```python
# --------------------- Step 10: Build array5 ---------------------
# Every remaining unmapped atom.
array5 = [atom[0] for atom in section if final[atom[0]] == ""]
pairs5 = []
if len(array5) >= 2:
    for i in range(0, len(array5) - 1, 2):
        pairs5.append((array5[i], array5[i+1]))
for pair in pairs5:
    rstr = generate_random_string()
    atom1 = next(a for a in section if a[0] == pair[0])
    atom2 = next(a for a in section if a[0] == pair[1])
    if atom1[1].lower() == 'c' and atom2[1].lower() == 'c':
        final[pair[0]] = "TC5" + rstr
        final[pair[1]] = "TC5" + rstr
    elif 'n' in (atom1[1].lower(), atom2[1].lower()):
        final[pair[0]] = "TN6a" + rstr
        final[pair[1]] = "TN6a" + rstr
```

for this step we need to change the followings: currently, it is just taking every 2 atoms and map them as TC5 or TC6a without accounting for connections:

we need to change the formula

if len(array5) == 6:

we pair them by connectivity by taking any node and find its neighbor as a pair then take the neighbor of the neighbor as the next node and pair it with its neighbor and the last 2 should be together -> 3 pairs in total

elif len(array5) == 4:

we pair them by connectivity by taking a node that has a neighbor that is already mapped and pair it with the neighbor that has not been mapped and the last 2 should be together -> 2 pairs in total

else (this is the case for if it is 2):

check if these 2 nodes are connected. If it is make them a pair, if it is not check to see if there is pairs in pairs0 (from step 2) . if there is, remove the final of both of those to empty again, so now we have 4 unampped atoms, treat it as len(array5) == 4 case

After all the pairings, we do the rest just like how we are doing it (mapping it to TC5 or TN6a)

Step 6 and Step 8 of the benzene ring currently is not accounting for the case where 2 ring section fight for 1 non-ring atom -> add an extra argument to fix this issue

Benzene ring Step 1 is having issues:

```
# ---------------------- Step 1: Build array0 ----------------------

array0 = []

for atom in section:

idx = atom[0]

if atom[1].lower() == 'c' and final[idx] == "" and len(atom[3]) != 0:

valid_foreign = any(full_mapping[tup[0]][tup[1]][2] != 0 for tup in atom[3])

if valid_foreign:

array0.append(idx)

# array0 should have length 2, 4, or 6.
```

we need to fix this array[0] as follows:
for each atom in sections, we will check exactly that but we also need to check if any of its inner neighbor is connected to the exact ring section
so

```
array0 = []

for atom in section:

check if the atom is already in array0
```

```
idx = atom[0]

if atom[1].lower() == 'c' and final[idx] == "" and len(atom[3]) != 0:

valid_foreign = any(full_mapping[tup[0]][tup[1]][2] != 0 for tup in atom[3])

if valid_foreign:
```

check if any of its inner neighbor is connecting to this same section

if there is, add both to this array0

```
# array0 should have length 2, 4, or 6.
```

now that this is fix, we should fix the same for the non-benzene 5-ring section
the non-benzene 5-ring section is fixed, but we now need to take care of some edge cases:

```
    # -----------------------

    # (D) Final check: Ensure every atom in the section is mapped.

    for atom in section:

        if final[atom[0]] == "":

            raise ValueError("Non-benzene 5-ring section not fully mappable!")

    return final
```

this is not the end of the world, we can fix this still!

```
if final[atom[0]] == "":
```

we will now look for the final of each neighbor (yes, even outer neighbors),

if any final start with TC5, find the other index in final with the exact string with the neighbor with TC5, remember both indices and emptied the final on both

then call this function:

what we want to do is merge the 2 TC5 atoms with this atom we will call this function to detemine the right key based on the atom type of the main atom we are working with

