```python
# The fasta_data class which holds and processes DNA sequences in multi-FASTA format.
class fasta_data:
    # Initializer that takes a file name, reads the file and organizes the data.
    def   init  (self, file name):
        # Reads the entire file content
        self.f = open(file_name).read()
        self.records = {}
        self.id = ""
        # Processes each line of the file content
        for line in self.f.splitlines():
            # If the line is a header line (starts with ">")
            if line.startswith(">"):
                # If there was a previous sequence, save it to the records
                if self.id != "":
                    self.records[self.id] = self.seq
                # Extracts the identifier from the header line and resets the sequence
                self.id = line.split(" ")[0][1:]
                self.seq = ""
            # If the line is a sequence line, append it to the current sequence
            else:
                self.seq += line.strip()

    # Getter methods for accessing basic data
    def get_records(self):
        return self.records

    def get_id(self):
        return self.records.keys()

    def get_seq(self):
        return self.records.values()

    # Task 1: How many records are in the file?
    def reads_number(self):
        return len(self.records)

    # Task 2: What are the lengths of the sequences in the file?
    def get_lengths(self):
        length = {}
        # Calculate the length for each sequence
        for id, seq in self.records.items():
            length[id] = len(seq)
        # Find the shortest and longest sequences and their identifiers
        min_length = min(length.values())
        max_length = max(length.values())
        min_ids = [id for id, seq_len in length.items() if seq_len == min_length]
        max_ids = [id for id, seq_len in length.items() if seq_len == max_length]
        return length, min_length, max_length, min_ids, max_ids

    # Task 3: Identify all ORFs in each sequence and find the longest one
    def reading_frames(self, frame):
        start_codon = 'ATG'
        stop_codons = ['TAA', 'TAG', 'TGA']
        orfs = {}
        # Process each sequence
        for id, sequence in self.records.items():
            frame_seq = sequence[frame-1:]
            orfs[id] = []
            # Check each codon in the sequence
            for i in range(0, len(frame_seq), 3):
                # If the codon is a start codon
                if frame_seq[i:i+3] == start_codon:
                    # Check the rest codons in the sequence
```

```
                    for j in range(i+3, len(frame_seq), 3):
                        # If the codon is a stop codon
                        if frame_seq[j:j+3] in stop_codons:
                            # Record the ORF
                            orfs[id].append((i+1, j+3, frame_seq[i:j+3]))
                            break
        # Find the longest ORF
        longest_orf = max((tuple(list(orf) + [id]) for id, orfs_list in orfs.items() for
orf in orfs_list), key=lambda x: x[1] - x[0])
        return orfs, longest_orf

    # Task 4: Identify all repeats of length n in all sequences
    def repeats(self, n):
        repeats = {}
        # Process each sequence
        for sequence in self.records.values():
            # Check each possible repeat in the sequence
            for i in range(len(sequence) - n + 1):
                # If the repeat is already recorded
                if sequence[i:i+n] in repeats:
                    # Increase the count
                    repeats[sequence[i:i+n]] += 1
                else:
                    # Record the repeat
                    repeats[sequence[i:i+n]] = 1
        # Find the most common repeat
        most_common = max(repeats.items(), key=lambda x: x[1])
        return repeats, most_common

# Create an instance of the fasta_data class with the file "dna.example.fasta"
reads = fasta_data("dna.example.fasta")
records = reads.get_records()

# Part 1: Print out the number of records
reads_number = reads.reads_number()
print("number of records: " + str(reads_number))

# Part 2: Print out the minimum and maximum lengths of sequences and their identifiers
records_lengths = reads.get_lengths()
print("minimum length: " + str(records_lengths[1]))
print("maximum length: " + str(records_lengths[2]))
print("minimum ids: " + str(records_lengths[3]))
print("maximum ids: " + str(records_lengths[4]))

# Part 3: Print out the longest ORFs in each frame and their starting positions,
# and the identifier of the sequence containing the longest ORF in frame 1
orfs1, longest_orf1 = reads.reading_frames(1)
orfs2, longest_orf2 = reads.reading_frames(2)
orfs3, longest_orf3 = reads.reading_frames(3)
print("longest ORF in frame 1: " + str(longest_orf1[:2]))
print("longest ORF in frame 2: " + str(longest_orf2[:2]))
print("longest ORF in frame 3: " + str(longest_orf3[:2]))
print("longest ORF in frame 1 id: " + str(longest_orf1[-1]))

# Part 4: Print out the most common repeats of lengths ranging from 5 to 20 in the
sequences
for i in range(5,20):
    repeat, most_common = reads.repeats(i)
    print(str(i) + " most common: " + str(most_common))



final_reads = fasta_data("dna2.fasta")
```

```
orfs1, longest_orf1 = final_reads.reading_frames(1)
print("longest ORF in frame 1: " + str(longest_orf1[1]-longest_orf1[0] + 1))
orfs2, longest_orf2 = final_reads.reading_frames(2)
print("longest ORF in frame 2: " + str(longest orf2[1]-longest orf2[0] + 1))
orfs3, longest_orf3 = final_reads.reading_frames(3)
print("longest ORF in frame 3: " + str(longest_orf3[:2]))
print("longest ORF in frame 3: " + str(longest_orf3[1]-longest_orf3[0] + 1))
lengths = []
lengths.append(max( (repeats[1]-repeats[0] + 1 for repeats in
orfs1["gi|142022655|gb|EQ086233.1|16"])))
lengths.append(max( (repeats[1]-repeats[0] + 1 for repeats in
orfs2["gi|142022655|gb|EQ086233.1|16"])))
lengths.append(max( (repeats[1]-repeats[0] + 1 for repeats in
orfs3["gi|142022655|gb|EQ086233.1|16"])))

orfs4, longest_orf4 = final_reads.reading_frames(2)
print("longest ORF in frame 4: " + str(longest_orf4[:2]))
print(lengths )

repeat, most_common = final_reads.repeats(12)
seqs = []
for seq, reap in repeat.items():
    if reap == most common[1]:
        seqs.append(seq)
print(seqs)
print(len(seqs))

repeat, most_common = final_reads.repeats(7)
print(most_common)
```

