# Biology Meets Programming

Bioinformatics for Begginers

Amin Kasrou Aouam

2020-10-10

# Contents

# Biology Meets Programming: Bioinformatics for Beginners

## Week 1

### DNA replication

1. Origin of replication (ori)

   Locating an ori is key for gene therapy (e.g. viral vectors), to introduce a theraupetic gene.

2. Computational approaches to find ori in Vibrio Cholerae

   a) Exercise: find Pattern

   We'll look for the **DnaA box** sequence, using a sliding window, in that case we will use this function to find out how many times a sequence appears in the genome:

```python
def PatternCount(Text, Pattern):
    count = 0
    for i in range(len(Text)-len(Pattern)+1):
        if Text[i:i+len(Pattern)] == Pattern:
            count = count+1
    return count
```

   For the second part, we're going to calculate the frequency map of the sequences of length *k*:

```python
def FrequentWords(Text, k):
    words = []
    freq = FrequencyMap(Text, k)
    m = max(freq.values())
    for key in freq:
        if freq[key] == m:
            words.append(key)
    return words


def FrequencyMap(Text, k):
    freq = {}
    n = len(Text)
    for i in range(n - k + 1):
        Pattern = Text[i:i + k]
        freq[Pattern] = 0
    for i in range(n - k + 1):
        Pattern = Text[i:i + k]
        freq[Pattern] += 1
    return freq
```

b) Exercise: Find the reverse complement of a sequence

We're going to generate the reverse complement of a sequence, which is the complement of a sequence, read in the same direction (5' -> 3').

```
def ReverseComplement(Pattern):
    Pattern = Reverse(Pattern)
    Pattern = Complement(Pattern)
    return Pattern


def Reverse(Pattern):
    reversed = Pattern[::-1]
    return reversed


def Complement(Pattern):
    compl = ""
    complement_letters = {"A": "T", "T": "A", "C": "G", "G": "
        C"}
    for char in Pattern:
        compl += complement_letters[char]
    return compl
```

After using our function on the *Vibrio Cholerae's* genome, we realize that some of the frequent *k-mers* are reverse complements of other frequent ones.

c) Exercise: Find a subsequence within a sequence

We're going to find the ocurrences of a subsquence inside a sequence, and save the index of the first letter in the sequence.

```
def PatternMatching(Pattern, Genome):
    positions = []
    for i in range(len(Genome)-len(Pattern)+1):
        if Genome[i:i+len(Pattern)] == Pattern:
            positions.append(i)
    return positions
```

We find out that the *9-mers* with the highest frequency appear in cluster. There is strong statistical evidence that our subsequences are *DnaA boxes*.

3. Computational approaches to find ori in any bacteria

Now that we're pretty confident about the *DnaA boxes* sequences that we found, we are going to check if they are a common pattern in the rest of bacterias. We're going to find the ocurrences of the sequences in *Thermotoga petrophila*:

```
def PatternCount(Text, Pattern):
    count = 0
    for i in range(len(Text)-len(Pattern)+1):
        if Text[i:i+len(Pattern)] == Pattern:
            count = count+1
    return count
```

We observe that there are **no** ocurrences of the sequences found in *Vibrio Cholerae*. We can conclude that different bacterias have different *DnaA boxes*.

We have to try another computational approach, find clusters of *k-mers* repeated in a small interval.

## Week 2

### DNA replication (II)

1. Replication process

   The *DNA polymerases* start replicating while the parent strands are unraveling. On the lagging strand, the DNA polymerase waits until the replication fork opens around 2000 nucleotides, and because of that it forms Okazaki fragments. We need 1 primer for the leading strand and 1 primer per Okazaki fragment for the lagging strand. While the Okazaki fragments are being synthetized, a *DNA ligase* starts joining the fragments together.

