#! /usr/bin/env python

**import** sys, math, os, argparse, csv

csv.field\_size\_limit(sys.maxsize)

# January 16, 2018 version 1.1

# Find dispersed repeated sequences in genomes.

# Designed for plant mitochondrial genomes of up to a few Mbp.

# May be very slow with larger genomes.

# Blast can also sometimes give odd results with large or highly repetetive genomes.

# Gaps, or runs of 'N's in the sequence will definitely give weird results.

# The program assumes there aren't any, and that the longest repeat will be the full sequence to itself.

# If there are long repeats in the output that are listed as being only at one location, this is probably what happened.

# If there are a lot of repeats within repeats the results can also be odd.

# Copyright Alan C. Christensen, University of Nebraska, 2018

# No guarantees, warranties, support, or anything else is implicit or explicit.

# Input is a fasta format file of a sequence. Genbank format works but generates lots of error messages in stdout.

# Output is a list of unique, ungapped repeated sequences, fasta formatted.

# The names are in the format '>Repeat/ROUS\_name\_start\_end\_length'.

# Percent identity is limited to >=99%, to allow for sequencing errors of <1%.

# A table of repeats with the coordinates of each one is generated.

# A list of repeat name, length and copy number is generated.

# A binned table of the total number of repeats in size ranges is generated.

#

# PARAMETERS

#   REQUIRED:

#      input file in fasta format

#   Optional

#      -o output file name

#      -m minimum length of exact matches to keep

#      -b path to blastn (default is /usr/bin/)

#      -k keep temp files

#      -gb to write the repeats to a genbank format file

parser = argparse.ArgumentParser(description='Find repeats in a fasta sequence file')

parser.add\_argument('infile', action='store', help='Input .fasta file')

parser.add\_argument('-o', action='store', dest='outfile', help='Output file name seed, default is input\_repeats', default='default')

parser.add\_argument('-m', action='store', dest='minlen', help='Minimum length of matches to keep, default=24', default='24')

parser.add\_argument('-b', action='store', dest='blast\_path', help='Path to blastn program, default is /usr/bin/', default='/usr/bin/')

parser.add\_argument('-k', action='store\_true', dest='keep', help='True to keep temp files', default=False)

parser.add\_argument('-gb', action='store\_true', dest='genbank', help='True to write GenBank format file', default=False)

results = parser.parse\_args()

infile = results.infile

outfile = results.outfile

minlen = int(results.minlen)

blast\_path = results.blast\_path

keep = results.keep

genbank = results.genbank

# It might be useful to define the wordsize as something less than minlen, so both variables are used.

# Wordsize smaller than minlen would give smaller core identical sequences in the middle of repeats.

# An example might be to change this to wordsize = str(int(minlen/2)).

wordsize = str(minlen)

# If no output file seed is specified, make one by stripping leading directory information

# and stripping trailing .fa or .fasta from the input file name and using that.

**if** outfile == 'default':

    outfile = infile

    **if** outfile.count('/') > 0:

        **for** i **in** range(outfile.count('/')):

            index = outfile.index('/')

            outfile = outfile[index+1:]

    **if** outfile.endswith('.fa') **or** outfile.endswith('.fasta'):

        outfile = outfile.rstrip('fasta')

    outfile = outfile.rstrip('.')

outfa = outfile+'\_rep.fasta'

outtab = outfile+'\_rep\_table.txt'

outbin = outfile+'\_binned.txt'

outcount = outfile+'\_rep\_counts.txt'

outgb = outfile+'\_repeats.gb.txt'

tempblast = outfile+'\_tempblast.txt'

temprepeats = outfile+'\_temprepeats.txt'

tempparse = outfile+'\_sequence\_parsing.txt'

# Get sequence name and length from fasta file.

seq = open(infile, 'r')

seqname = seq.readline()

seqname = seqname.lstrip('> ')

seqname = seqname.rstrip()

seqlen = 0

**for** line **in** seq:

    **if**(line[0] == ">"):

        **continue**

    seqlen += len(line.strip())

seq.close()

