TASK 3: Create a tree-like algorithm to map martini beads to the molecule

I. Let's look of our possible inputs, examples, how they are made, and what it means (for synchronization purpose, we will use inputs that all represent 1 molecule: a single benzene ring):

1. Smiles string:
    a. Example: c1ccccc1
    b. Made from user input
    c. it is the smiles representation of the molecules
2. Smiles token list:
    a. Example: ['c', '1', 'c', 'c', 'c', 'c', 'c', '1']
    b. Made from parsing the smiles string
    c. This is important because we can ignore a lot of unnecessary symbols as well as merging the important characters as 1 token
        i. we will ignore characters ()=#$@H[]+-%
        ii. we will merge Cl, Br, NH, and digits together(if there are 10 rings or more)
3. connectivity matrix:
    a. Example:

	0	1	2	3	4	5
0	0	1.5	0	0	0	1.5
1	1.5	0	1.5	0	0	0
2	0	1.5	0	1.5	0	0
3	0	0	1.5	0	1.5	0
4	0	0	0	1.5	0	1.5
5	1.5	0	0	0	1.5	0

    [[0. 1.5 0. 0. 0. 1.5]
    [1.5 0. 1.5 0. 0. 0.]
    [0. 1.5 0. 1.5 0. 0.]
    [0. 0. 1.5 0. 1.5 0.]
    [0. 0. 0. 1.5 0. 1.5]
    [1.5 0. 0. 0. 1.5 0.]]
    b. Made from turning the Smiles string to mol and analyze the bond order
    c. Type of bonds: 0 for no bond, 1 for singular bond, 2 for double bond, 3 for triple bond, and 1.5 for benzene bond
4. bead dictionary:
    a. martini_dict["TC5"] = [1, 0, "CC", 2, ""]

b. Made manually from analyzing the building block table of the Martini 3 mapping examples
c. key is represented by the bead name, if there are duplicates, uses +, ++, ... so that we can add more than one way the same bead can mean
d. Each key is mapped to a value that is a list of length 5, the form is [section, sidechains, representation, continuation, preferred neighbors]
    i. section: a total of 8 sections: with value ranging from 1 to 8. which the values denote that the bead is: 1 composes of the atoms inside a benzene ring. 2 composes of at least 1 atom inside a benzene ring and 1 atom outside. 3 and 4 are like 1 and 2 but of non-benzene hex-ring. 5 and 6 are like 1 and 2 but of penta-ring. 7 composes of all non-ring atoms. 8 composes of the atoms that are parts of 2 rings or more.
    ii. sidechains: 0 means the bead can be traced linearly. 1 means the opposite
    iii. representation: The atom representation of this node, including the atom type, the amount of bonds, and (...) for if there are branching paths
    iv. continuation: how many atoms outside of this bead is connecting to this bead
    v. preferred neighbors: the bead that is likely to be a neighbor of this bead

5. mapping (of the molecules):
    a. Examples:

```
[[[0, 'c', 2, [], [(1, np.float64(1.5)), (5, np.float64(1.5))], False],
[1, 'c', 2, [], [(0, np.float64(1.5)), (2, np.float64(1.5))], False],
[2, 'c', 2, [], [(1, np.float64(1.5)), (3, np.float64(1.5))], False],
[3, 'c', 2, [], [(2, np.float64(1.5)), (4, np.float64(1.5))], False],
[4, 'c', 2, [], [(3, np.float64(1.5)), (5, np.float64(1.5))], False],
[5, 'c', 2, [], [(0, np.float64(1.5)), (4, np.float64(1.5))], False]]]
```

    b. Made from analyzing the smiles token, connectivity matrix, and a properties matrix that we discussed in previous tasks
    c. This is an array of arrays of arrays of 6 different values
    d. The most outer array is just the whole molecule
    e. Each inner array is a “section”, where a section is classified as any ring or any connected non-ring atoms. Here since we have only benzene, we will only have 1 section which is the whole benzene ring
    f. Each inner-inner array represents an atom and everything about it. The next part will explain what is in this array:

i. index 0: This is an integer that represents the appearance of this atom when tracing the smiles tokens, where 0 means it is the first atom encountered when tracing the token
ii. index 1: represent the atom type, we get this from tracing the token
iii. index 2: represent the section type, 0 means non-ring, 1 means non-benzene ring, and 2 means a benzene ring
iv. index 3: an array representing the connection from this atom to another atom that is not found in this section. The form of this array is [section index, atom index (this index is not the same as index 0 of the atom, this is the index location of the section array where we can find this atom), bond amount]
v. index 4: an array representing the connection from this atom to another atom that is found in this section. The form of this array is [(atom index (this index is not the same as index 0 of the atom, this is the index location of the section array where we can find this atom), bond amount), (more tuples if needed]
vi. index 5: is this an edge atom, an edge atom is only connected to exactly 0 or 1 other atom within its group (foreign connection does not matter)