2. Computational approach to find ori using deamination

   As the lagging strand is always waiting for the helicase to go forward, the lagging strand is mostly in single-stranded configuration, which is more prone to mutations. One frequent form of mutation is **deamination**,a process that causes cytosine to convert into thymine. This means that cytosine is more frequent in half of the genome.

   a) Exercise: count the ocurrences of cytosine

      We're going to count the ocurrences of the bases in a genome and include them in a symbol array.

```
def SymbolArray(Genome, symbol):
    array = {}
    n = len(Genome)
    ExtendedGenome = Genome + Genome[0:n//2]
    for i in range(n):
        array[i] = PatternCount(ExtendedGenome[i:i+(n//2)],
            symbol)
    return array


def PatternCount(Text, Pattern):
    count = 0
    for i in range(len(Text)-len(Pattern)+1):
        if Text[i:i+len(Pattern)] == Pattern:
            count = count+1
    return count
```

After executing the program, we realize that the algorithm is too inefficient.

b) Exercise: find a better algorithm for the previous exercise

This time, we are going to evaluate an element *i+1*, using the element *i*.

```
def FasterSymbolArray(Genome, symbol):
    array = {}
    n = len(Genome)
    ExtendedGenome = Genome + Genome[0:n//2]
    array[0] = PatternCount(symbol, Genome[0:n//2])
    for i in range(1, n):
        array[i] = array[i-1]
        if ExtendedGenome[i-1] == symbol:
            array[i] = array[i]-1
        if ExtendedGenome[i+(n//2)-1] == symbol:
            array[i] = array[i]+1
    return array


def PatternCount(Text, Pattern):
    count = 0
    for i in range(len(Text)-len(Pattern)+1):
        if Text[i:i+len(Pattern)] == Pattern:
            count = count+1
    return count
```

It's a viable algorithm, with a complexity of $O(n)$ instead of the previous $O(n^2)$. In *Escherichia Coli* we plotted the result of our program:

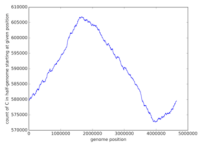

**Figure 1:** Symbol array for Cytosine in E. Coli Genome]

We can conclude that ori is located around position 4000000, because that's where the Cytosine concentration is the lowest, which indicates that the region stays single-stranded for the longest time.

3. The Skew Diagram

   Usually scientists measure the difference between *G* - *C*, which is **higher on the lagging strand** and **lower on the leading strand**.

   a) Exercise: Synthetize a Skew Array

   We're going to make a Skew Diagram, for that we'll first need a Skew Array. For that purpose we wrote:

```
def SkewArray(Genome):
    Skew = []
    Skew.append(0)
    for i in range(0, len(Genome)):
        if Genome[i] == "G":
            Skew.append(Skew[i] + 1)
        elif Genome[i] == "C":
            Skew.append(Skew[i] - 1)
        else:
            Skew.append(Skew[i])
    return Skew
```

   We can see the utility of a Skew Diagram looking at the one from *Escherichia Coli*:

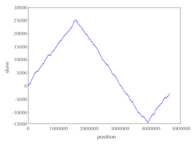

**Figure 2:** Symbol array for Cytosine in E. Coli Genome]

Ori should be located where the skew is at its minimum value.

   b) Exercise: Efficient algorithm for locating ori

Now that we know more about ori's skew value, we're going to construct a better algorithm to find it:

```
def MinimumSkew(Genome):
    positions = []
    skew = SkewArray(Genome)
    minimum = min(skew)
    return [i for i in range(0, len(Genome)) if skew[i] ==
       minimum]


def SkewArray(Genome):
    Skew = []
    Skew.append(0)
    for i in range(0, len(Genome)):
        if Genome[i] == "G":
            Skew.append(Skew[i] + 1)
        elif Genome[i] == "C":
            Skew.append(Skew[i] - 1)
        else:
            Skew.append(Skew[i])
    return Skew
```

4. Finding *DnaA boxes*

When we look for *DnaA boxes* in the minimal skew region, we can't find highly repeated *9-mers* in *Escherichia Coli*. But we found approximate sequences that are similar to our *9-mers* and only differ in 1 nucleotide.

