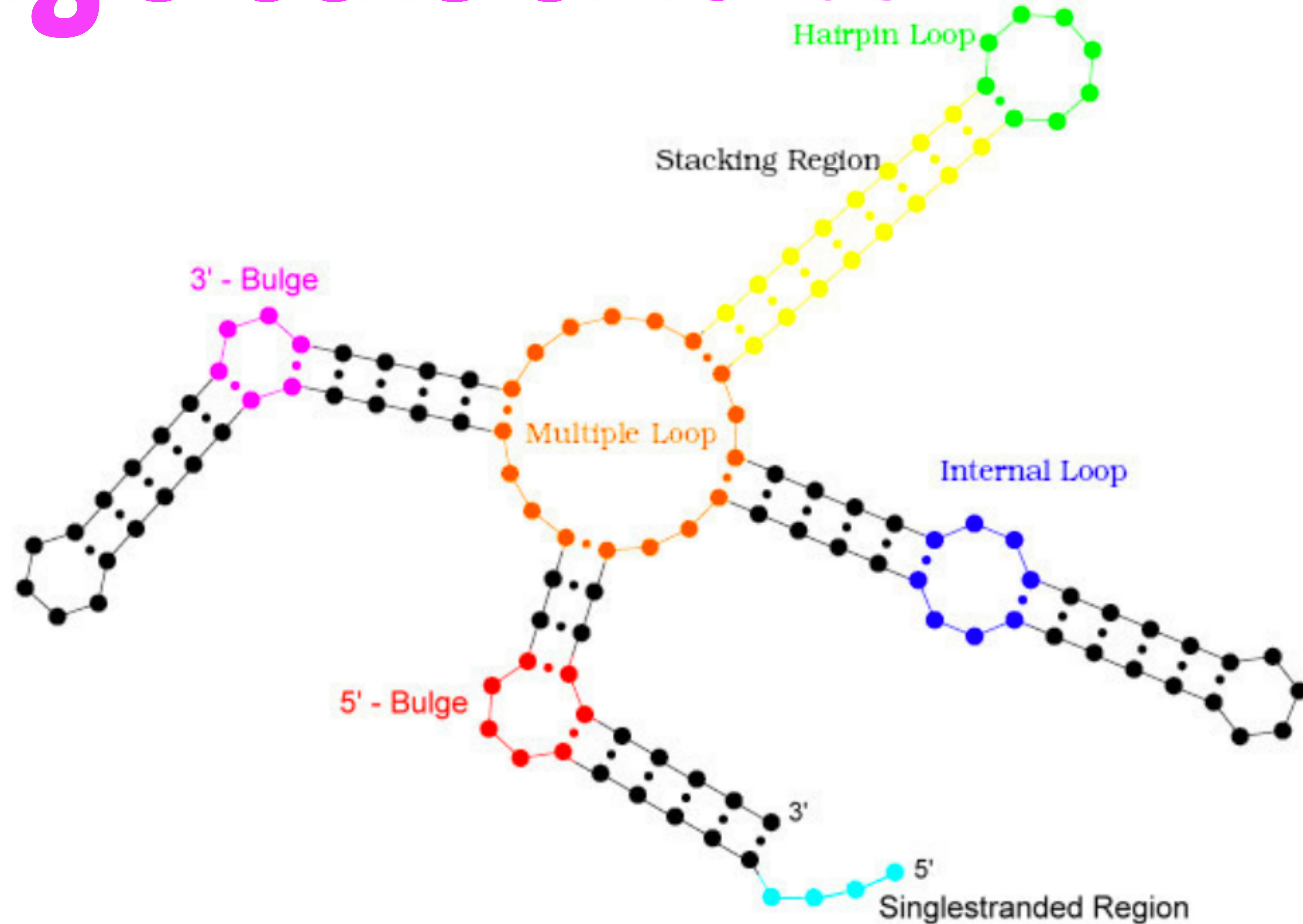
probabilities (for predicting based on probabilities)

free energy (for thermodynamic folding of molecules)

candidate representation (as strings / trees / graphics)

candidate counts

# building blocks of RNA



```
1  algebra pretty implements
2    FoldRNA(alphabet = char, answer = string) {
3      string sr(Subseq lb,string e,Subseq rb) {
4        string res;
5        append(res, '(');
6        append(res, e);
7        append(res, ')');
8        return res;
9      }
10     string hl(Subseq lb,Subseq region,Subseq rb) {
11       string res;
12       append(res, '(');
13       append(res, '.', size(region));
14       append(res, ')');
15       return res;
16     }
17     ...
18     choice [string] h([string] i) { return i; }
19 }
```

# algebra pretty

we evaluate

```
sr(C, sr(C, ml(A, sr(C, hl(C, UUUU, G), G),
                  sr(C, bl(AUA, hl(C, CCC, G)), G),
             U),
        G),
   G) =          "((((....))(...(...))))"
```

# algebra count

```
algebra count mycount auto count
```

we evaluate

sr(C, sr(C, ml(A, sr(C, hl(C, UUUU, G), G),
sr(C, bl(AUA, hl(C, CCC, G)), G),
U), G), G) = 1

# programs are grammars

# where do we stand?

we know now:

- how to represent candidates
- how to score and choose

we still need to know:

- the candidates for a given input (problem instance)

We do this using tree grammars!!

string grammar: describes a language of strings.

tree grammar: describes a language of trees (candidates).
with “input strings” as their yield sequences.

