23 Oct 2023, 3 Aug 2023 , 3 May 2023

You will be given a list of strings as a database of some DNAs of Monocells. Given a child DNA, write a function find_parents(child_dna,dna_database) to return a list of possible pairs of parents that can produce that child Monocell where each pair is expressed as a tuple. Note that the orders of the parents matter in each tuple. The child's DNA must be the one of the first parent followed by the second one in a tuple.

In [1]:
```
def checkSubset(parent, child):

    plength = len(parent)

    for i in range(plength):

        if child[i] != parent[i]:

            return None

    return (parent, child[plength:])
```

In [2]:
```
checkSubset('ab', 'abb')
```

Out[2]:
```
('ab', 'b')
```

In [3]:
```
def find_parents(child_dna,dna_database):

    subsetResult = {}
    indexResult = {}
    result = []
    databaseLength = len(dna_database)

    for elt in dna_database:

        check =  checkSubset(elt,child_dna)

        if check != None and check[0] != check[1]:

            subsetResult[check[0]] = check[1]

    for i in range(databaseLength):

        indexResult[dna_database[i]] = i

    for parent1 in subsetResult.keys():

        parent2 = subsetResult.get(parent1)

        if indexResult.get(parent2) != None:

            result.append((parent1,parent2))

    return result
```

In [4]:
```
find_parents('ACGTA',['ACT','TA','CGTA','ACG','A','ACGT'])
```

Out[4]:
```
[('ACG', 'TA'), ('A', 'CGTA'), ('ACGT', 'A')]
```

In [5]:
```
print(find_parents('ATGATG',['ATGAT','ACT','GAT','ATG','G']))
```

```
[('ATGAT', 'G')]
```

In [6]:
```
from itertools import product
w10 = [''.join(x) for x in product(list('ACGT'),repeat=10)]
```

In [7]:
```
print(find_parents('ACGTTTTTTAATATTTATGG',w10))
```

```
[('ACGTTTTTTA', 'ATATTTATGG')]
```

Write a function three_little_pigs_defence(seq) to take in a sequence of actions seq and return the final string of what is left over after all events happened with len(seq)≥ 0. Your return string should have the correct order according to the events.

In [8]:
```
def three_little_pigs_defence(stringInput):

    length = len(stringInput)

    if stringInput[0] == 'H':

        return three_little_pigs_defence(stringInput[1:])

    for i in range(1,length):

        if stringInput[i] == 'H' and stringInput[i-1] != 'B':

            result = stringInput[:i-1] + stringInput[i+1:]

            return three_little_pigs_defence(result)

        elif stringInput[i] == 'H' and stringInput[i-1] == 'B':

            result = stringInput[:i] + stringInput[i+1:]

            return three_little_pigs_defence(result)

    return stringInput
```

In [9]:
```
three_little_pigs_defence('SBWHHW')
```

Out[9]:
```
'SBW'
```

In [10]:
```
print(three_little_pigs_defence('SHHWSHB'))
```

```
WB
```

In [11]:
```
three_little_pigs_defence('BSHHSWHS')
```

Out[11]:
```
'BSS'
```

Write a Python function $prefix(aString, keys)$ that takes in a text string and a list of keys, and returns the longest key in the list that matches the beginning of $aString$. It returns the Boolean value False otherwise

In [12]:
```
x = 'ACGTTTTTTAATATTTATGG'
print(x[:1])
```

```
A
```

In [13]:
```
def prefix(aString,keys):

    prefixDict = {}
    stringLength = len(aString)
    current = ''
    currentLength = 0

    for i in range(stringLength):

        prefixDict[aString[:i+1]] = i+1

    for elt in keys:

        if prefixDict.get(elt) != None and prefixDict.get(elt) > currentLength:

            current = elt
            currentLength = prefixDict.get(elt)

    if currentLength == 0:

        return False

    else:

        return current
```

In [14]:
```
prefix('thout',['t','a','th'])
```

Out[14]:
```
'th'
```

In [15]:
```
prefix('Thas', ['t','a','th'])
```

Out[15]:
```
False
```

In [16]:
```
prefix('ttas',['t','a','th'])
```

Out[16]:
```
't'
```

Write a Python function ***trans***(***aString***, ***dictionary***) that uses *dictionary* to translate *aString*. The function returns the translated string