6. final:
    a. example: ['', '', '', '', '', '']
    b. Made from this line of code

    ```
    atom_properties = get_atom_properties(mol)
    final = ["" for _ in atom_properties]
    ```

    c. Each index represents an atom, and this index is the index that is in index 0 of mapping the molecules section. We preempt them as empty and update them to the correct bead as we go.
7. As we work on the algorithm, we can see which inputs are needed. For now, these are all the inputs we can use, we have them already from previous tasks, and we do not need to build an algorithm for them

II. Algorithm planner:

1. Plan: we will build a tree-like algorithm planner where we will use the value from the dictionary, go through each index, which in this case serves as arguments, and narrow it down to exactly 1 bead that matches the atom group. Once we have this bead, we will save this result into 'final' so that we know the atom is already accounted for.
2. Actual pseudocode for this algorithm. This will be the START of TASK 3:

We will try to map 1 section at a time. We will try to map the benzene ring related section first, then non-benzene ring related section, then non-ring section last.

Section 1: (Mapping of benzene ring and benzene ring related atoms only)

loop through the final array, check if any index is “”. If it’s found, find the section of this index by inspecting mapping in the atom section and find where index 0 = index. Check if index 2 of that atom = 2, if not, continue the loop. if it is, we have found the section that is a benzene ring that has not been mapped.
REFER TO TASK 3.a HERE
OK, after task 3.a, That benzene ring should be mapped, and after the loop, we will know that ALL benzene ring should be mapped.

Section 2: (mapping of non-benzene ring, 5 or 6 long):
loop through the final array, check if any index is “”. If it’s found, find the section of this index by inspecting mapping in the atom section and find where index 0 = index. Check if index 2 of that atom = 1, if not, continue the loop. if it is, we have found the section that is not a benzene ring but is a ringthat has not been mapped.
we will check the length of this section:
if length = 6: REFER TO TASK 3.b HERE
if length = 5: REFER TO TASK 3.c HERE

After task 3.b/c. we know that non-benzene ring section will be mapped and after the loop, we will know that all rings should be mapped

Section 3: (mapping of non-ring structures):
loop through the final array, check if any index is “”. If it’s found, find the section of this index by inspecting mapping in the atom section and find where index 0 = index. Check if index 2 of that atom = 0, if not, throw an error saying there are unmapped ring leftover. if it is, we have found the section that is not a ring that has not been mapped.

Unlike the other 2 sections, we need to classify this section into 2 groups: groups that can be represented as 1 single bead, and the group that can’t.

REFER TO TASK 3.d HERE

now that we know if the node is 1-2 bead mappable, we can break this into 2 parts:

1. If the return from task 3.d is true:

REFER TO TASK 3.e HERE

2. if the return from task 3.d is false:

REFER TO TASK 3.f HERE

TASK 3.f will not map anything. it it divide the mapping page into even more sections. After task 3.f, we will need to redo task 3.d for each new section. then return task 3.e and task3.f depends on if task 3.d is true or false. If Task 3.f is ran, we will need to keep rerunning task 3.d and then 3.e/3.f until task 3.f is no longer running. Then we are done mapping

After everything. We check one last time if final is fully mapped, if not we return an error saying final is not mapped
we will now return the new mapping and the new final.

