Input format personalizado:

1. Extender FileInputFormat<KeyType,ValueType>:
   - KeyType debe implementar WritableComparable y
   - ValueType debe implementar Writable
2. Sobreescribir el método createRecordReader.

- Líneas: [dia] [mes] [año] [temp. mínima] [temp. máxima]

```
public class Temp1InputFormat extends FileInputFormat<CustomDate,CityTemperature> {

    @Override
    public RecordReader createRecordReader(InputSplit inputSplit,TaskAttemptContext context)
      throws IOException, InterruptedException {

      Temp1RecordReader temp1RecordReader = new Temp1RecordReader();
      temp1RecordReader.initialize(inputSplit, context);
      return temp1RecordReader;
    }
}
```

El InputFormat personalizado utilizará un RecordReader personalizado que se encargará de transferir el archivo en HDFS a registros en formato de pares (clave, valor).
El procedimiento a realizar sobre el RecordReader personalizado es:

initialize(split, context)
while (getProgress() !=1) {
nextKeyValue()
getCurrentKey(), getCurrentValue()
}
close()

```
public class Temp1RecordReader<CustomDate,CityTemperature> extends RecordReader {
  LineRecordReader lineRecordReader; // divide ('split') archivos línea por línea
  CustomDate key; CityTempurature value;

   @Override
  public void initialize(InputSplit inputSplit, TaskAttemptContext context) throws IOException,
  InterruptedException {
    lineRecordReader = new LineRecordReader();
    lineRecordReader.initialize(inputSplit, context);
  }
```

```
// método que transforma una línea en un par (clave, valor)
@Override
public boolean nextKeyValue() throws IOException, InterruptedException {
    if (!lineRecordReader.nextKeyValue()) {
        return false;
    }
    String line = lineRecordReader.getCurrentValue().toString();
    String[] keyValue = line.split("\t");
    String[] keyFields = keyValue[0].split(" ");
    valueFields = keyValue[1].split(" ");

   key = new CustomDate(keyFields[0],keyFields[1],keyFields[2]);
   value = new CityTempurature(valueFields[0],valueFields[1]);
   return true;
}

@Override
public CustomDate getCurrentKey() throws IOException, InterruptedException {
    return key;
}
```

```
    @Override
    public CityTempurature getCurrentValue() throws IOException, InterruptedException {
        return value;
    }

    @Override
    public float getProgress() throws IOException, InterruptedException {
        return lineRecordReader.getProgress();
    }

    @Override
    public void close() throws IOException {
        lineRecordReader.close();
    }
}
```

Utilizando el nuevo InputFormat/RecordReader el mapper no necesita parsear los datos, puede obtener clave y valor sin analizar la línea.

```
public class TempuratureMapper extends Mapper<CustomDate,CityTemperature,
                                                CustomDate,IntWritable>{


    @Override
    protected void map(CustomDate key, CityTempurature value, Mapper.Context
                                context) {
       context.write(key,value.getTemperature());
    }
}
```

El reducer calcula la media de temperaturas

```
public class TempuratureReducer extends
Reducer<CustomDate,IntWritable,CustomDate,IntWritable>{

    @Override
    protected void reduce(CustomDate key, Iterable<IntWritable> values, Reducer.Context
                          context)  throws IOException, InterruptedException {

        int sum = 0;
        int nbr =0 ;
        for (IntWritable value : values) {
           nbr++;
           sum=sum+value.get();
        }

        context.write(key, new IntWritable(sum/nbr));
    }
}
```