# run blastn with query file plus strand (removing first line which is full length sequence), minus strand, and concatenate

**print** 'Performing self-blastn comparison with '+seqname

os.system(blast\_path+'blastn -query '+infile+' -strand plus -subject '+infile+' -word\_size '+wordsize+' -reward 1 -penalty -20 -ungapped -dust no -soft\_masking false -evalue 1000 -outfmt "10 qstart qend length sstart send mismatch sstrand qseq" | tail -n+2 > tempblast1.txt')

os.system(blast\_path+'blastn -query '+infile+' -strand minus -subject '+infile+' -word\_size '+wordsize+' -reward 1 -penalty -20 -ungapped -dust no -soft\_masking false -evalue 1000 -outfmt "10 qstart qend length sstart send mismatch sstrand qseq" > tempblast2.txt')

os.system('cat tempblast1.txt tempblast2.txt > '+tempblast)

os.system('rm tempblast1.txt tempblast2.txt')

# open tempblast.txt, convert to list of lists, and sort by length and position descending

# This is necessary because blastn does not output every possible pair of hits when there are more than 2 copies of a repeat

**print** 'Sorting alignments...'

f = open(tempblast, 'r')

reader = csv.reader(f)

alignments = list(reader)

f.close()

alignments = sorted(alignments, key=**lambda** x: (-1\*int(x[2]), -1\*int(x[0])))

alignments.append(['1','1','1','1','1','0','A','X'])

# New list of uniques

# Text file '\_sequence\_parsing.txt' includes the information on how duplicates were found.

# Start at row 0. Compare to subsequent rows.

# If repeat length is different from the next row, it has passed all the tests, write it to the file.

# If query or subject coordinates are the same as the query or subject or reversed coordinates

# of a subsequent row, it is not unique, so go to the next row and do the comparisons again.

# Thanks to Alex Kozik for repeatedly testing and finding bugs in the algorithm.

**print** 'Finding unique repeats...'

uniques = []

sp = open(tempparse, 'w')

**for** row **in** range(len(alignments)):

    sp.write('row '+str(row)+'\n')

    **if** int(alignments[row][2]) < minlen:

        # This won't happen unless the word\_size is defined as something other than minlen.

        # That could be useful under some circumstances.

        sp.write('row '+str(row)+' is less than minlength')

        **break**

    **else**:

        **for** compare **in** range(row+1,len(alignments)):

            **if** alignments[row][2] != alignments[compare][2]:

                uniques.append(alignments[row])

                sp.write('\tadding row '+str(row)+' to unique list\n')

                **break**

            **else**:

                sp.write('\tcomparing to '+str(compare)+'\n')

                **if** alignments[row][0] == alignments[compare][0] **and** alignments[row][1] == alignments[compare][1]:

                    sp.write('\tqstart and qend of row '+str(row)+' and '+str(compare)+' are the same\n')

                    **break**

                **elif** alignments[row][0] == alignments[compare][1] **and** alignments[row][1] == alignments[compare][0]:

                    sp.write('\tqstart and qend of row '+str(row)+' is the same as qend and qstart of '+str(compare)+'\n')

                    **break**

                **elif** alignments[row][0] == alignments[compare][3] **and** alignments[row][1] == alignments[compare][4]:

                    sp.write('\tqstart and qend of row '+str(row)+' is the same as sstart and send of '+str(compare)+'\n')

                    **break**

                **elif** alignments[row][0] == alignments[compare][4] **and** alignments[row][1] == alignments[compare][3]:

                    sp.write('\tqstart and qend of row '+str(row)+' is the same as send and sstart of '+str(compare)+'\n')

                    **break**

                **elif** alignments[row][3] == alignments[compare][0] **and** alignments[row][4] == alignments[compare][1]:

                    sp.write('\tsstart and send of row '+str(row)+' is the same as qstart and qend of '+str(compare)+'\n')