a) Exercise: Calculate Hamming distance

The Hamming distance is the number of mismatches between 2 strings.

```
def HammingDistance(p, q):
    count = 0
    for i in range(0, len(p)):
        if p[i] != q[i]:
            count += 1
    return count
```

b) Exercise: Find approximate patterns

Now that we have our Hamming distance, we use it to find the approximate sequences:

```python
def ApproximatePatternMatching(Text, Pattern, d):
    positions = []
    for i in range(len(Text)-len(Pattern)+1):
        if Text[i:i+len(Pattern)] == Pattern:
            positions.append(i)
        elif HammingDistance(Text[i:i+len(Pattern)], Pattern)
            <= d:
            positions.append(i)
    return positions


def HammingDistance(p, q):
    count = 0
    for i in range(0, len(p)):
        if p[i] != q[i]:
            count += 1
    return count
```

c) Exercise: Count the approximate patterns

The final part is counting the approximate sequences:

```python
def ApproximatePatternCount(Pattern, Text, d):
    count = 0
    for i in range(len(Text) - len(Pattern) + 1):
        if (
            Text[i : i + len(Pattern)] == Pattern
            or HammingDistance(Text[i : i + len(Pattern)],
                Pattern) <= d
        ):
            count += 1
    return count


def HammingDistance(p, q):
    count = 0
    for i in range(0, len(p)):
        if p[i] != q[i]:
            count += 1
    return count
```

After trying out our ApproximatePatternCount in the hypothesized ori region, we find a frequent *k-mer* with its reverse complement in *Escherichia Coli*. We've finally found a computational method to find ori that seems correct.

## Week 3

### The circadian clock

Variation in gene expression permits the cell to keep track of time.

1. Computational approaches to find regulatory motifs

   a) Exercise: Find the most common nucleotides in each position

   We are going to create a **t x k** Motif Matrix, where **t** is the *k-mer* string. In each position, we'll insert the most frequent nucleotide, in upper case, and the nucleotide in lower case (if there's no popular one). Our goal is to select the **most** conserved Matrix, i.e. the Matrix with the most upper case letters. We'll use a **4 x k** Count Matrix, one row for each base.

```python
def Count(Motifs):
    k = len(Motifs[0])
    count = {'A': [0]*k, 'C': [0]*k, 'G': [0]*k, 'T': [0] * k}
    t = len(Motifs)
    for i in range(t):
        for j in range(k):
            symbol = Motifs[i][j]
            count[symbol][j] += 1
    return count
```

   Now that we have a Count Matrix, we will generate a Profile Matrix, which has the frequency of the nucleotide instead of the count:

```python
def Profile(Motifs):
    t = len(Motifs)
    profile = Count(Motifs)
    for key, v in profile.items():
        v[:] = [x / t for x in v]
    return profile


def Count(Motifs):
    k = len(Motifs[0])
    count = {'A': [0]*k, 'C': [0]*k, 'G': [0]*k, 'T': [0] * k}
    t = len(Motifs)
    for i in range(t):
        for j in range(k):
            symbol = Motifs[i][j]
            count[symbol][j] += 1
    return count
```

   b) Exercise: Form the most frequent sequence of nucleotides

Finally, we can form a Consensus string, to get a candidate regulatory motif:

```
def Consensus(Motifs):
    consensus = ""
    count = Count(Motifs)
    k = len(Motifs[0])
    for j in range(k):
        m = 0
        frequentSymbol = ""
        for symbol in "ACGT":
            if count[symbol][j] > m:
                m = count[symbol][j]
                frequentSymbol = symbol
        consensus += frequentSymbol
    return consensus


def Count(Motifs):
    k = len(Motifs[0])
    count = {'A': [0]*k, 'C': [0]*k, 'G': [0]*k, 'T': [0] * k}
    t = len(Motifs)
    for i in range(t):
        for j in range(k):
            symbol = Motifs[i][j]
            count[symbol][j] += 1
    return count
```