Resultado de ejecución con archivos suministrados:

“number of records: 24
minimum length: 512
maximum length: 4805
minimum ids: ['gi|142022655|gb|EQ086233.1|521']
maximum ids: ['gi|142022655|gb|EQ086233.1|323']
longest ORF in frame 1: (2824, 4509)
longest ORF in frame 2: (64, 1434)
longest ORF in frame 3: (139, 1746)
longest ORF in frame 1 id: gi|142022655|gb|EQ086233.1|323
5 most common: ('GCGCG', 434)
6 most common: ('CGGCGC', 158)
7 most common: ('GCGCGGC', 74)
8 most common: ('GCGCGGCG', 27)
9 most common: ('CGCGGCGCG', 13)
10 most common: ('GCGCGGCGCG', 8)
11 most common: ('CGCTGCGCGGC', 4)
12 most common: ('CGCGGTCGAGCG', 3)
13 most common: ('GATCGTCGACGAC', 3)
14 most common: ('GTCGCGGTCGAGCG', 2)
15 most common: ('GTCGCGGTCGAGCGG', 2)
16 most common: ('CTTGCTCGCCGCGCCC', 2)
17 most common: ('GCGGCGTCCGGCGCGTC', 2)
18 most common: ('ACCGTGCCGGCCGGCAAC', 2)
19 most common: ('GACCAGCGCGAACGCGCCG', 2)
longest ORF in frame 1: 2394

longest ORF in frame 2: 1458
longest ORF in frame 3: (1438, 3081)
longest ORF in frame 3: 1644
longest ORF in frame 4: (3070, 4527)
[1509, 1458, 1644]
['CATTCGCCATTC', 'ATTCGCCATTCG', 'TTCGCCATTCGC', 'TCGCCATTCGCC']
4
('CGCGCCG', 59)"

1. Clase `fasta_data`

Es una clase que procesa datos de ADN en formato multi-FASTA, con métodos para realizar análisis sobre las secuencias.

a) Inicializador (`__init__`)

```
def __init__(self, file_name):
    # Reads the entire file content
    self.f = open(file_name).read()
    self.records = {}
    self.id = ""
    # Processes each line of the file content
    for line in self.f.splitlines():
        # If the line is a header line (starts with ">")
        if line.startswith(">"):
            # If there was a previous sequence, save it to the records
            if self.id != "":
                self.records[self.id] = self.seq
            # Extracts the identifier from the header line and resets the sequence
            self.id = line.split(" ")[0][1:]
            self.seq = ""
        # If the line is a sequence line, append it to the current sequence
        else:
            self.seq += line.strip()
```

¿Qué hace?

- Abre un archivo FASTA (`file_name`) y lo lee por completo.
- Organiza las secuencias en un diccionario (`self.records`), donde:
    - Las claves son los identificadores de las secuencias (líneas que empiezan con `>`).
    - Los valores son las secuencias asociadas.

b) Métodos básicos

- **`get_records`**: Devuelve todas las secuencias en forma de diccionario.
- **`get_id`**: Devuelve los identificadores de las secuencias.
- **`get_seq`**: Devuelve las secuencias como una lista de valores.

c) Métodos funcionales

1. Número de registros (`reads_number`)

```
def reads_number(self):
    return len(self.records)
```

Devuelve el número de secuencias en el archivo.