def pick_bead_key(martini_dict: Dict[str, List[Any]], element: str, bond_order: Optional[int] = None, kind: str = 'S') -> Optional[str]: """ Pick a bead-type key from martini_dict whose • key starts with kind ('S' or 'T'), • val[0] == 2, • and whose val[2] (the "(...)" part) matches our element + optional '='.

```
element: e.g. 'O', 'N', 'S', 'CL', 'I', 'C', 'BR'
bond_order: if element == 'O' and bond_order == 2, we look for '(=O)'
else '(O)'
kind: 'S' for SN6*, 'SC6'…, 'SN6a', etc.; 'T' for TN6*, 'TC4'…, 'TX2',
etc.
"""
# build the pattern to look for inside the parentheses
el = element.capitalize()  # so 'CL' -> 'Cl'
if element == 'O' and bond_order == 2:
    pat = '(=O)'
else:
    pat = f'({el})'

for key, val in martini_dict.items():
    if not key.startswith(kind):
        continue
    if val[0] != 2:
        continue
    # val[2] is something like 'CC(O)' or 'C(Cl)' or 'CC(=O)'
    if val[2].endswith(pat):
        return key
return None


We use S here
```

```
then assign the previous TC5s with this new atom as the new key +
random string
```

```
if we can't find any start with TC5, we check if the atom we are left
with is .upper() = C, if it is not, now we throw the error, if it is,
we find any final[any neighbor] that start with T,
```

then find all index with matching final strings with the neighbor string in the final array

then we will change the first letter from T to S and apply it to the atom as well as its neighbor and the matching indices.

if we can’t find any that start with T, we can find any that start with S,

then find all index with matching final strings with the neighbor string in the final array

then we will remove the S and apply it to the atom as well as its neighbor and the matching indices.

if we still can’t find any with T or S, we throw the same error here (non-benzene 5-ring not mappable)

OK now that that is fixed, let’s move on to an even bigger issue: Non-ring section with only 1 edge

First, since this section is super important, let’s print some statements

1. Any breaking should print what the sections after breaking looks like
2. what is the size of the section(if not 1, we are doing something wrong)

Ok so the section we care about is:

--- Main body of TASK 3.e ---

```
# Build array0: list of global indices for edge atoms.
edge_globals = [atom[0] for atom in section if atom[5]]
num_edges = len(edge_globals)
if num_edges not in (2, 3, 4):
    raise ValueError("Non-ring section 1-bead mapping requires 2, 3,
or 4 edge atoms; found: " + str(num_edges))
```

fix, as a debug, let’s print all the atoms in this section if the num_edge is not in (2,3,4), so, if num_edges not in (2, 3, 4):

for atom in section add atom type to array0

print array0, then raise value error

```
    raise ValueError("Non-ring section 1-bead mapping requires 2, 3, 
or 4 edge atoms; found: " + str(num_edges))
```

--- Main body of TASK 3.e ---

```
     # Build array0: list of global indices for edge atoms.
edge_globals = [atom[0] for atom in section if atom[5]]
num_edges = len(edge_globals)
if num_edges not in (2, 3, 4):
    # debug: print all atom types in this section
    debug_types = [atom[1] for atom in section]
    print("DEBUG section atom types:", debug_types)
    print("DEBUG edge_globals (indices):", edge_globals)
    raise ValueError("Non-ring section 1-bead mapping requires 2, 3, 
or 4 edge atoms; found: " + str(num_edges))("Non-ring section 1-bead 
mapping requires 2, 3, or 4 edge atoms; found: " + str(num_edges))
```

ok debugging is working, let me see a lot of things now. So to fix this

we can stop printing. We will check if the size of this section is 1. we will proceed this whole process just like for the non-benzene 5-ring section:

we will now look for the final of each outer neighbor

if any final start with TC5 or TC3, find the other index in final with the exact string with the neighbor with TC5 or TC3, remember both indices and emptied the final on both then call this function:

what we want to do is merge the 2 TC5 or TC3 atoms with this atom

we will call this function to detemine the right key based on the atom type of the main atom we are working with

