Given a nucleotide p, we denote its complementary nucleotide as $\overline{p}$. The **reverse complement** o stringPattern = $p_1 \ldots p_n$ is the string $\overline{Pattern} = \overline{p}_n \ldots \overline{p}_1$ formed by taking the complement of each nucleo in*Pattern*, then reversing the resulting string. We will need the solution to the following problem throughout book:

Reverse Complement Problem: *Reverse complement a nucleotide pattern.*

Input: A DNA string *Pattern*.

Output: $\overline{Pattern}$, the reverse complement of *Pattern*.

CODE CHALLENGE: Solve the Reverse Complement Problem.

Sample Input:

```
AAAACCCGGT
```

Sample Output:

```
ACCGGGTTTT
```

Extra Dataset

CODE CHALLENGE: Solve the Pattern Matching Problem (restated below).

Pattern Matching Problem: *Find all occurrences of a pattern in a string.*

Input: Two strings, *Pattern* and *Genome*.

Output: All starting positions where *Pattern* appears as a substring of *Genome*.

Note: Throughout this chapter, we will use **0-based indexing** in problem implementations, meaning that count starting at 0 instead of 1. For example, the starting positions of `ATA` in `CGATATATCCATAG` are 2, 4, and instead of 3, 5, and 11.

Sample Input:

```
ATAT
GATATATGCATATACTT
```

Sample Output:

```
1 3 9
```

CODE CHALLENGE: Solve the Clump Finding Problem (restated below). You will need to make sure that your algorithm is efficient enough to handle a large dataset.

Clump Finding Problem: *Find patterns forming clumps in a string.*

Input: A string *Genome*, and integers *k*, *L*, and *t*.

Output: All distinct *k*-mers forming (*L*, *t*)-clumps in *Genome*.

Sample Input:

```
CGGACTCGACAGATGTGAAGAACGACAATGTGAAGACTCGACACGACAGAGTGAAGAGAAGAGGAAACATTGTAA
5 50 4
```

Sample Output:

```
CGACA GAAGA
```

Extra Dataset

Let's follow the 5' → 3' direction of DNA and walk along the chromosome from *terC* to *oriC* (along a reverse h strand) and continue on from *oriC* to *terC* (along a forward half-strand). In our previous discussion, we saw the skew is decreasing along the reverse half-strand and increasing along the forward half-strand. Thus, skew should achieve a minimum at the position where the reverse half-strand ends and the forward half-str begins, which is exactly the location of *oriC*!

We have just developed an insight for a new algorithm for locating *oriC*: it should be found where the s attains a minimum:

Minimum Skew Problem: *Find a position in a genome minimizing the skew.*

Input: A DNA string *Genome*.

Output: All integer(s) *i* minimizing $Skew(Prefix_i(Text))$ among all values of *i* (from 0 to |*Genome*|).

CODE CHALLENGE: Solve the Minimum Skew Problem.

Sample Input:

```
TAAAGACTGCCGAGAGGCCAACACGAGTGCTAGAACGAGGGGCGTAAACGCGGGTCCGAT
```

Sample Output:

```
11 24
```

We say that position i in k-mers $p_1 \ldots p_k$ and $q_1 \ldots q_k$ is a **mismatch** if $p_i \neq q_i$. example, CGAAT andCGGAC have two mismatches. Our observation that a *DnaA* box may appear with s variations leads to the following generalization of the Pattern Matching Problem:

Approximate Pattern Matching Problem: *Find all approximate occurrences of a pattern in a string.*

Input: Two strings *Pattern* and *Text* along with an integer *d*.

Output: All positions where *Pattern* appears in *Text* with at most *d* mismatches.

CODE CHALLENGE: Solve the Approximate Pattern Matching Problem

Sample Input:

```
ATTCTGGA
CGCCCGAATCCAGAACGCATTCCCATATTTCGGGACCACTGGCCTCCACGGTACGGACGTCAATCAAAT
3
```

Sample Output:

```
6 7 26 27
```

Our goal is to modify our previous algorithm to find *DnaA* boxes by identifying frequent *k*-mers, possibly mismatches. Given strings *Text* and *Pattern* as well as an integer *d*, we define $Count_d(Text, Pattern)$ as the number of occurrences of *Pattern* in *Text* with at most *d* mismatches. example, $Count_1$(AACAAGCTGATAAACATTTAAAGAG, AAAAA) = 4 because AAAAA appears four times in this st with at most one mismatch: AACAA, ATAAA, AAACA, and AAAGA. Note that two of these occurrences overlap.

A **most frequent *k*-mer with up to *d* mismatches** in *Text* is simply string *Pattern* maximizing $Count_d(Text, Pattern)$ among all *k*-mers. Note that *Pattern* does not need to actu appear as a substring of *Text*; for example, as we saw above, AAAAA is the most frequent 5-mer with 1 mism inAACAAGCTGATAAACATTTAAAGAG, even though it does not appear in this string. Keep this in mind while sol the following problem:

Frequent Words with Mismatches Problem: *Find the most frequent k-mers with mismatches in a string*.

Input: A string *Text* as well as integers *k* and *d*. (You may assume $k \leq 12$ and $d \leq 3$.)

Output: All most frequent *k*-mers with up to *d* mismatches in *Text*.

CODE CHALLENGE: Solve the Frequent Words with Mismatches Problem.

Sample Input:

```
ACGTTGCATGTCGCATGATGCATGAGAGCT 4 1
```

Sample Output:

```
GATG ATGC ATGT
```

We now redefine the Frequent Words Problem to account for both mismatches and reverse complements. Re
that $\overline{Pattern}$ refers to the reverse complement of *Pattern*.

Frequent Words with Mismatches and Reverse Complements Problem: *Find the most frequent k-mers (*
mismatches and reverse complements) in a DNA string.

Input: A DNA string *Text* as well as integers *k* and *d*.

Output: All *k*-mers *Pattern* maximizing the sum $Count_d(Text, Pattern) + Count_d(Text, \overline{Pattern})$ over all possible *k*-mers.

CODE CHALLENGE: Solve the Frequent Words with Mismatches and Reverse Complements Problem.

Sample Input:

```
ACGTTGCATGTCGCATGATGCATGAGAGCT
4 1
```

Sample Output:

```
ATGT ACAT
```

We now have a rigorously defined computational problem:

Frequent Words Problem: *Find the most frequent k-mers in a string.*

Input: A string *Text* and an integer *k*.

Output: All most frequent *k*-mers in *Text*.

CODE CHALLENGE: Solve the Frequent Words Problem. Some notes on how code challenges work:

1. When you click “Download Dataset”, you will receive a randomized dataset. Run your program on this data and then return your answer in the text field below.
2. You can see how you should format your answer by looking at the sample output and extra dataset.
3. Every time you click “Try Again”, you will need to download a *new* dataset.
4. At this point in the book, you may implement any practical algorithm (rather than the most efficient algorithm for solving the Frequent Words Problem that will work for |*Text*| < 1000 and *k* < 15. Later in the book, we will show you the most efficient algorithm for solving this problem.

Sample Input:

```
ACGTTGCATGTCGCATGATGCATGAGAGCT
4
```

Sample Output:

```
CATG GCAT
```