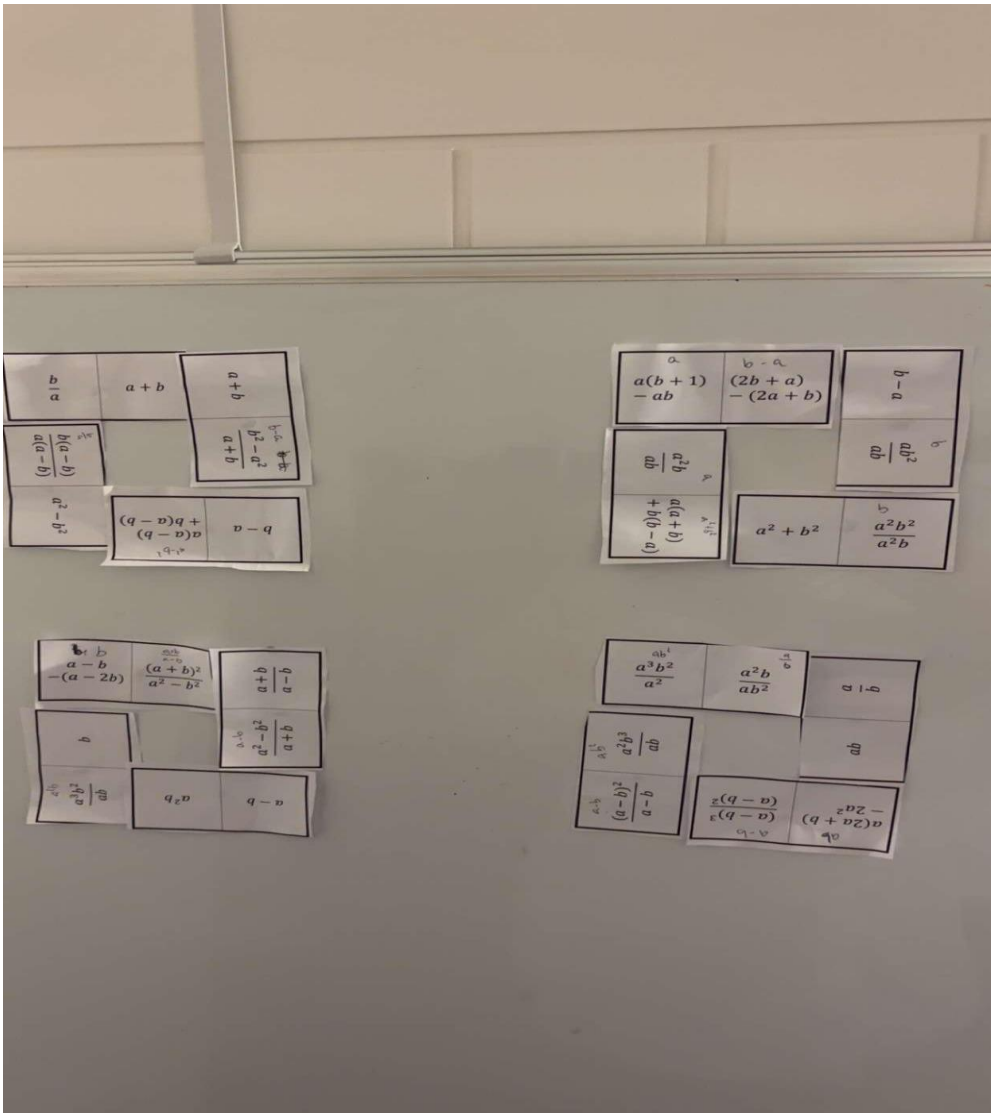
Simplifying Doughnuts – Submission by Neel and Noah

First Manual Arrangement of 4 doughnuts of 4 dominoes:



Following the graph intuition, our step-by-step thought process was:

Domino → graph edge:
A domino is two connected matching “states” (expressions). So, this can be thought of as an edge between two vertices.

Expressions → vertices:
Distinct simplified algebraic expressions are the nodes of a graph.

Doughnuts are just cycles:
A closed loop of dominoes with matching ends is a cycle in the graph.

Use all dominoes → edge partition:
We must partition all 16 edges into disjoint cycles of given lengths (4, 8, 16).

Eulerian viewpoint (for the 16-doughnut case):
A single doughnut using all edges is a Eulerian cycle, while the smaller doughnuts are the same idea just with smaller lengths.

DFS (depth-first search) for fixed-length cycles:
For each required cycle length L, we do a constrained DFS from a starting edge to find all simple cycles of length L that include that edge.

Backtracking over cycle lengths:
We recursively assign cycles of the target lengths, marking edges as used and backtracking to explore all decompositions.

Canonical forms → no duplicates:
Canonical cycles and canonical decompositions ensure we count each mathematical configuration exactly once, regardless of rotations or reversals.