2. Longitudes de secuencias (`get_lengths`)

```
def get_lengths(self):
    length = {}
    # Calculate the length for each sequence
    for id, seq in self.records.items():
        length[id] = len(seq)
    # Find the shortest and longest sequences and their identifiers
    min_length = min(length.values())
    max_length = max(length.values())
    min_ids = [id for id, seq_len in length.items() if seq_len == min_length]
    max_ids = [id for id, seq_len in length.items() if seq_len == max_length]
    return length, min_length, max_length, min_ids, max_ids
```

¿Qué hace?

1. Calcula la longitud de cada secuencia y las guarda en el diccionario `length`.
2. Encuentra:
    - La longitud mínima y máxima de las secuencias.
    - Los identificadores asociados a las secuencias más cortas y más largas.

3. Marcos de lectura abiertos (ORFs) (`reading_frames`)

```
def reading_frames(self, frame):
    start_codon = 'ATG'
    stop_codons = ['TAA', 'TAG', 'TGA']
    orfs = {}
    # Process each sequence
    for id, sequence in self.records.items():
        frame_seq = sequence[frame-1:]
        orfs[id] = []
        # Check each codon in the sequence
        for i in range(0, len(frame_seq), 3):
            # If the codon is a start codon
            if frame_seq[i:i+3] == start_codon:
                # Check the rest codons in the sequence
                for j in range(i+3, len(frame_seq), 3):
                    # If the codon is a stop codon
                    if frame_seq[j:j+3] in stop_codons:
                        # Record the ORF
                        orfs[id].append((i+1, j+3, frame_seq[i:j+3]))
                        break
    # Find the longest ORF
    longest_orf = max((tuple(list(orf) + [id]) for id, orfs_list in orfs.items() for orf
in orfs_list), key=lambda x: x[1] - x[0])
    return orfs, longest_orf
```

¿Qué hace?

1. Analiza las secuencias para encontrar marcos de lectura abiertos (ORFs):
    - Empiezan con un codón de inicio (`ATG`).
    - Terminan con un codón de parada (`TAA`, `TAG`, `TGA`).
2. Identifica los ORFs para un marco de lectura específico (`frame`).
3. Devuelve:
    - Todos los ORFs encontrados.
    - El ORF más largo.

4. Repeticiones (`repeats`)

```
def repeats(self, n):
    repeats = {}
    # Process each sequence
    for sequence in self.records.values():
        # Check each possible repeat in the sequence
        for i in range(len(sequence) - n + 1):
            # If the repeat is already recorded
            if sequence[i:i+n] in repeats:
                # Increase the count
                repeats[sequence[i:i+n]] += 1
            else:
                # Record the repeat
                repeats[sequence[i:i+n]] = 1
    # Find the most common repeat
    most_common = max(repeats.items(), key=lambda x: x[1])
    return repeats, most_common
```

¿Qué hace?

1. Encuentra todas las subsecuencias de longitud `n` en las secuencias.
2. Cuenta la frecuencia de cada subsecuencia.
3. Identifica la subsecuencia más frecuente.

2. Análisis Principal

a) Procesamiento inicial

```
reads = fasta_data("dna.example.fasta")
records = reads.get_records()
print("number of records: " + str(reads_number))
```

¿Qué hace?

- Carga el archivo `dna.example.fasta`.
- Imprime el número total de registros.

b) Longitudes de secuencias

```
records_lengths = reads.get_lengths()
print("minimum length: " + str(records_lengths[1]))
print("maximum length: " + str(records_lengths[2]))
print("minimum ids: " + str(records_lengths[3]))
print("maximum ids: " + str(records_lengths[4]))
```

¿Qué hace?

- Calcula y muestra:
    - La longitud mínima y máxima.
    - Los identificadores asociados a las secuencias más cortas y largas.

c) ORFs por marcos de lectura

```
orfs1, longest_orf1 = reads.reading_frames(1)
print("longest ORF in frame 1: " + str(longest_orf1[:2]))
```

¿Qué hace?

- Identifica los ORFs en el marco 1.
- Imprime el ORF más largo y su posición.

d) Repeticiones comunes

```
for i in range(5,20):
    repeat, most_common = reads.repeats(i)
    print(str(i) + " most common: " + str(most_common))
```

¿Qué hace?

- Busca repeticiones de longitud entre 5 y 20.
- Muestra la repetición más frecuente para cada longitud.

e) Análisis en un segundo archivo

```
final_reads = fasta_data("dna2.fasta")
orfs1, longest_orf1 = final_reads.reading_frames(1)
print("longest ORF in frame 1: " + str(longest_orf1[1]-longest_orf1[0] + 1))
```

¿Qué hace?

- Realiza los mismos análisis, pero sobre un nuevo archivo (`dna2.fasta`).

3. Conclusión

Este código:

- Carga y organiza datos de ADN en formato FASTA.
- Realiza análisis básicos como el conteo de secuencias y longitudes.
- Identifica ORFs y repeticiones comunes.
- Ofrece flexibilidad para trabajar con múltiples archivos FASTA.