```
import re
import time
from collections import Counter
```

Il file di input deve essere di reads duplicati in cui l'id è terminato da `_1` (originale) oppure `_0` (dopo reverse and complement).

Dimensione e minima lunghezza totale dei kmers:

```
k = 7
min_total_length = 40
```

Minimo numero di kmers (unici) che due reads in overlap devono condividere (in assoluto):

```
min_shared_kmers = 4
```

Massimo numero di occorrenze di un kmer nei reads (valore compatibile con la coverage dell'input):

```
max_kmer_occurrence = -1
```

Un valore pari a -1 indica assenza del controllo.

Parametri di filtraggio delle regioni comuni in output:

- massima differenza percentuale tra le lunghezze (bp) delle due stringhe della regione comune (mettere 0.0 nel caso di reads senza errore)
- minima lunghezza (bp) delle due stringhe della regione comune

```
max_diff_region_percentage = 0.0
min_region_length = 100
```

Parametro di filtraggio delle regioni comuni:

- minima percentuale di copertura dei kmers contigui rispetto alla lunghezza della regione comune.

Una regione lunga L bp deve contenere almeno `%L/min_total_length`di kmers comuni

```
min_region_kmer_coverage = 0.27
```

Parametri di filtraggio degli overlap in output:

- minima copertura percentuale (bp) della regione comune rispetto all'overlap
- minima lunghezza dell'overlap da produrre in output (1200 per read senza errore)

```
min_overlap_coverage = 0.70

min_overlap_length = 600
```

```
start_time_begin = time.perf_counter()
```

Lettura delle fingerprints

Ogni fingerprint deve essere su unica riga.

Esempio:

```
ID 45 7 9 1 1 | 7 65 2 3 54 |
```

L'eventuale simbolo | separa i segmenti di fingerprint.

```
begin_time = start_time = time.perf_counter()

with open('./fingerprint_CFL_ICFL_COMB-30_s_300_noerr.txt', 'r') as
input_file:
    file_rows = input_file.readlines()

whole_rows = [re.findall(r'[^\s|]+', row) for row in file_rows]
```

Determinare la lista delle fingerprint e degli identificatori dei reads.

```
read_ids = []
fingerprint_list = []

for row in whole_rows:
    read_ids.append(row.pop(0))
    fingerprint_list.append(list(map(int, row)))

end_time = time.perf_counter()
print('Upload the fingerprint: ', end_time-start_time)
```

Numero di fingerprints da processare.

```
len(fingerprint_list)
```

Dizionario delle occorrenze dei kmers unici aventi minima lunghezza totale

a) Costruire un dizionario:

- chiave: kmer

- valore: lista di tuple `(read, start)`, dove `read` è la posizione di una fingerprint all'interno della lista delle fingerprint e `start` è la posizione 0-based di inizio dell'occorrenza del kmer "chiave" all'interno della fingerprint. Le tuple sono ordinate per valore crescente del valore `read` e poi per `start`.

NB: un kmer viene rappresentato dalla stringa ottenuta concatenando le lunghezze con il separatore _. Ad esempio il kmer `(34,6,7,8)` viene rappresentato dalla stringa `34_6_7_8`

```
def compute_kmer_occurrences(fingerprint_list):
    kmer_occurrences = {}

    for (j,finger) in enumerate(fingerprint_list):
        check_unique = []
        occ_kmer_list = []
        for (i,c) in enumerate(finger):
            kmer = tuple(finger[i:i+k])
            if len(kmer) == k:
                check_unique.append(kmer)
                occ_kmer_list.append((j,i,kmer))

        c = Counter(check_unique)

        for kmer_t in occ_kmer_list:
            #if c[kmer_t[2]] == 1:
            if c[kmer_t[2]] >= 1:
                cfr_kmer = kmer_t[2]

                if sum(cfr_kmer) >= min_total_length:
                    cfr_kmer = '_'.join(list((map(str, cfr_kmer))))

                    value = kmer_occurrences.get(cfr_kmer, [])
                    value.append((kmer_t[0], kmer_t[1]))
                    kmer_occurrences[cfr_kmer] = value

    for kmer in kmer_occurrences:
        kmer_occurrences[kmer] = tuple(kmer_occurrences[kmer])

    return kmer_occurrences

start_time = time.perf_counter()

%time kmer_occurrences = compute_kmer_occurrences(fingerprint_list)

end_time = time.perf_counter()
print('Computing the kmer occurrences: ', end_time-start_time)
```