After obtaining the Consensus string, all we need to do is obtain the total score of our selected *k-mers*:

```python
def Score(Motifs):
    score = 0
    for i in range(len(Motifs)):
        score += HammingDistance(Motifs[i], Consensus(Motifs))
    return score


def Consensus(Motifs):
    consensus = ""
    count = Count(Motifs)
    k = len(Motifs[0])
    for j in range(k):
        m = 0
        frequentSymbol = ""
        for symbol in "ACGT":
            if count[symbol][j] > m:
                m = count[symbol][j]
                frequentSymbol = symbol
        consensus += frequentSymbol
    return consensus


def Count(Motifs):
    k = len(Motifs[0])
    count = {'A': [0]*k, 'C': [0]*k, 'G': [0]*k, 'T': [0] * k}
    t = len(Motifs)
    for i in range(t):
        for j in range(k):
            symbol = Motifs[i][j]
            count[symbol][j] += 1
    return count


def HammingDistance(p, q):
    count = 0
    for i in range(0, len(p)):
        if p[i] != q[i]:
            count += 1
    return count
```

c) Exercise: Find a set of *k-mers* that minimize the score

Applying a brute force approach for this task is not viable, we'll use a Greedy Algorithm. We first have to determine the probability of a sequence:

```
def Pr(Text, Profile):
    probability = 1
    k = len(Text)
    for i in range(k):
        probability *= Profile[Text[i]][i]
    return probability
```

We'll use this function to find the most probable *k-mer* in a sequence:

```
def ProfileMostProbableKmer(text, k, profile):
    kmer = ""
    keys = ["A", "C", "G", "T"]
    d = dict(zip(keys,profile))
    prob = -1
    for i in range(len(text)-k+1):
        if (Pr((text[i:i+k]), profile) > prob):
            prob = Pr(text[i:i+k], profile)
            kmer = text[i:i+k]
    return kmer


def Pr(Text, Profile):
    probability = 1
    k = len(Text)
    for i in range(k):
        probability *= Profile[Text[i]][i]
    return probability
```

Now we're finally ready to assemble all the pieces and implement a Greedy Motif Search Algorithm:

```python
def GreedyMotifSearch(Dna, k, t):
    BestMotifs = []
    for i in range(0, t):
        BestMotifs.append(Dna[i][0:k])
    n = len(Dna[0])
    for i in range(n-k+1):
        Motifs = []
        Motifs.append(Dna[0][i:i+k])
        for j in range(1, t):
            P = Profile(Motifs[0:j])
            Motifs.append(ProfileMostProbableKmer(Dna[j], k, P
                ))
        if Score(Motifs) < Score(BestMotifs):
            BestMotifs = Motifs
    return BestMotifs

def Score(Motifs):
    score = 0
    for i in range(len(Motifs)):
        score += HammingDistance(Motifs[i], Consensus(Motifs))
    return score


def Consensus(Motifs):
    consensus = ""
    count = Count(Motifs)
    k = len(Motifs[0])
    for j in range(k):
        m = 0
        frequentSymbol = ""
        for symbol in "ACGT":
            if count[symbol][j] > m:
                m = count[symbol][j]
                frequentSymbol = symbol
        consensus += frequentSymbol
    return consensus


def Count(Motifs):
    k = len(Motifs[0])
    count = {'A': [0]*k, 'C': [0]*k, 'G': [0]*k, 'T': [0] * k}
    t = len(Motifs)
    for i in range(t):
        for j in range(k):
            symbol = Motifs[i][j]
            count[symbol][j] += 1
    return count


def HammingDistance(p, q):
    count = 0
    for i in range(0, len(p)):
        if p[i] != q[i]:
            count += 1
    return count


def Profile(Motifs):
    t = len(Motifs)
    profile = Count(Motifs)
    for key, v in profile.items():
```

d) Motifs in tuberculosis

Tuberculosis is an infectious disease, caused by a bacteria called *Mycobacterium tuberculosis*. The bacteria can stay latent in the host for decades, in hypoxic environments. Our Greedy Algorithm can help us identify a motif that might be involved in the process.

