For cipher text 1,

1. Considering the name ”split-julius.txt” and there are many repeated character patterns in the cipher text, so I guess it’s likely to be mono alphabetic substitution cipher, which means one character in cipher text maps to exactly one character in plaintext.
2. Then as the regular practice and guidance of decrypting substitution cipher, I wrote some codes in Jupiter NoteBook to get the letter frequency in cipher text and try to compare these frequency datas with the the frequency of the letters of the alphabet in English. In the following code, s1 is the cipher text and res1 is the sorted frequency datas after running. The high frequency is mapped kind of well.

```
num = len(s1)
for abc in 'abcdefghijklmnopqrstuvwxyz':
number = s1.count(abc)
print(abc,':',number, (number / num)*100)

res1 = sorted(res.items(), key=lambda x: x[1])  // sort the frequency result
print(res1)
```

3. Find there are some repeated blocks like “vry”, and find a table of most common 3 letters words and try all these possibilities. Make a final most likely guess which is “vry” maps to “YOU”. For some specific pattern like “vyrjbjbq” in the first line in cipher text, it looks like “YOUDIDIT”, which makes a lot of sense here.
4. Python function String.replace( ) in the Jupiter NoteBook allows me to see the intermediate results quickly, which is a good tool for me to do instant analysis for the cipher text.
5. After try amount of times of letter substitution, more and more “words” emerge. At final, it’s cracked. The following is the final version of replace code, which receives the correct matching of letters in cipher and plaintext.

```
s1 = s1.replace("v", "Y")
s1 = s1.replace("y", "O")
s1 = s1.replace("r", “U")

s1 = s1.replace("k","E")
s1 = s1.replace("q", "T")
s1 = s1.replace("b","I")
s1 = s1.replace("p","A")
s1 = s1.replace("j","D")

s1= s1.replace("x","N")
s1= s1.replace("o","R")
s1= s1.replace("h","B")
s1= s1.replace("l","F")

s1= s1.replace("A","p")
s1= s1.replace("g","A")
s1= s1.replace("e","L")
s1= s1.replace("m","G")

s1= s1.replace("i","C")
s1= s1.replace("f","M")
s1= s1.replace("z","P")
s1= s1.replace("p","S")
s1= s1.replace("s","V")
s1= s1.replace("a","H")
```

```
s1= s1.replace("t","W")
s1= s1.replace("d","K")
s1= s1.replace("n","Q")
s1= s1.replace("u","X")
print(s1, end="\n")
```

For cipher text 2 with file name ”alternating.txt”, after decrypting the first cipher, strongly suspect it’s Vigenere cipher.

1. As known, the Vigenere cipher is really just k different Caesar ciphers, where k is the length of the key. So to break this type of cipher, I process in two steps: first figure out the k, and then break each Caesar cipher.
2. There are some letter blocks appearing multiple times in the cipher text, which might correspond to short common words in plaintext like ”the”. Based on Kasiski examination, the distance between these repeated blocks of cipher text is typically a multiple of the keyword length. For instance, “mlx” in the first line of cipher appears twice and the distance between them is 14, whose factors are 2 and 7. “rsn” in the second line appears twice and the distance is 22, whose factors are 2 and 11. So the only common factor is 2, indicating the keyword length should be 2. In terms of the file name “alternating”, which implying the length should be 2 and might swap at some points. the it confirms the conjecture.
3. Now I have the key length which is 2, so what Im supposed to do next is to divide the cipher into two arrays. Each of them is a Caesar cipher with the same shift number. Before doing splitting process, spaces between words need to be eliminated firstly. And for my convenience, I think it’s not necessary to process whole cipher. Instead I could just pick up the first sentence and decrypt it, figuring out the shift number for each Caesar cipher.
4. For decryption part, dealing with Index of Coincidence, I firstly decompose cipher text into 2 groups. Each group is decrypted with letter frequency analysis. Firstly Count the frequency of each letter of the sub cipher text segment, check the probability table of the frequency distribution of letters, Calculate the length of the subciphertext segment, then calculate index. Then the subciphertext segment is shifted 25 times with values of index1 to index 25 being found according to the above method. According to the definition of the index of coincident: a meaningful English text, index ≈ 0.065, so find the shift of index value close to 0.065, which is the corresponding letter in the key.
5. After running the code, find out the key is ET. And the plainText is perfectly translated.It works.

```
2 Caeser cipher should be:
RGAVWIXSWSTIIRCYEWEXLJRPXTJLLQAV
BXHDMIPBVFEMTWHVGMKMXBTLXHMXHXHD

the key should be: ET

The palin Text should be:
niceworksteptwoiscompleteandyoucanstartthefinalstepofthehomework
```