Eliminare i kmer che occorrono una volta sola nel set dei reads oppure che occorrono troppe volte.

```
h_kmer_occurrences = dict()
for kmer in kmer_occurrences:
```

```
    size = len(kmer_occurrences[kmer])
    if size > 1 and (max_kmer_occurrence == -1 or size <=
max_kmer_occurrence):
        h_kmer_occurrences[kmer] = kmer_occurrences[kmer]

kmer_occurrences = h_kmer_occurrences
```

Dizionari dei leftmost e rightmost kmers comuni

Costruire i dizionari:

- chiave: `(read1, read2)`
- valore: `[start1, start2]`

Nei dizionari "leftmost" e "righttmost" vengono memorizzati per ogni coppia `(read1, read2)` il leftmost e il rightmost (rispetto a `read1`) kmer comune, rispettivamente.

NB: viene anche costruito il dizionario del numero di kmers condivisi da usare in seguito:

- chiave: `(read1, read2)`
- valore: numero di kmers unici condivisi

```
def compute_matches(kmer_occurrences):
    min_sharing_dict = {}
    matches_dict = {}

    for (p, kmer) in enumerate(kmer_occurrences):
        occ_list = kmer_occurrences[kmer]
        for (i, first_occ) in enumerate(occ_list):
            read1 = first_occ[0]
            for second_occ in occ_list[i+1:]:
                read2 = second_occ[0]

                min_sharing_dict[(read1, read2)] =
min_sharing_dict.get((read1, read2), 0) + 1

                value = matches_dict.get((read1, read2), [-1, -1, -1,
-1])

                if value[0] == -1 or value[0] > first_occ[1]:
                    value[0] = first_occ[1]
                    value[1] = second_occ[1]

                if value[2] == -1 or value[2] < first_occ[1]:
                    value[2] = first_occ[1]
                    value[3] = second_occ[1]

                matches_dict[(read1, read2)] = value

    return (min_sharing_dict, matches_dict)
```

```
start_time = time.perf_counter()

%time (min_sharing_dict, matches_dict) =
compute_matches(kmer_occurrences)

end_time = time.perf_counter()
print('Computing the matches: ', end_time-start_time)
```

Produrre in output le regioni comuni e gli overlaps

Per ogni coppia di reads, si hanno a disposizione:

- il leftmost kmer comune L
- il rightmost kmer comune R

Si deve verificare se L ed R danno origine a una regione comune (coppia di sottostringhe) sui due reads, che sarà la coppia di sottostringhe che partono da L e arrivano fino a R.

Per costruzione, si ha che su `read1` lo start di L è sempre ≤ dello start di R.

La regione comune viene prodotta in output solo se lo start di L su `read2` è ≤ dello start di R su `read2`.

Da una regione comune viene prodotto un overlap se la regione copre a sufficienza l'overlap.

Al termine, tutti gli overlaps vengono riconciliati e ogni overlap verrà prodotto come record dei campi seguenti:

- id del primo read (senza il terminatore di strand)
- lunghezza del primo read
- posizione 0-based di inizio dell'overlap sul primo read
- posizione 1-based di fine dell'overlap sul primo read
- id del secondo read (senza il terminatore di strand)
- lunghezza del secondo read
- posizione 0-based di inizio dell'overlap sul secondo read
- posizione 1-based di fine dell'overlap sul secondo read
- strand del secondo read rispetto al primo (0: se uguale; 1: se opposto)