In [17]:
```
x = 'ACGTTTTTTAATATTTATGG'
print(x[2:])
```

```
GTTTTTTAATATTTATGG
```

In [18]:
```
def trans(aString,dictionary):

    result = ''
    stringLength = len(aString)
    i = 0
    #currentString = aString


    while i < stringLength:

        currentString = aString[i:]
        currentStringLength = len(currentString)
        currentAlphabet = aString[i]
```

```
        currentPrefix = ''
        prefixDict = {}
        currentPrefixLength = 0
        flag = False

        for j in range(currentStringLength):

            analyze = currentString[:j+1]
            prefixDict[analyze] = j+1

        for elt in dictionary.keys():

            if prefixDict.get(elt) != None and prefixDict.get(elt) > currentPrefixL
                flag = True
                currentPrefix = elt
                currentPrefixLength = prefixDict.get(elt)


        if flag == True:

            result += dictionary.get(currentPrefix)
            i += len(currentPrefix)


        else:

            result += currentAlphabet
            i += 1


    return result
```

In [19]:
```
tmap = { 'th' : 'zh', 'a' : 'ai', 't' : 'se' }
trans('Without That Breath!',tmap)
```

Out[19]:
```
'Wizhouse Thaise Breaizh!'
```

In [20]:
```
trans("that's not the sameth",tmap)
```

Out[20]:
```
"zhaise's nose zhe saimezh"
```

In [21]:
```
def transR(aString,dictionary):

        currentString = aString
        currentStringLength = len(currentString)

        if currentStringLength == 0:

            return ''


        currentAlphabet = aString[0]
        currentPrefix = ''
        prefixDict = {}
        currentPrefixLength = 0
        flag = False

        for j in range(currentStringLength):

            analyze = currentString[:j+1]
            prefixDict[analyze] = j+1

        for elt in dictionary.keys():
```

```
            if prefixDict.get(elt) != None and prefixDict.get(elt) > currentPrefixL
                flag = True
                currentPrefix = elt
                currentPrefixLength = prefixDict.get(elt)


        if flag == True:

            return dictionary.get(currentPrefix) + transR(aString[len(currentPrefix

        else:

            return currentAlphabet + transR(aString[1:],dictionary)
```

In [22]:
```
transR('Without That Breath!',tmap)
```

Out[22]:
```
'Wizhouse Thaise Breaizh!'
```

In [23]:
```
transR("that's not the sameth",tmap)
```

Out[23]:
```
"zhaise's nose zhe saimezh"
```

Write the function teleport(x) to return the integer of the cell that a white wizard can teleport to from cell x.

In [24]:
```
def teleport(x):

    return (3*x**3 + 7)%100
```

In [25]:
```
teleport(45)
```

Out[25]:
```
82
```

In [26]:
```
teleport(82)
```

Out[26]:
```
11
```

In [27]:
```
teleport(11)
```

Out[27]:
```
0
```

If a white wizard starts from cell x, how many times does he need to teleport to cell 0? Write a function number_of_teleport(x) to return the integer of number of teleports he needs to exit the castle if he starts from cell x. For example, if a white wizard starts at cell 45

In [28]:
```
def number_of_teleport(x):

    numDict = {}
    result = x
    numDict[result] = True
    count = 0

    while result != 0:

        result = teleport(result)
```

```
        count += 1

        if numDict.get(result) == None:

            numDict[result] = True

        else:

            return "Trapped"

    return count
```

In [29]:
```
number_of_teleport(45)
```

Out[29]:
```
3
```

In [30]:
```
number_of_teleport(17)
```

Out[30]:
```
5
```

In [31]:
```
number_of_teleport(96)
```

Out[31]:
```
4
```

In [32]:
```
number_of_teleport(83)
```

Out[32]:
```
'Trapped'
```

In [33]:
```
number_of_teleport(39)
```

Out[33]:
```
'Trapped'
```

In [34]:
```
number_of_teleport(40)
```

Out[34]:
```
6
```