The transcription factor behind this behaviour is **DosR**, we'll identify the binding sites:

```python
def GreedyMotifSearch(Dna, k, t):
    BestMotifs = []
    for i in range(0, t):
        BestMotifs.append(Dna[i][0:k])
    n = len(Dna[0])
    for i in range(n - k + 1):
        Motifs = []
        Motifs.append(Dna[0][i : i + k])
        for j in range(1, t):
            P = Profile(Motifs[0:j])
            Motifs.append(ProfileMostProbableKmer(Dna[j], k, P
                ))
        if Score(Motifs) < Score(BestMotifs):
            BestMotifs = Motifs
    return BestMotifs


def Score(Motifs):
    score = 0
    for i in range(len(Motifs)):
        score += HammingDistance(Motifs[i], Consensus(Motifs))
    return score


def Consensus(Motifs):
    consensus = ""
    count = Count(Motifs)
    k = len(Motifs[0])
    for j in range(k):
        m = 0
        frequentSymbol = ""
        for symbol in "ACGT":
            if count[symbol][j] > m:
                m = count[symbol][j]
                frequentSymbol = symbol
        consensus += frequentSymbol
    return consensus


def Count(Motifs):
    k = len(Motifs[0])
    count = {"A": [0] * k, "C": [0] * k, "G": [0] * k, "T":
        [0] * k}
    t = len(Motifs)
    for i in range(t):
        for j in range(k):
            symbol = Motifs[i][j]
            count[symbol][j] += 1
    return count


def HammingDistance(p, q):
    count = 0
    for i in range(0, len(p)):
        if p[i] != q[i]:
            count += 1
    return count


def Profile(Motifs):
    t = len(Motifs)
```

Our algorithm is pretty fast, but it's not optimal, and that's just a characteristic of Greedy Algorithms, they trade optimality for speed.

## Week 4

### The circadian clock (II)

The French mathematician Laplace estimated the probability that the sun will not raise tomorrow, this approach plays an important role in statistics, we cannot simplify the probability of a low-probability event to zero.

1. Motif finding with pseudocounts

   In order to account for this problem, bioinformaticians often substitute zeroes with small numbers called pseudocounts.

   a) Exercise: Create a count matrix with pseudocounts

   We are going to generate the count matrix, while adding 1 to each value as a pseudocount. As a starting point we'll use our Count(Motifs) function, and we'll tweak it to achieve our objective.

```
1  def CountWithPseudocounts(Motifs):
2      k = len(Motifs[0])
3      count = {'A': [1]*k, 'C': [1]*k, 'G': [1]*k, 'T': [1] * k}
4      t = len(Motifs)
5      for i in range(t):
6          for j in range(k):
7              symbol = Motifs[i][j]
8              count[symbol][j] += 1
9      return count
```

   We only modified the third line of the function, by starting the count at 1 instead of 0. With this simple adjustement, we have solved the problem.

   b) Exercise: Create a profile matrix with pseudocounts

   Now that we have our count matrix, we can generate a profile matrix. We'll adjust our Profile(Motifs) function for this purpose.

```python
def ProfileWithPseudocounts(Motifs):
    t = len(Motifs) + 4
    profile = CountWithPseudocounts(Motifs)
    for key, v in profile.items():
        v[:] = [x / t for x in v]
    return profile


def CountWithPseudocounts(Motifs):
    k = len(Motifs[0])
    count = {'A': [1]*k, 'C': [1]*k, 'G': [1]*k, 'T': [1] * k}
    t = len(Motifs)
    for i in range(t):
        for j in range(k):
            symbol = Motifs[i][j]
            count[symbol][j] += 1
    return count
```