Orientation → algebraic loop:
Once a cycle (as edge IDs) is known, we orient each edge so consecutive edges match, then print the algebraic chain showing each repeated matching expression

CODE:

Copy and run from the link for correct formatting/identation of code.

https://github.com/neelSchool/SimplifyingDoughnutsEnrich/blob/main/SimplifyingDoughnuts.py

```
from collections import defaultdict
import math
dominoes = [
("a-b", "(a+b)/(a-b)"),
("b-a", "b"),
("b/a", "a^2 - b^2"),
("a-b", "a^2 * b"),
("a*b", "a/b"),
("a-b", "a * b^2"),
```

```
("a^2 + b^2", "b"),
("a+b", "b-a"),
("b/a", "a+b"),
("b", "(a+b)/(a-b)"),
("a", "a^2 + b^2"),
("a^2 * b", "b"),
("a * b^2", "a/b"),
("b-a", "a^2 - b^2"),
("a*b", "a-b"),
("a", "b-a"),
]

n_edges = len(dominoes)
edge_list = list(dominoes)

adj = defaultdict(list)
for eid, (u, v) in enumerate(edge_list):
adj[u].append(eid)
adj[v].append(eid)

def other(eid, v):
u, w = edge_list[eid]
return w if v == u else u

def canonical_cycle(c):
k = len(c)
seq = list(c)
rev = list(reversed(c))
candidates = []
for i in range(k):
candidates.append(tuple(seq[i:] + seq[:i]))
candidates.append(tuple(rev[i:] + rev[:i]))
return min(candidates)

def canonical_decomposition(decomp):
canon = [canonical_cycle(c) for c in decomp]
canon.sort()
return tuple(canon)

def generate_cycles_from_edge(start_eid, L, used_global):
if start_eid in used_global:
return []
```

```
u0, v0 = edge_list[start_eid]
start_vertex = u0

used_local = {start_eid}
path = [start_eid]
cycles = []

def dfs(curr, rem):
if rem == 0:
if curr == start_vertex:
cycles.append(list(path))
return

for eid in adj[curr]:
if eid in used_global or eid in used_local:
continue
nxt = other(eid, curr)

used_local.add(eid)
path.append(eid)

dfs(nxt, rem - 1)

path.pop()
used_local.remove(eid)

dfs(v0, L - 1)
return cycles

def all_decompositions(target_lengths):
used_global = set()
current_cycles = []
solutions = []
seen = set()

target_lengths = sorted(target_lengths, reverse=True)

def backtrack(i):
if i == len(target_lengths):
if len(used_global) == n_edges:
canon = canonical_decomposition(current_cycles)
if canon not in seen:
```

```
seen.add(canon)
solutions.append([list(c) for c in current_cycles])
return

L = target_lengths[i]
remaining = n_edges - len(used_global)
if remaining < sum(target_lengths[i:]):
return

try:
start_eid = min(e for e in range(n_edges) if e not in used_global)
except ValueError:
return

possible = generate_cycles_from_edge(start_eid, L, used_global)
local_seen = set()

for cyc in possible:
canon_c = canonical_cycle(cyc)
if canon_c in local_seen:
continue
local_seen.add(canon_c)

for e in cyc:
used_global.add(e)
current_cycles.append(cyc)

backtrack(i + 1)

current_cycles.pop()
for e in cyc:
used_global.remove(e)

backtrack(0)
return solutions

def orient_cycle(cycle):
oriented = []
first = cycle[0]
u, v = edge_list[first]
oriented.append((first, u, v))
prev = v
```

```
for eid in cycle[1:]:
x, y = edge_list[eid]
if x == prev:
oriented.append((eid, x, y))
prev = y
elif y == prev:
oriented.append((eid, y, x))
prev = x
else:
oriented.append((eid, x, y))
prev = y
return oriented

def print_cycle_list(cycle):
oriented = orient_cycle(cycle)
print(" Cards:")
for eid, u, v in oriented:
print(f" [{eid}] {u} → {v}")
chain = []
for i, (eid, u, v) in enumerate(oriented):
if i == 0:
chain.append(u)
chain.append(v)
print("\n Algebraic loop:")
print(" " + " → ".join(chain))
print()

def print_decomposition(decomp, index):
print(f"Decomposition {index}:")
for i, cyc in enumerate(decomp, 1):
print(f"Doughnut {i} (length {len(cyc)}):")
print_cycle_list(cyc)

if __name__ == "__main__":
print("4 doughnuts of 4")
sols = all_decompositions([4,4,4,4])
print(f"Total decompositions: {len(sols)}")
for idx, d in enumerate(sols, 1):
print_decomposition(d, idx)

print("2 doughnuts of 8")
sols = all_decompositions([8,8])
print(f"Total decompositions: {len(sols)}")
```

```
for idx, d in enumerate(sols, 1):
print_decomposition(d, idx)

print("1 doughnut of 16")
sols = all_decompositions([16])
print(f"Total decompositions: {len(sols)}")
for idx, d in enumerate(sols, 1):
print_decomposition(d, idx)
```