                    **break**

                **elif** alignments[row][3] == alignments[compare][1] **and** alignments[row][4] == alignments[compare][0]:

                    sp.write('\tsstart and send of row '+str(row)+' is the same as qend and qstart of '+str(compare)+'\n')

                    **break**

                **elif** alignments[row][3] == alignments[compare][3] **and** alignments[row][4] == alignments[compare][4]:

                    sp.write('\tsstart and send of row '+str(row)+' is the same as sstart and send of '+str(compare)+'\n')

                    **break**

                **elif** alignments[row][3] == alignments[compare][4] **and** alignments[row][4] == alignments[compare][3]:

                    sp.write('\tsstart and send of row '+str(row)+' is the same as send and sstart of '+str(compare)+'\n')

                    **break**

                **else**:

                    sp.write('\t'+str(row)+' is different\n')

sp.close()

# Write uniques into output file

# Start list for copy number table

rous\_count = 0

g = open(outfa, 'w')

repcopies = []

**for** i **in** range(len(uniques)):

    qstart = uniques[i][0]

    qend = uniques[i][1]

    length = uniques[i][2]

    seq = uniques[i][7]

    rous\_count += 1

    g.write('>Repeat\_'+str(rous\_count)+'\n'+seq+'\n')

    repcopies.append(['Repeat\_'+str(rous\_count),length])

**if** rous\_count == 0:

    **print** "\tRepeats of unusual size? I don't think they exist"

g.close()

**print** 'Repeat fasta file is done, as you wish.'

# Now find each copy of each repeat. Again, this is because the blastn output file does not have every possible alignment.

# It is also because the information on locations and strand is not organized well in the blastn output.

# In addition, this subroutine eliminates duplicates of nested repeats.

**print** "Finding all copies of repeats..."

g = open(outfa, 'r')

os.system(blast\_path+'blastn -query '+outfa+' -strand both -subject '+infile+' -word\_size '+wordsize+' -reward 1 -penalty -20 -ungapped -dust no -soft\_masking false -evalue 1000 -outfmt "10 qseqid length sstart send sstrand qcovhsp" > '+temprepeats)

g.close()

tempr = open(temprepeats, 'r')

reader = csv.reader(tempr)

replist = list(reader)

tempr.close()

**print** "Making a table of the repeats..."

sum\_rep\_len = 0

bin\_dict = {}

binned = [seqname,seqlen,0]

# defining the bins

i = 0

j = 50

**while** j < 1000:

    bin\_dict[i] = j

    binned.append(0)

    i += 1

    j += 50

**while** j <= 10000:

    bin\_dict[i] = j

    binned.append(0)

    i +=1

    j += 250

# make list for entire sequence, set each position as 0

posit = []

**for** n **in** range(seqlen):

    posit.append(0)

# Thanks to Emily Wynn for suggesting qcovhsp for this loop.

# if qcovhsp is >98%, write to the file

# write tab separated values of repeat name, length, start, end, strand to outtab

# make list for genbank file

# Keep stats on lengths

rt = open(outtab, 'w')

rt.write(seqname+'\t'+str(seqlen)+'\n')

templist = []

gblist =[]

# look at each repeat in turn

**for** i **in** range(len(replist)):

    # if repeat is good (>98% identical to another one), write it to the file, and put the name in a list

    **if** int(replist[i][5])>98:

        rt.write(str(replist[i][0])+'\t'+str(replist[i][1])+'\t'+str(replist[i][2])+'\t'+str(replist[i][3])+'\t'+str(replist[i][4])+'\n')

        **if** replist[i][4] == 'minus':

            location = 'complement('+replist[i][3]+'..'+replist[i][2]+')'

        **else**:

            location = replist[i][2]+'..'+replist[i][3]

        gblist.append('     repeat\_region   '+location+'\n                     /rpt\_type=dispersed\n                     /label='+replist[i][0]+'\n')

        templist.append(replist[i][0])