def pick_bead_key(martini_dict: Dict[str, List[Any]], element: str, bond_order: Optional[int] = None, kind: str = 'S') -> Optional[str]: """ Pick a bead-type key from martini_dict whose • key starts with kind ('S' or 'T'), • val[0] == 2, • and whose val[2] (the "(...)" part) matches our element + optional '='.

```
element: e.g. 'O', 'N', 'S', 'CL', 'I', 'C', 'BR'
bond_order: if element == 'O' and bond_order == 2, we look for '(=O)'
else '(O)'
kind: 'S' for SN6*, 'SC6'…, 'SN6a', etc.; 'T' for TN6*, 'TC4'…, 'TX2',
etc.
"""
# build the pattern to look for inside the parentheses
el = element.capitalize()  # so 'CL' -> 'Cl'
if element == 'O' and bond_order == 2:
    pat = '(=O)'
else:
    pat = f'({el})'

for key, val in martini_dict.items():
    if not key.startswith(kind):
        continue
    if val[0] != 2:
        continue
    # val[2] is something like 'CC(O)' or 'C(Cl)' or 'CC(=O)'
    if val[2].endswith(pat):
        return key
return None


We use S here

then assign the previous TC5s or TC3s with this new atom as the new
key + random string
```

if we can’t find any start with TC5 or TC3, we check if the atom we are left with is .upper() = C, if it is not, now we throw the error, if it is, we find any final[any neighbor] that start with T,

then find all index with matching final strings with the neighbor string in the final array

then we will change the first letter from T to S and apply it to the atom as well as its neighbor and the matching indices.

if we can’t find any that start with T, we can find any that start with S,

then find all index with matching final strings with the neighbor string in the final array

then we will remove the S and apply it to the atom as well as its neighbor and the matching indices.

if we still can’t find any with T or S, we throw the same error here (non-ring has 1 edge that is not mappable)

OK now that that is “somewhat” fixed, we need to focus on the Cancer cases with .

from rdkit import Chem from setup import get_atom_properties, connectivity_matrix from martini_3_dictionary import get_m3_dict from mapping_scheme import map_molecule, parse_smiles from algorithm import map_martini_beads from text_file import create_beads_text_file import copy import sys

def main(): # Input SMILES string compound_name = input() smiles = input() mol = Chem.MolFromSmiles(smiles) atom_properties = get_atom_properties(mol) prop_copy = copy.deepcopy(atom_properties) final = ["" for _ in atom_properties] conn_mat = connectivity_matrix(mol, len(atom_properties)) bead_dict = get_m3_dict() smiles_token = parse_smiles(smiles) mapping = map_molecule(smiles_token, conn_mat, atom_properties) mapping_copy = copy.deepcopy(mapping) final = map_martini_beads(mapping, final, bead_dict) create_beads_text_file(final, mapping_copy, compound_name, smiles)

if **name** == "**main**": try: main() except Exception as e: # only print a one-line error, no traceback print(f"Error: {e}") sys.exit(1)

so after inputting SMILES, if there is a “.” somewhere in the string, we will need to split it by ., ex. CC.CC becomes two strings, CC and CC

then we sort the array of string based on descending length

then for each new string has size > 4, we run the algorithm for that part and save it as “compund_name” + “_cut{n}.txt” where n increament from 1

note: only when all splits with string size > 4 work we will do this, otherwise we will just put that whole compound into the non-working list.

OMG I FOUND A HUGE ISSUE:

CC1COC2(CC1OC(=O)c1ccccc1)OC1CC(C(=O)OC3C(O)C(O)C(O)C(O)C3OC3OC(CO)C(O)C(O)C3O)CC(O)C1(O)C2O
look at this SMILES code, there are 6 instances of number 1!!

this means that some with the current mapping, we can’t even find the rings.