Calcolo e riconciliazione degli overlaps.

```
overlap_dict = {}

start_time = time.perf_counter()

for (read1, read2) in matches_dict:
    if min_shared_kmers == 1 or min_sharing_dict[(read1, read2)] >=
min_shared_kmers:
```

```
        (first_match1, first_match2, second_match1, second_match2) =
matches_dict[(read1, read2)]

        if second_match2 >= first_match2:
            #print('here ', min_sharing_dict[(read1, read2)])
            (start1, end1, start2, end2) = (first_match1,
second_match1+k, first_match2, second_match2+k)

            up1 = sum(fingerprint_list[read1][:start1])
            up2 = sum(fingerprint_list[read2][:start2])
            l1 = sum(fingerprint_list[read1][start1:end1])
            l2 = sum(fingerprint_list[read2][start2:end2])
            read1_length = sum(fingerprint_list[read1])
            read2_length = sum(fingerprint_list[read2])

            #min_region_kmer_coverage = 0.30

            min_cov_number = int(min_region_kmer_coverage * min(l1,l2)
/ min_total_length)
            min_cov_number = min(min_cov_number, 15)
            #print((read1, read2, min_cov_number,
min_sharing_dict[(read1,read2)], min_cov_number, min_region_length))
            #print(abs(l1-l2) <= max_diff_region_percentage *
max(l1,l2))
            #print(l1, ' ', l2)
            #print('here ', abs(l1-l2))
            if min_sharing_dict[(read1,read2)] >= min_cov_number and
(abs(l1-l2) <= max_diff_region_percentage * max(l1,l2) and max(l1,l2)
>= min_region_length):
                min_up = min(up1,up2)
                start_ov1 = up1 - min_up
                start_ov2 = up2 - min_up
                min_down = min(read1_length-(up1+l1), read2_length-
(up2+l2))
                end_ov1 = up1 + l1 + min_down
                end_ov2 = up2 + l2 + min_down

                ov_length = min(end_ov1-start_ov1, end_ov2-start_ov2)
                #print((min(l1,l2)/ov_length, min(l1,l2), ov_length))
                if min(l1,l2) >= min_overlap_coverage * ov_length and
ov_length >= min_overlap_length:
                    #print('OK ', (read1, read2))
                    value = overlap_dict.get((read_ids[read1][:-2],
read_ids[read2][:-2]), [])
                    if value == [] or ov_length > value[-1]:
                        value = [int(read_ids[read1][-1]),
int(read_ids[read2][-1]), read1_length, read2_length, start_ov1,
end_ov1, start_ov2, end_ov2, ov_length]
                        overlap_dict[(read_ids[read1][:-2],
read_ids[read2][:-2])] = value
```

```
end_time = time.perf_counter()
print('Computing the overlaps: ', end_time-start_time)

overlap_list = []

for (read1, read2) in overlap_dict:
    (flag1, flag2, length1, length2, start1, end1, start2, end2,
ov_length) = overlap_dict[(read1, read2)]
    strand = 0
    if flag1 == 1:
        if flag2 == 0:
            (start2, end2) = (length2 - end2, length2 - start2)
            strand = 1
    else:
        (start1, end1) = (length1 - end1, length1 - start1)
        if flag2 == 0:
            (start2, end2) = (length2 - end2, length2 - start2)
        else:
            strand = 1

    overlap_list.append((read1, read2, length1, length2, start1, end1,
start2, end2, strand))
    #print('\t'.join(map(str, [read1, read2, length1, length2, start1,
end1, start2, end2, strand])))

end_time = time.perf_counter()
print('Total: ', end_time-begin_time)
```

Numero di overlaps in output:

```
len(overlap_list)
```

Controlla se qualche read non è stato allineato.

```
read_number = len(read_ids) // 2

check_set = set(range(read_number))

cfr_set = set()

for t in overlap_dict:
    cfr_set.add(t[0])
    cfr_set.add(t[1])

cfr_set.difference(cfr_set)
```

Controlla i risultati

```
import Bio
from Bio.Seq import Seq

with open('./sampled_read-noerr.fasta', 'r') as in_sequence:
    file_seq = in_sequence.read()

file_seq = re.findall(r'[^>]+', file_seq)
file_seq = [re.findall(r'.+', seq) for seq in file_seq]
read_seq_list = [seq[1] for seq in file_seq]

print_first = -1
which_strand = [0,1]

max_diff_length = -1
min_ov_length = -1

count = 0
min_count = 0

for r in overlap_list:
    read1 = int(r[0])
    read2 = int(r[1])
    (start1, end1, start2, end2, strand) = (r[4], r[5], r[6], r[7],
r[8])
    if strand in which_strand:
        seq1 = read_seq_list[read1][start1:end1]
        seq2 = read_seq_list[read2][start2:end2]
        if (seq1 != seq2):
            print(read1, ' ', read2, ' diff ', min(len(seq1),
len(seq2)))
            if max_diff_length == -1 or min(len(seq1), len(seq2)) >
max_diff_length:
                max_diff_length = min(len(seq1), len(seq2))
            else:
                if min_ov_length == -1 or min(len(seq1), len(seq2)) <
min_ov_length:
                    min_ov_length = min(len(seq1), len(seq2))

                if min(len(seq1), len(seq2)) <= min_overlap_length:
                    min_count = min_count + 1

        if strand == 1:
            seq2 = str(Seq(seq2).reverse_complement())

        chunk1 = re.findall(r'(.{,60})', seq1)
        chunk2 = re.findall(r'(.{,60})', seq2)

        if read1 == -1 and read2 == -1:
            print(r)
```

```
            for (i, c) in enumerate(chunk1):
                print(c)
                print(chunk2[i])
                print()

        count = count + 1

    if count == print_first:
        break
```