TASK 3.a: Mapping benzene ring
use the inputs from task 3 as necessary
array0 = [] (rename for clarification)
loop through this section:
for every atom that has index 1 = 'c' or 'C', index 3 length != 0, mapping[index 3[0]][index 3[1]][2] != 0, and final[index 0 of the atom] = "": add the index 0 of the atom to the array0
this array should have length 2, 4, 6: then we will use similar logic as in array1 section
we will use index 4 in the atom array in mapping which is the connection between atoms of the same section of each atom to find a way to split this array into pairs where each pair contains atoms that are neighbors (note that in the tuple in index 4, the atom index is not the index 0 of the atom array but rather the index of that atom in the section so we will need an intermediate step to find neighbors) - save the pairs as tuple of 2 index 0 of 2 the atom arrays
for every tuples in pairing:
    a = generate_random_string() (this function is already made and it will create a random 6 character string)

    final[index 0 of the atom1] = "TC5e" + a
    final[index 0 of the atom2] = "TC5e" + a

array1 = [] (rename for clarification)
loop through the section:
for every atom that has index 3 length = 0, final[index 0 of the atom] = "" add the index 0 of the atom to this empty array
(We know that the possible length of this array is either 3, 4, 5, or 6)
pairing = []
if the length is 6, we will use index 4 in the atom array in mapping which is the connection between atoms of the same section of each atom to find a way to split this array into pairs where each pair contains atoms that are neighbors (note that in the tuple in index 4, the

atom index is not the index 0 of the atom array but rather the index of that atom in the section so we will need an intermediate step to find neighbors) - save the pairs as tuple of 2 index 0 of 2 the atom arrays – there will be 3 tuples for sure
we will do the same for length = 5 with an extra constraint because since we have 1 atom left over, we make sure the left over atom is next to the last atom that has not been mapped (final of that atom is empty) – there will be 2 tuples for sure
we will do the same for length = 4 with an extra constraint because sometimes pairing is not doable. If it is not, just find 1 pair, any pair, and that will be it – there will be 1 or 2 tuples
we will do the same for length 3. if pairing is doable, make sure the 1 atom left over is next to an atom that has not been mapped (final of that atom is empty) - there will be 0 or 1 tuples
save the not-able-to-pair atoms into a list called array2 (rename for clarification)
for every tuples in pairing:
check the atom type (index 1) of the pair,
if they are both “c” or “C”:

```
a = generate_random_string()
final[index 0 of the atom1] = “TC5” + a
final[index 0 of the atom2] = “TC5” + a
```

if one atom is “n” or “N”:

```
a = generate_random_string()

final[index 0 of the atom1] = “TN6a” + a

final[index 0 of the atom2] = “TN6a” + a
```

array3 = [] (rename for clarification)
loop through the section:
for every atom that has index 3 length != 0, final[index 0 of the atom] = “”, mapping[index 3[0]][index 3[1]][2] == 0, and mapping[index 3[0]] length == 1 add the index 0 of the atom to this empty array
loop through this array:
if the atom in this array has exactly 1 neighbor that is in found in array2:
delete that neighbor in array 2
check the foreign atom type:
case “O”:

```
a = generate_random_string()
final[index 0 of the atom] = “SN6” + a
```

final[index 0 of the neighbor atom] = “SN6” + a
final[index 0 of the foreign atom] = “SN6” + a
case “N”:
a = generate_random_string()
final[index 0 of the atom] = “SN6d” + a
final[index 0 of the neighbor atom] = “SN6d” + a
final[index 0 of the foreign atom] = “SN6d” + a
case “S”:
a = generate_random_string()
final[index 0 of the atom] = “SC6” + a
final[index 0 of the neighbor atom] = “SC6” + a
final[index 0 of the foreign atom] = “SC6” + a
case “Cl”:
a = generate_random_string()
final[index 0 of the atom] = “SX3” + a
final[index 0 of the neighbor atom] = “SX3” + a
final[index 0 of the foreign atom] = “SX3” + a
case “I”:
a = generate_random_string()
final[index 0 of the atom] = “X1” + a
final[index 0 of the neighbor atom] = “X1” + a
final[index 0 of the foreign atom] = “X1” + a
case “C”:
a = generate_random_string()
final[index 0 of the atom] = “SC4” + a
final[index 0 of the neighbor atom] = “SC4” + a
final[index 0 of the foreign atom] = “SC4” + a
if the atom in this array has 2 neighbor that is found in array2: pick a neighbor that neighbor to an atom that is already mapped (final[that] != “”), if both are next to an atom that has not been mapped, pick any:
delete that neighbor in array 2

do the same casing as with the 1 neighbor case above

if the atom in this array does not have a neighbor that is in found in array2:
check the foreign atom type:
if “O” and the bond between the foreign atom and the atom is 2:
a = generate_random_string()