# programs are grammars

```
1 grammar alignment uses Align(axiom = ali) {
2   ali = replace(<CHAR, CHAR>, ali) |
3         delete(<CHAR, EMPTY>, ali) |
4         insert(<EMPTY, CHAR>, ali) |
5         empty(<EMPTY, EMPTY>) # h;
6 }
```

# programs are grammars

```
ali  →  replace        |   delete
        /  |  \              |  \
     CHAR  ali  CHAR       CHAR ali

         insert         |   empty
        /  |                  |
     CHAR  ali              EMPTY
```

# problem specification

Definition An Algebraic DP algorithm is specified by
- an evaluation signature $\Sigma$
- a tree grammar $G$ over $\Sigma$
- a concrete evaluation algebra $A$ with an objective function $h$

satisfying Bellman's Principle

# bellman's principle of optimality

Richard Bellman (1964):

"An optimal solution can be composed solely from optimal solutions to sub-problems."

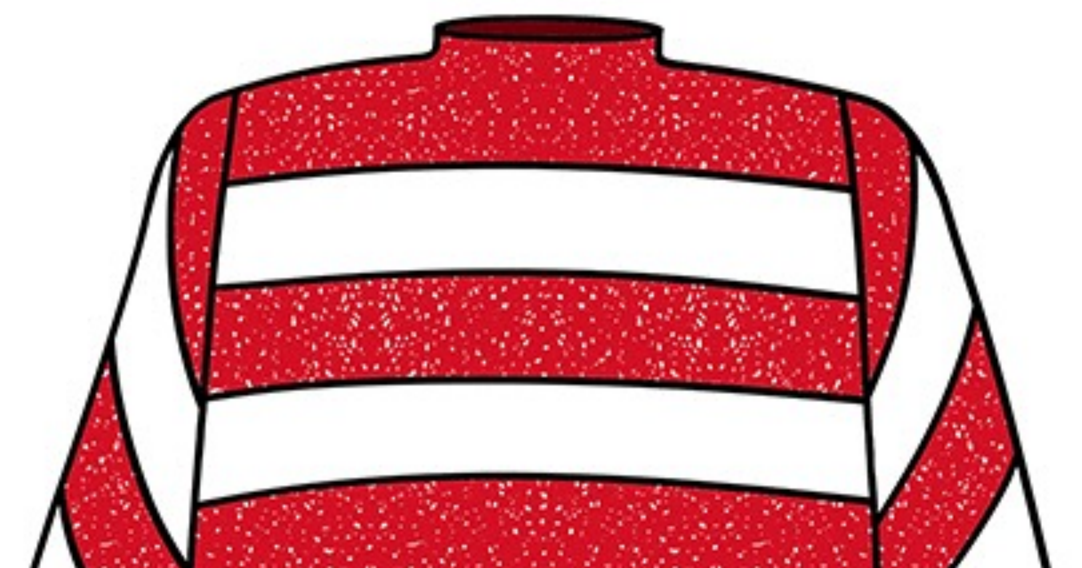
That's a requirement, not a theorem!!

# bellman's principle of optimality

Richard Bellman (1964):

"An optimal solution can be composed solely from optimal

Moving the choice function around in the formula should not affect the final result list.



# bellman's principle of optimality

Richard Bellman (1964):

"An optimal solution can be composed solely from optimal solutions to sub-problems."

That's a requirement, not a theorem!!

Proof: by proving distributivity of choice over scoring:

$$h(f(X, Y)) = h(f(h(X), h(Y)))$$

# phase amalgamation

```
rnafold(basepair,"ACAGGUUGU")⇒3
```

grammar algebra input

Conceptual view:

- Phase 1: yield parsing
- Phase 2: evaluation & choice

Reality: Both phases are merged

products are fun !!!

# where do we stand?

We can

- describe algorithms on an abstract level
- generate correct and efficient code
- independently vary tree grammar or evaluation algebra
- run one analysis at a time

# where do we stand?

....

How about doing several analyses at a time?

- find best score and print the best scoring candidate
- best RNA structure for each different shape of a molecule

# products of algebras

Product algebras $A := A_1 * A_2$

compute answer-value pairs

using functions f and h

- $f_1 * f_2$ component wise
- $h_1 * h_2$ dependent

# semantics of *

Phase 1 computes all candidates via $f_1 * f_2$

Phase 2 applies $h_1 * h_2$ once in the end

Reality: everything is interleaved! (again!)

No programming, no debugging, but proof obligation with *:

$A_1 * A_2$ must satisfy Bellman's Principle

# fun with products

- Number of co-optimal solutions basepair*count
- Easy candidate output (backtracking) basepair*pretty
- Classified DP shape*count, shape*bpmax
- Ambiguity checking pretty*count
- Sampling A | B
- Products of products...

# tools developed with ADP

| Tools | Problems solved |
| --- | --- |
| - RNAhybrid | - miRNA target prediction |
| - pknotsRG | - pseudoknot folding |
| - RNAshapes | - abstract shape analyis |
| - Locomotif | - consensus structure prediction |
| - KnotInFrame | - probabilistic shape analysis |
| - RNAsifter | - RNA motif search description and search |
| | - programmed ribosomal frame shift detection |
| | - filtering out unproductive Rfam searches |

# what’s cool about Algebraic DP?

## Advantages:

- our work is reduced to the creative apects
- we explore ideas rather than debug code
- we create re-usable and reliable components
- we turn tricks into techniques
- we make DP easier to learn

## Disadvantages:

- textbooks use old-fashioned recurrences ;)
- limited to sequence-like data, decomposition into subwords

remember reverse engineering of 42

tell me your favorite DP problem!!

sschirme@gmail.com
@linse on twitter

Thank you! <3

# resources

| | |
|---|---|
| Bellman's Gap Cafe | http://gapc.eu |
| The compiler | http://gapc.eu/compiler.html |
| Literature | http://gapc.eu/literature.html |