6. Then I check the results using online decryption tools. It shows the result is great. And the I Just repeat and decrypt the remaining cipher parts. The key ”ET” needs to be alternated sometimes.

The code is as shown:

```
public static void main(String[] args) {
    String s = "rbgxahvdwmiixpsbwvsfteimitrwchyvegwmekxmlxjbrtplxxthjmlxlhqxahvd";
    s = s.toUpperCase();
    Test vigenere = new Test();
```

```
    vigenere.decryptCipher(2,s);
  }

        // use Index of Coincidence to figure out key
public void decryptCipher(int keyLength, String ciphertext) {
    int[] key = new int[keyLength];
    ArrayList<String> cipherGroup = new ArrayList<>();

    double[] probability = new double[]{0.082, 0.015, 0.028, 0.043, 0.127, 0.022, 0.02, 0.061, 0.07,
0.002, 0.008, 0.04, 0.024, 0.067, 0.075, 0.019, 0.001, 0.06, 0.063, 0.091, 0.028, 0.01, 0.023,
0.001, 0.02, 0.001};

    // split by keyword length
    for (int i = 0; i < keyLength; ++i) {
       StringBuilder temporaryGroup = new StringBuilder();
       for (int j = 0; i + j * keyLength < ciphertext.length(); ++j) {
          temporaryGroup.append(ciphertext.charAt(i + j * keyLength));
       }
       cipherGroup.add(temporaryGroup.toString());
    }
    System.out.println("\n2 Caeser cipher should be: ");
    for (String s : cipherGroup) {
        System.out.println(s);
    }

    // confirm key
    for (int i = 0; i < keyLength; ++i) {
       double MG; //  Index of Coincidence
       int flag; // moving distance
       int g = 0; //  cipher moves g distance
       HashMap<Character, Integer> occurrenceNumber; // occurrence Number of letter
       String subCipher;

       while (true && g < 26) {
          MG = 0;
          flag = 65 + g;
          subCipher = cipherGroup.get(i);
          occurrenceNumber = new HashMap<>();

          //  initialize letter and counter
          for (int h = 0; h < 26; ++h) {
             occurrenceNumber.put((char) (h + 65), 0);
          }

          // count occurrence number of letters
          for (int j = 0; j < subCipher.length(); ++j) {
             occurrenceNumber.put(subCipher.charAt(j), occurrenceNumber.get(subCipher.charAt(j))
+ 1);
          }

          //  calculate Index of Coincidence
          for (int k = 0; k < 26; ++k, ++flag) {
             double p = probability[k];
             flag = (flag == 91) ? 65 : flag;

             double f = (double) occurrenceNumber.get((char) flag) / subCipher.length();
             MG += p * f;
          }
```

```
            // check condition of quitting loop
            if (MG >= 0.055) {
                key[i] = g;
                break;
            } else {
                ++g;
            }
        }
    }

    // print out key
    StringBuilder keyString = new StringBuilder();
    for (int i = 0; i < keyLength; ++i) {
        keyString.append((char) (key[i] + 65));
    }
    System.out.println("\nthe key should be: " + keyString.toString());

    // decrypt
    StringBuilder plainBuffer = new StringBuilder();
    for (int i = 0; i < ciphertext.length(); ++i) {
        int keyFlag = i % keyLength;
        int change = (int) ciphertext.charAt(i) - 65 - key[keyFlag];
        char plainLetter = (char) ((change < 0 ? (change + 26) : change) + 65);
        plainBuffer.append(plainLetter);
    }
    System.out.println("\nThe palin Text should be：\n" + plainBuffer.toString().toLowerCase());
}
```