final[index 0 of the atom] = “TN6a” + a
final[index 0 of the foreign atom] = “TN6a” + a
if “O” and the bond between the foreign atom and the atom is 1:
a = generate_random_string()
final[index 0 of the atom] = “TN6” + a
final[index 0 of the foreign atom] = “TN6” + a

if “C”:

a = generate_random_string()
final[index 0 of the atom] = “TC4” + a
final[index 0 of the foreign atom] = “TC4” + a
array4 = [] (rename for clarification)
loop through the section:
for every atom that has index 3 length != 0, final[index 0 of the atom] = “”, mapping[index 3[0]][index 3[1]][2] == 0, and mapping[index 3[0]] length == 2 add the index 0 of the atom to this empty array
loop through this array:
check the foreign atom type:
if the foreign atom is O and the atom type of the atom connect to the foreign atom in the foreign atom region is C:
a = generate_random_string()
final[index 0 of the atom] = “SN2a” + a
final[index 0 of the foreign atom] = “SN2a” + a
final[index 0 of the foreign atom’s neighbor in its section] = “SN2a” + a
array5 = [] (rename for clarification)
loop through the section:
for every atom that has final[index 0 of the atom] = “”, add the index 0 of the atom to this array
this array should have length 2, 4, 6: then we will use the same logic as in array1 section
we will use index 4 in the atom array in mapping which is the connection between atoms of the same section of each atom to find a way to split this array into pairs where each pair contains atoms that are neighbors (note that in the tuple in index 4, the atom index is not the index 0 of the atom array but rather the index of that atom in the section so we will need an intermediate step to find neighbors) - save the pairs as tuple of 2 index 0 of 2 the atom arrays
for every tuples in pairing:
check the atom type (index 1) of the pair,

if they are both “c” or “C”:

a = generate_random_string()

final[index 0 of the atom1] = “TC5” + a

final[index 0 of the atom2] = “TC5” + a

if one atom is “n” or “N”:

a = generate_random_string()

final[index 0 of the atom1] = “TN6a” + a

final[index 0 of the atom2] = “TN6a” + a

Check if every atom in this section is mapped, if not return an error saying benzene ring is not fully mappable

return final

TASK 3.b: Mapping non-benzene 6-ring:

array = [] (rename for clarification)
loop through the section:
for every atom that has index 1 == “O” final[index 0 of the atom] = “”, add index 0 of the atom to this array
if array length == 1:

a = generate_random_string()

final[index 0 of the atom] = “SN4a” + a

final[index 0 of the innerneighbor1] = “SN4a” + a

final[index 0 of the innerneighbor2] = “SN4a” + a

if array length == 2:
for each i in array:

a = generate_random_string()

final[index 0 of the atom] = “SN3a” + a

final[index 0 of the innerneighbor1] = “SN3a” + a

final[index 0 of the innerneighbor2] = “SN3a” + a

array1 = [] (rename for clarification)

after everything, loop through the section again:

for every atom that has final[index 0 of the atom] = “”, add index 0 of the atom to this array if the size of this array is 3

a = generate_random_string()

for every atom in this array:

    final[array1[0]] = “SC3” + a
    final[array1[1]] = “SC3” + a
    final[array1[2]] = “SC3” + a

if the size of this array is 6: break it into sections of connected 3 and then:

for each section:

a = generate_random_string()

    final[array[0]] = “SC3” + a
    final[array[1]] = “SC3” + a
    final[array[2]] = “SC3” + a

Check if every atom in this section is mapped, if not return an error saying non-benzene 6-ring is not fully mappable

return final

TASK 3.c: Mapping non-benzene 5-ring:

array0 = [] (rename for clarification)

loop through the 5-ring section:

for every atom that has index 0 of the atom not found in the array0[], index 1 == “C”or “c”, final[index 0 of the atom] = “”, has a neighboring node with the index 1 of the neighboring node = “C” or “c” or ‘N’ or “n” and the bond between this node and the neighboring node is 2 add index 0 of the atom and the index 0 of the neighbor atom to this array as a tuple
array0 should contain 0-2 tuples now. For every tuple:
if index 1 of the 2 atoms are both ‘C’ or ‘c’:

a = generate_random_string()
final[atom 1] = “TC5” + a
final[atom 2] = “TC5” + a

if index 1 of the 2 atoms are ‘C’ or ‘c’ and one is ‘N’ or ‘n’:

a = generate_random_string()
final[atom 1] = “TN6a” + a
final[atom 2] = “TN6a” + a