We have only modified the value of t, adding 4 because the total sum of each column is different. Now that we have pseudocounts, we initialize each cell to 1, and because we have 4 rows, the total sum is now t+4;

c) Exercise: An improved Greedy Motif Search algorithm

We have all the required functions to construct a Greedy Motif Search algorithm with pseudocounts. As with all the previous exercises, we'll start with our GreedyMotifSearch(Motifs) function.

```python
def GreedyMotifSearchWithPseudocounts(Dna, k, t):
    BestMotifs = []
    for i in range(0, t):
        BestMotifs.append(Dna[i][0:k])
    n = len(Dna[0])
    for i in range(n - k + 1):
        Motifs = []
        Motifs.append(Dna[0][i : i + k])
        for j in range(1, t):
            P = ProfileWithPseudocounts(Motifs[0:j])
            Motifs.append(ProfileMostProbableKmer(Dna[j], k, P
                ))
        if Score(Motifs) < Score(BestMotifs):
            BestMotifs = Motifs
    return BestMotifs


def ProfileWithPseudocounts(Motifs):
    t = len(Motifs) + 4
    profile = CountWithPseudocounts(Motifs)
    for key, v in profile.items():
        v[:] = [x / t for x in v]
    return profile


def CountWithPseudocounts(Motifs):
    k = len(Motifs[0])
    count = {'A': [1]*k, 'C': [1]*k, 'G': [1]*k, 'T': [1] * k}
    t = len(Motifs)
    for i in range(t):
        for j in range(k):
            symbol = Motifs[i][j]
            count[symbol][j] += 1
    return count


def Score(Motifs):
    score = 0
    for i in range(len(Motifs)):
        score += HammingDistance(Motifs[i], Consensus(Motifs))
    return score


def Consensus(Motifs):
    consensus = ""
    count = Count(Motifs)
    k = len(Motifs[0])
    for j in range(k):
        m = 0
        frequentSymbol = ""
        for symbol in "ACGT":
            if count[symbol][j] > m:
                m = count[symbol][j]
                frequentSymbol = symbol
        consensus += frequentSymbol
    return consensus


def Count(Motifs):
    k = len(Motifs[0])
    count = {"A": [0] * k, "C": [0] * k, "G": [0] * k, "T":
```

All we had to do was replace the function call Profile(Motifs) with ProfileWithPseudocounts(Motif).

The performance of the new algorithm is better, but we are still not satisfied. We are going to try a different motif finding algorithm.

2. Randomized Motif Search

A randomized algorithm sounds like a very bad idea, luck doesn't seem to be a good scientific asset. Well, as nonintuitive as it might sound they are used in many cases.

We will consider **Monte Carlo algorithms** for our use case.

a) Exercise: Generate the most probable *k-mers*

We'll start off our pipeline by crafting a function that outputs the most probable *k-mers* using a profile matrix, and a DNA sequence.

We'll reuse our ProfileMostProbableKmer and Pr functions:

```
def Motifs(Profile, Dna):
    kmer_list = []
    for sequence in Dna:
        kmer = ProfileMostProbableKmer(text=sequence, k=len(
           Profile), profile=Profile)
        kmer_list.append(kmer)
    return kmer_list



def ProfileMostProbableKmer(text, k, profile):
    kmer = ""
    keys = ["A", "C", "G", "T"]
    d = dict(zip(keys,profile))
    prob = -1
    for i in range(len(text)-k+1):
        if (Pr((text[i:i+k]), profile) > prob):
            prob = Pr(text[i:i+k], profile)
            kmer = text[i:i+k]
    return kmer


def Pr(Text, Profile):
    probability = 1
    k = len(Text)
    for i in range(k):
        probability *= Profile[Text[i]][i]
    return probability
```

We obtain a list of the most probable *k-mer* for each DNA sequence.

A common approach to is starting from a collection of randomly chosen *k-mers* Motifs, construct a profile matrix and use this matrix to generate a new collection of *k-mers*.

We'll iterate many times, hoping to get a better result and seeing if the score keeps improving. This is basically what a Randomized Motif Search does, now that we understand the underlying mechanism, we'll implement a random number generator.

b) Exercise: Implement a random *k-mer* generator

In order to implement our random *k-mer* generator, we need a way to generate random integers. Each integer will be used as the index for a character in our sequences.