What to fix, let’s look around

nothing to fix so that’s add another function that process this case call it duplicate ring number handler:

goal: make sure all ring numbers are unique

read over the string

find 1, then find the closest 1 to it that does not have a % before it and keep track of its location, when that later 1 is encountered, ignore it

find 2, then 3, then 4, (that does not have a % before it)... stop when a number is not found

if we find from 1 to 9, keep going with 10, this is different before if we ever encounter 10 it must have a % before it. So

starting from 2 digits we will find %10, %11, ... and its pairs and keep track of its location until we can't find it no more.

if we stop before 10 we dont need to keep going with 10

after this step, keep track of the highest found number

now scan through the string again, whenever a number is found that we did not keep track (ex. a 3rd "1" without a % or a 3rd "2" without a % and so on), we will find its pair, (so if there exists a 3rd "1", there should be a 4th and if there is a 5th there should be a 6th and so on). then we will change this pair to a number 1 higher than the highest found number (remember to add % if it goes past 10), then we keep track of this new number and its pair so we know if we encounter them again we dont care.

do this until no numbers is unaccounted for and return the new smiles string.

ANOTEHR BIG ISSUE:

TASK 1a: Parse the SMILES string into tokens.

def parse_smiles(smiles: str) -> List[str]: """ Tokenize the SMILES string while ignoring characters we can get from connectivity. Recognizes multi-character tokens like "Cl", "Br", "NH" and ignores parentheses and bond symbols. Example: "c1c(CC)cccc1" -> ['c', '1', 'c', 'C', 'C', 'c', 'c', 'c', 'c', '1'] """ tokens = [] i = 0 while i

< len(smiles): ch = smiles[i] # Skip ignored characters if ch in "()=#$@H[]+-%/\": i += 1 continue # Check for multi-letter tokens: if ch == 'C' and i + 1 < len(smiles) and smiles[i+1] == 'l': tokens.append("Cl") i += 2 continue if ch == 'B' and i + 1 < len(smiles) and smiles[i+1] == 'r': tokens.append("Br") i += 2 continue if ch == 'N' and i + 1 < len(smiles) and smiles[i+1] == 'H': tokens.append("NH") i += 2 continue # If digit, add it as a token. if ch.isdigit(): if i + 1 < len(smiles) and smiles[i+1].isdigit(): tokens.append(ch + smiles[i+1]) i += 2 continue else: tokens.append(ch) i += 1 continue # Otherwise, assume it is an element symbol (like C, c, O, etc.) tokens.append(ch) i += 1 return tokens

this code currently merge numbers together if they are next to each other. WHICH MEANS, for cases for the smiles is like c1ccc2ccccc12

12 will be read as 12

FIX:

we need to fix how % work and we dont want to delete % but use them as helper for the numbers

```
        # If digit, add it as a token.

        if ch.isdigit():

            if i + 1 < len(smiles) and smiles[i+1].isdigit():

                tokens.append(ch + smiles[i+1])

                i += 2

                continue

            else:

                tokens.append(ch)

                i += 1

                continue
```

fix this to, if digit is found, add it to the token immediately and do i+=1 only, but if % is found, then we check if the next 2 characters are numbers. If they are, we add both as 1 token (ex. %11 token would be 11) then we do i+=3

next fix:

CASE 1: 2 edge atoms.

```
if num_edges == 2:
    start_global = edge_globals[0]
    start_local = next(i for i, atom in enumerate(section) if atom[0]
== start_global)
    path_str = trace_linear_path(section, start_local).upper()

    # 1) Try section 7
    candidate_keys = [
        key for key, val in martini_dict.items()
        if val[0] == 7 and val[1] == 0
           and (val[2].upper() == path_str or val[2].upper() ==
path_str[::-1])
    ]
    candidate_keys = list(set(candidate_keys))

    # 2) Fallback to section 11 if needed
    if not candidate_keys:
        # ---- new: special split logic for 4-atom sections ----
        if len(section) == 4:
            # find its only inner-connecting neighbor
            nbr_local, bond_order = section[start_local][4][0]
            t1 = section[start_local][1]
            t2 = section[nbr_local][1]
            # choose symbol: '=' double, '#' triple, '-' single
            sym = "=" if bond_order == 2 else "#" if bond_order == 3
else ""
            split_str = f"{t1}{sym}{t2}".upper()
            # try section 7 first
            candidate_keys = [
                k for k, v in martini_dict.items()
                if v[0] == 7 and v[1] == 0
                   and (v[2].upper() == split_str or v[2].upper() ==
split_str[::-1])
            ]
            candidate_keys = list(set(candidate_keys))

            # fallback to section 11 if none
            if not candidate_keys:
                candidate_keys = [
                    k for k, v in martini_dict.items()
                    if v[0] == 11 and v[1] == 0
                       and (v[2].upper() == split_str or v[2].upper()
== split_str[::-1])
                ]
                candidate_keys = list(set(candidate_keys))
                if candidate_keys:
                    warnings.warn(
                        f"Warning: falling back to section 11 for 2-
edge split mapping (split: {split_str})",
                        UserWarning
                    )
        else:
            candidate_keys = [
                key for key, val in martini_dict.items()
                if val[0] == 11 and val[1] == 0
                   and (val[2].upper() == path_str or val[2].upper()
== path_str[::-1])
            ]
            candidate_keys = list(set(candidate_keys))
            if candidate_keys:
                warnings.warn(
                    f"Warning: falling back to section 11 for 2-edge
mapping (path: {path_str})",
                    UserWarning
                )
```

```
    # 3) Commit or error
    if len(candidate_keys) == 1:
        rstr = generate_random_string()
        bead = candidate_keys[0] + rstr
        for atom in section:
            final[atom[0]] = bead
        return final
```