After this step, every double bonds should be accounted for.

array1 = [] (rename for clarification)

loop through the 5-ring section:

for every atom that has index 1 != “C”or “c”, final[index 0 of the atom] = “” add index 0 of the atom to this array:

if length of array1 is 1:
case:
1.index 1 is “S” and final[index0 of the 2 inner neighbor atom] are both “”:
a = generate_random_string()
final[index 0 of S] = “SC6” + a
final[index 0 of left neighbor] = “SC6” + a
final[index 0 of right neighbor] = “SC6” + a

2.index 1 is “S” and final[index0 of the 2 inner neighbor atom] are both not “”:

a = generate_random_string()
final[index 0 of S] = “TC6” + a

3.index 1 is “NH”:

a = generate_random_string()

final[index 0 of NH] = “TN6d” + a

4.index 1 is “O” and final[index0 of the 2 inner neighbor atom] are both “”:

a = generate_random_string()
final[index 0 of S] = “TN4a” + a

final[index 0 of left neighbor] = “TN4a” + a

5.index 1 is “O” and final[index0 of the 2 inner neighbor atom] are both not “”:

a = generate_random_string()
final[index 0 of S] = “TN2a” + a

6. index 1 is “N” and it has a foreign connection (index 3 is not empty):

a = generate_random_string()
final[index 0 of N] = “TN1” + a
final[index 0 of the foreign neighbor] = “TN1” + a

if length of array1 is 2:
if index 1 of both of these atoms are “O”:
find a neighbor that both of these O shares:
once that neighbor is found:
a = generate_random_string()
final[index 0 of O1] = “SN5a” + a
final[index 0 of O2] = “SN5a” + a
final[index 0 of the matching neighbor] = “SN5a” + a

array2 = [] (rename for clarification)

loop through the 5-ring section:

for every atom that has index 1 == “C”or “c”, final[index 0 of the atom] = “” add index 0 of the atom to this array.
We know for sure this array will have either 5, 3, or 2 elements:
if it has 5:
loop through the array:
find an atom that has index 1 == “C” or “c” and index 3 length != 0 (has a foreign neighbor)
if found:
if the foreign neighbor index 1 == “C” or “c” and the size of the section that the foreign atom is in is 1:
a = generate_random_string()
final[index 0 of that atom] = “SC3” + a
final[index 0 of the foreign neighbor] = “SC3” + a
final[index 0 of the first inner neighbor of that atom] = “SC3” + a

if the foreign neighbor index 1 == “O” and the size of the section that the foreign atom is in is 1:

a = generate_random_string()
final[index 0 of that atom] = “SN6” + a
final[index 0 of the foreign neighbor] = “SN6” + a
final[index 0 of the first inner neighbor of that atom] = “SN6” + a

if the foreign neighbor section has a size of > 1

a = generate_random_string()
final[index 0 of that atom] = “TC3” + a

final[index 0 of the first inner neighbor of that atom] = “TC3” + a

after this step if any changes are made, we want to loop through the array and delete all the atom that has final[index 0] != “”, now we should really only have either 2 or 3 atom left

if the size is 2 and all the index 0 of the atoms are “c” or “C”:

a = generate_random_string()
final[index 0 atom 1] = “TC3” + a
final[index 0 atom 2] = “TC3” + a

if the size is 3 and all the index 0 of the atoms are “c” or “C”:

a = generate_random_string()
final[index 0 atom 1] = “SC3” + a
final[index 0 atom 2] = “SC3” + a

Check if every atom in this section is mapped, if not return an error saying non-benzene 5-ring is not fully mappable

return final

TASK 3.d: determine if the non-ring section is 1-bead mappable
use the inputs from task 3 as necessary
array0 = [] (rename for clarification)
loop through the section
for every atom that has isedge, add the index 0 of it to the array
if the size of this array is 1, return true
if the size of this array is 0, return error: can’t map non-ring section
if the size is > 1: use bfs or dfs to find the shortest path between every 2 edge atoms.