        # then write 1's at every position in the sequence covered by that repeat

        # these can then be summed to get total bases of repeats

        # bases in overlapping repeats are only counted once

        **for** n **in** range(int(replist[i][2]), int(replist[i][3])):

            posit[n] = 1

        # then scan through bin sizes and if a repeat is greater than the

        # bin\_dict size cutoff, add one to the bin

        **for** j **in** range(len(binned)-4, -1, -1):

            **if** int(replist[i][1]) >= bin\_dict[j]:

                binned[j+3] +=1

                **break**

sum\_rep\_len = posit.count(1)

binned[2] = sum\_rep\_len

rt.close()

**if** genbank == True:

    gb = open(outgb, 'w')

    **for** i **in** range(len(gblist)):

        gb.write(gblist[i])

    gb.close()

# write tab separated values of repeat name, length, copy number to outcount

# first two lines are also a table of stats on repeats

rc = open(outcount,'w')

rc.write('Sequence\tGenome\_size\tNumRepeats\tAvgSize\tAvgCopyNum\n')

numrous = 0

sizerous = 0

copyrous = 0

**for** i **in** range(len(repcopies)):

    repname = repcopies[i][0]

    replen = float(repcopies[i][1])

    repcop = float(templist.count(repname))

    numrous += 1

    sizerous += replen

    copyrous += repcop

**if** numrous == 0:

    avsizerous = 'NA'

    avcopyrous = 'NA'

**else**:

    avsizerous = sizerous/numrous

    avcopyrous = copyrous/numrous

rc.write(seqname+'\t'+str(seqlen)+'\t'+str(numrous)+'\t'+str(avsizerous)+'\t'+str(avcopyrous)+'\n')

**for** i **in** range(len(repcopies)):

    rc.write(repcopies[i][0]+'\t'+repcopies[i][1]+'\t'+str(templist.count(repcopies[i][0]))+'\n')

rc.close()

# Write binned table headers, then stats for this sequence.

binfile = open(outbin, 'w')

binfile.write('Sequence\tSeq\_len\tRep\_len\t')

**for** i **in** range(len(bin\_dict)):

    binfile.write(str(bin\_dict[i])+'\t')

binfile.write('\n')

**for** i **in** range(len(binned)):

    binfile.write(str(binned[i])+'\t')

binfile.write('\n')

binfile.close()

**print** "Repeat tables are done, as you wish."

# Removing temp files if necessary

**if** keep == False:

    os.system('rm '+tempblast+' '+temprepeats+' '+tempparse)

# Rachael Schulte, William Goldman and Rob Reiner inspired this section of code

quote\_dict = {0:"48656c6c6f2e204d79206e616d6520697320496e69676f204d6f6e746f79612e20596f75206b696c6c6564206d79206661746865722e205072657061726520746f206469652e", 1:"5768656e20492077617320796f7572206167652c2074656c65766973696f6e207761732063616c6c656420626f6f6b732e", 2:"486176652066756e2073746f726d696e2720646120636173746c6521", 3:"4d79207761792773206e6f7420766572792073706f7274736d616e6c696b652e", 4:"596f75206b656570207573696e67207468617420776f72642e204920646f206e6f74207468696e6b206974206d65616e73207768617420796f75207468696e6b206974206d65616e732e", 5:"4d75726465726564206279207069726174657320697320676f6f642e",6:"496e636f6e6365697661626c6521", 7:"5468657265277320612062696720646966666572656e6365206265747765656e206d6f73746c79206465616420616e6420616c6c20646561642e", 8:"596f7520727573682061206d697261636c65206d616e2c20796f752067657420726f7474656e206d697261636c65732e", 9:"476f6f64206e696768742c20576573746c65792e20476f6f6420776f726b2e20536c6565702077656c6c2e2049276c6c206d6f7374206c696b656c79206b696c6c20796f7520696e20746865206d6f726e696e672e",10:"4e6f206d6f7265207268796d65732c2049206d65616e2069742120416e79626f64792077616e742061207065616e75743f"}

**import** random, binascii

z = random.randint(0,10)

**print** binascii.unhexlify(quote\_dict[z])+'\n'