this case need fixing

especially on the case that if len(section) == 4:
this case is not we cut them in half and only consider 2 atoms, in this case, we cut them in half and we treat them each as a complete different sections!!

so how this work

for this part:

```
# find its only inner-connecting neighbor
            nbr_local, bond_order = section[start_local][4][0]
            t1 = section[start_local][1]
            t2 = section[nbr_local][1]
            # choose symbol: '=' double, '#' triple, '-' single
            sym = "=" if bond_order == 2 else "#" if bond_order == 3
else ""
            split_str = f"{t1}{sym}{t2}".upper()
```

we should have 2 sections each with a pair, then we apply the rules for section 7 and section 11 for each section separately, even the CO/OC case and we should have 2 different keys each contains 2 atoms only

Next section fix:

```
    # CASE 4: 5 or 6 edge atoms.

    elif num_edges > 4:
```

if the amount of edge atoms exceed 4, we need to break it down into 2 sections. in both of these cases, there will always be 2 centers find both center.

for each center, trace out the path that is not the other center and make that a section, then, run each section again in def map_non_ring_section_1bead(section: List[List[Any]], final: List[str], martini_dict: Dict[str, List[Any]], full_mapping: List[List[List[Any]]]) -> List[str]:

then return final

```
[

[[7, 'C', 1, [(1, 6, np.float64(1.0))], [(1, np.float64(1.0)), (4,
np.float64(1.0))], False],

[8, 'O', 1, [], [(0, np.float64(1.0)), (2, np.float64(1.0))], False],
[9, 'C', 1, [(2, 0, np.float64(2.0))], [(1, np.float64(1.0)), (3,
np.float64(1.0))], False],

[11, 'C', 1, [], [(2, np.float64(1.0)), (4, np.float64(1.0))], False],
[12, 'C', 1, [(3, 0, np.float64(1.0)), (4, 0, np.float64(1.0))], [(0,
np.float64(1.0)), (3, np.float64(1.0))], False]],

[[0, 'C', 0, [], [(1, np.float64(1.0))], True], [1, 'C', 0, [], [(0,
np.float64(1.0)), (2, np.float64(1.0)), (3, np.float64(1.0))], False],
[2, 'C', 0, [], [(1, np.float64(1.0))], True], [3, 'O', 0, [], [(1,
np.float64(1.0))], True]],



[[4, 'C', 0, [], [(5, np.float64(1.0)), (6, np.float64(1.0))], False],
[5, 'O', 0, [], [(4, np.float64(1.0))], True], [6, 'C', 0, [(0, 0,
np.float64(1.0))], [(4, np.float64(1.0))], True]],
[[10, 'O', 0, [(0, 2, np.float64(2.0))], [], True]], [[13, 'C', 0,
[(0, 4, np.float64(1.0))], [], True]], [[14, 'O', 0, [(0, 4,
np.float64(1.0))], [], True]]]
```