for example: if we have the mapping matrix be:
[[0, 'C', 0, [], [(1, np.float64(1.0)), (2, np.float64(1.0)), (3, np.float64(1.0))], False],
[1, 'F', 0, [], [(0, np.float64(1.0))], True],
[2, 'F', 0, [], [(0, np.float64(1.0))], True],
[3, 'F', 0, [], [(0, np.float64(1.0))], True]]
we see that there are 3 edge node, index 0 = 1, 2, 3. we will find the path between 1 and 2, 2 and 3, and 1 and 3. In this case we see that 1 can go to 0 can go to 2/3, same as 2 and 3, so the path between the edge atoms are all 3. If any of these path is more than 4 atom long, return false
else return true

TASK 3.e: Map the whole section using 1-2 bead
use the inputs from task 3 as necessary
array0 = [] (rename for clarification)
loop through the section
for every atom that has isedge, add the index 0 of it to the array
We should have an array with 2, 3, or 4 elements:
if we have an array with 2 elements:
use index 0, index 1, and index 4 from the atoms of the sections to find a string that represents the smiles structure of this section. For example:
string = “”
[[0, 'C', 0, [], [(1, np.float64(1.0))], True],
[1, 'C', 0, [], [(0, np.float64(1.0)), (2, np.float64(1.0))], False],
[2, 'C', 0, [], [(1, np.float64(1.0)), (3, np.float64(1.0))], False],
[3, 'C', 0, [], [(2, np.float64(1.0))], True]]
we will look at any edge atom, here we can either look at 0 or 3:
let’s say we choose to look from 0:
add the index 1 to the string (C)
(repeat)check index 1 of index 4, if it is 2 add = to the string, if it is 3 add # to the string (none added)
check index 0 of index 4, go to that index of the section and retrieve the index 1 of that atom and add it to the string (C)
check if this is the edge node if it is we are finish if it is not:
we do not want to go back to 0 so we will try to go to the index that has index 0 != one that has already been traced
once we find that tuple: go back to repeat and we will repeat until we reach the edge node.
Now we will have something that look like CCCC for this example
Look into the dictionary to retrieve all the keys that have value [0] = 7 and value [1] = 0.

array1 = [] (rename for clarification)
From these keys, we can check which key match perfectly with the string we have, we will check 2 times, the string and the reverse of the string, we will delete the duplicate keys and only keep unique keys.
at this point if we have 1 solution left, we can:

```
a = generate_random_string()
for i in section length:
final[i[0]] = key + a
```

if we have no solution left, we can:

add a todo to find a similarity match

for now just return an error saying non-ring section cannot be mapped

if we have more than 1 solution left:

check if the string is “CO” or “OC” if it is:

check if index 3 of any of these 2 edge nodes from array0[] has a foreign atom. If it is found, look at the final[index 0 of foreign atom node]. if it starts with “SX4e”:

```
a = generate_random_string()

for i in section length:
final[i[0]] = key + a


if it starts with “TC5”:
a = generate_random_string()
for i in section length:
final[i[0]] = key + a
```

we have now mapped all 2 edge atoms section with path from 1 edge to another edge <= 4

now if length of array0 = 3: (we now have 3 edges in this section)

array2 = [] (rename for clarification)

since we have 3 edges, there is guaranteed to have 1 atom in this section that have 3 inner connections: we will find this atom and add the index 1 of this atom to array2. now every connection this atom is pointing to will be a 1 path direction to any of the 3 edge nodes.

we will go through every path, for each path: (every tuples in index 4)

string i = “”

(repeatable) we check the index 1 to find out the bond: if it is 2 add = to the string, if it is 3 add # to string i

we check index 0 and go to the atom that is [index 0] of the section

we then add index 1 of that atom to string i

we check if this atom is an edge atom, if it is not we find the next into node that this atom is going to (dont traverse back) and repeat the process until the edge node is found if it is, add i to array2

Example: for the mapping scheme of a section being:

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

[1, 'F', 0, [], [(0, np.float64(1.0))], True],

[2, 'F', 0, [], [(0, np.float64(1.0))], True],

[3, 'F', 0, [], [(0, np.float64(1.0))], True]]

now we should have something that look like:

[“C”, “F”, “F”, “F”]

we will now Look into the dictionary to retrieve all the keys that have value [0] = 7 and value [1] = 1.

array3[]

for each keys, we can trace the value[2] which is a string:

check the first characters until ‘(‘ (excluding ‘(‘) to see if it matches with array1[0],

then delete that index from array1

then for every (...), check if the ... matches with array1[any], if it is delete that index from array 1, if it is not continue. if array 1 is empty which means mapping is a perfect match, we save the key into array3

if array3[] length is 0 return an error saying non-ring section cannot be mapped

if array3[] length is > 1:

Add a todo to decypher

for now just return an error saying non-ring section cannot be mapped

if array3[] length is 1:

    a = generate_random_string()

    for i in section length:
    final[i[0]] = array3[0] + a

we have now mapped all 3 edge atoms section with path from 1 edge to another edge <= 4

now if length of array0 = 4: (we now have 4 edges in this section)

array4 = [] (rename for clarification)

since we have 4 edges, there is guaranteed to have 1 atom in this section that have 4 inner connections: we will find this atom and add the index 0 of this atom to array4. now every connection this atom is pointing to will be a 1 path direction to any of the 4 edge nodes.

we will go through every path, for each path: (every tuples in index 4)

(repeatable) we check index 0 and go to the atom that is [index 0] of the section

we check if the edge node is yes and index 1 of that atom is “F”, we add the index 0 of that neighbor atom to array 4

we will check now if the length of array 4 is 4, if it is not return an error saying non-ring section cannot be mapped

if array4[] length is 4:

    a = generate_random_string()

    for i in array 4:
    final[i] = “SX4e" + a

array5 = [] (rename for clarification)

loop through the section
for every atom that has final[index 0 of the atom] = “” add index 0 of that atom to array5:

if length of array 5 is 2 and the two atoms are C and O:

    a = generate_random_string()
    for i in array 5:
    final[i] = “TP1d" + a

add a todo here for if there are more beads like this because currently there is only 1 case.

else return an error saying non-ring section cannot be mapped

return final

TASK 3.f: dealing with non-ring section that are too freaking long:

THIS TASK CAN BE THE HARDEST TASK SO FOR NOW JUST USE THE ONLY CASE WE HAVE ON THE TABLE AS EXAMPLE

For now there is only 1 case so let’s deal with that, leave this as a todo list for later

use the inputs from task 3 as necessary:

array0 = [] (rename for clarification)
loop through the section
for every atom that has isedge, add the index 0 of it to the array
use bfs or dfs to find the shortest path between all edge atoms (ex: if we have 4 atom, we should have to do a total of 6 path, 1-2, 1-3, 1-4, 2-3, 2-4, 3-4)

For each path that is <= 4, delete both edge nodes from array0

pick the first element left in array 0, we trace out 3 more nodes from it and we need to make sure none of the nodes have 3 or more inner connection or if the dividing is going to make one section have only length = 1, if there is, we will only trace out 2 extra nodes. we then divide the mapping into new sections, the section that is from the element and the 2/3 nodes it traces to, and the rest.

remember to account for all of inner/outter connection, edge nodes

REFER TO TASK 3.g here.

return the new mapping.

TASK 3.g: Given a mapping, and the division of the section, return the mapping correctly depicting the division of the section into a mapping where that section is now two separate working sections.

Things needed to be updates besides the section split:

The inner connection between the node that is splitted from will become an outer connection,

The isedge of those 2 atoms will also be changed from false to true.

More explanation of section 3.f and 3.g

the idea is not to split CCCCCCCC into 2 even pieces, it is to split CCCC into a section and the rest into another section, but it just happen that the other section is also just CCCC and we dont need to split them anymore because it already pass the test on 3d. Also when you split the mapping into 2 parts, you have to change a few section, including isedge at the atom that is splitted and at the atom that is splitted, the inner connection will become the outter connection. Also, we can only guarantee the section that is traced 2/3 nodes out can be mapped using 1 bead, for the other section that we just basically take whatever is left and put it in another section, we have to check using task 3.d again to see if the works if it is not we have to rerun task 3.f for that section and recursively keep going until all sections have connection of all edges that are no more than 4 node long

Remember that whenever we do the splitting, there are cases where we dont want to trace out 3 more but rather 2 more in cases where "we trace out 3 more nodes from it and we need to make sure none of the nodes have 3 or more inner connection or if the dividing is going to make one section have only length = 1, if there is, we will only trace out 2 extra nodes. we then divide the mapping into new sections, the section that is from the element and the 2/3 nodes it traces to, and the rest.". So the first case that we need to only trace out 2 nodes is when the third node is the central node. and then the second case that we only trace out 2 nodes is when the tracing causes the other section to only have 1 atom left

For now, CCCCC is throwing out the no candidate edge remains error. It should first try to trace out 3 other node to make CCCC but then it realized the left over only have 1 node left. Then it should traced out 2 nodes only to make CCC, this makes the other part make sense with CC and we should have two working parts that is CCC or CC

Explanation

1. Candidate Edges:
   We first collect candidate edges as those atoms with atom[5] == True. If none are found, we fall back to using the endpoints (indices 0 and len(section)-1).
2. BFS Removal:
   Only if there are more than two candidate edges do we run a BFS removal step (removing pairs with distance ≤ 4). If this removal empties the list, we again fallback to the endpoints.
3. DFS Trace:
   Starting from the chosen seed, we recursively trace a branch.
   a. Condition 1: If the trace is long enough (at least 4 nodes) and the fourth node (at index 3) has three or more inner connections, we shorten the trace to 3 nodes.
   b. We then ensure the trace is at most 4 nodes.
   c. Condition 2: We then check that the remainder (all atoms not in the trace) has at least 2 atoms. If not, we iteratively remove the last node of the trace until the remainder has at least 2 atoms.
4. Return Value:
   The function returns the subdivided sections along with the corresponding original local indices.

OK THIS WORKS NOW, HOWEVER LET'S REVISE IT EVEN BETTER:

What if we have a structure CCC(C)CC? This structure have a path of 5 between 2 edge nodes but if we use map_non_ring_section_long, we will encounter a very interesting issue, let's recall how this section functions:

This section calls a function that first put all of the edge nodes into a list and then take off any edge nodes that we can find another edge node that we can trace to with length of 4 or less. Then we take the node that is left in the list and trace out 2/3 nodes from it depending on the rules shown before.

However if we look here, every edge node is able to trace to another edge node with length <= 4. But the reason why this is considered non-1-bead-mappable is because there is a path that is >4 long (5 to be exact). Let's try to fix this:

If after the nodes are deleted from the list and the list is empty, we will have to separate the section another way. Here is how we would do it,

find the shortest (shortest is important here) path between any 2 edge nodes that are length <=4 and make it 1 section, the rest will be another section, In this case:

remember that we will have 1 “center” node that has 3 or more inner connection, and we can guarantee that we can find this center node in the section that we just made, because the path that connect 2 edge nodes need to cross that center node.

when these section are splitted, the bordering node that needs to be fixed the outer connections are now the centernode, and the node that connect to the centernode in the other section.

Moreover, the isedge fixing is different aswell. After splitting, we will fix the inner and outter connection first, then, we check the 2 nodes that we just fixed and find out if it is isedge by checking its inner connection array to see if the length is 1 then it is isedge = true. Otherwise the isedge stay as false. Then we process these 2 sections as normal.

we can create more functions if needed, especially for the different splitting consideration, we might need a different function that handle the splitting in this special case

More fine-tuning:

we have created a case for if only 1 node is left in the array, we have also created a case for if no nodes are left in the array, how about if more than 1 nodes is left in the array? Let’s change the logic of the section above is that when only 1 node is left in the array, we trace out 2/3 nodes from that node instead of as long as there is a candidate edge we use it. We only run
seed = candidate_edges[0] if candidate edge have only 1 element left, if there are 2 or more, we do something similar to the previous step:

find the shortest (shortest is important here) path between any 2 edge nodes that are make it 1 section, this time the length does not need to be <=4, the rest will be another section, In this case:

remember that we will have 1 “center” node that has 3 or more inner connection, and we can guarantee that we can find this center node in the section that we just made, because the path that connect 2 edge nodes need to cross that center node.

when these section are splitted, the bordering node that needs to be fixed the outer connections are now the centernode, and the node that connect to the centernode in the other section.

Moreover, the isedge fixing is different aswell. After splitting, we will fix the inner and outter connection first, then, we check the 2 nodes that we just fixed and find out if it is isedge by checking its inner connection array to see if the length is 1 then it is isedge = true. Otherwise the isedge stay as false. Then we process these 2 sections as normal.

we can create more functions if needed, especially for the different splitting consideration, we might need a different function that handle the splitting in this special case