Python's **random** module is our choice of tool for this exercise, specifically the **randint** function.

```
1  import random
2
3
4  def RandomMotifs(Dna, k, t):
5      random_motifs = []
6      t = len(Dna)
7      l = len(Dna[0])
8      for i in range(t):
9          random_index = random.randint(1, l-k)
10         random_motifs.append(Dna[i][random_index:random_index+
               k])
11     return random_motifs
```

We are ready to develop our Randomized Search algorithm, we just need to iterate over the generation of random motifs until we stop getting good results.

c) Exercise: Random Motif Search algorithm

Finally, we only have to assemble our functions to iterate through the random motifs while the score keeps improving.

```python
import random


def RandomizedMotifSearch(Dna, k, t):
    M = RandomMotifs(Dna, k, t)
    BestMotifs = M
    while True:
        Profile = ProfileWithPseudocounts(M)
        M = Motifs(Profile, Dna)
        if Score(M) < Score(BestMotifs):
            BestMotifs = M
        else:
            return BestMotifs


def RandomMotifs(Dna, k, t):
    random_motifs = []
    t = len(Dna)
    l = len(Dna[0])
    for i in range(t):
        random_index = random.randint(1, l-k)
        random_motifs.append(Dna[i][random_index:random_index+k])
    return random_motifs


def Motifs(Profile, Dna):
    kmer_list = []
    for sequence in Dna:
        kmer = ProfileMostProbableKmer(text=sequence, k=len(Profile["A"]), profile=Profile)
        kmer_list.append(kmer)
    return kmer_list



def ProfileMostProbableKmer(text, k, profile):
    kmer = ""
    keys = ["A", "C", "G", "T"]
    d = dict(zip(keys,profile))
    prob = -1
    for i in range(len(text)-k+1):
        if (Pr((text[i:i+k]), profile) > prob):
            prob = Pr(text[i:i+k], profile)
            kmer = text[i:i+k]
    return kmer


def Pr(Text, Profile):
    probability = 1
    k = len(Text)
    for i in range(k):
        probability *= Profile[Text[i]][i]
    return probability


def ProfileWithPseudocounts(Motifs):
    t = len(Motifs) + 4
    profile = CountWithPseudocounts(Motifs)
    for key, v in profile.items():
        v[:] = [x / t for x in v]
```

Each time that we execute our algorithm, we'll get a different solution. We're going to do an experiment by running our algorithm multiple times and checking the final score.

d) Exercise: Evaluate performance of the algorithm

In order to evaluate the performance of this algorithm, we are going to run 100 iterations while keeping the motif with the greatest motif.

```python
import random
Dna = ["
    GCGCCCCGCCCGGACAGCCATGCGCTAACCCTGGCTTCGATGGCGCCGGCTCAGTTAGGGCCGGAAGTCCC
    ",
"
    CCGATCGGCATCACTATCGGTCCTGCGGCCGCCCATAGCGCTATATCCGGCTGGTGAAATCAATTGACAAC
    ",
"
    ACCGTCGATGTGCCCGGTCGCGCCGCGTCCACCTCGGTCATCGACCCCACGATGAGGACGCCATCGGCCGC
    ",
"
    GGGTCAGGTATATTTATCGCACACTTGGGCACATGACACACAAGCGCCAGAATCCCGGACCGAACCGAGCA
    ",
"
    GTAGGTCAAACCGGGTGTACATACCCGCTCAATCGCCCAGCACTTCGGGCAGATCACCGGGTTTCCCCGGTA
    ",
"
    CCGCTGGCGACGCTGTTCGCCGGCAGCGTGCGTGACGACTTCGAGCTGCCCGACTACACCTGGTGACCACCC
    ",
"
    GGCCAACTGCACCGCGCTCTTGATGACATCGGTGGTCACCATGGTGTCCGGCATGATCAACCTCCGCTGTTC
    ",
"
    GTACATGTCCAGAGCGAGCCTCAGCTTCTGCGCAGCGACGGAAACTGCCACACTCAAAGCCTACTGGGCGCA
    ",
"
    GGCAGCTGTCGGCAACTGTAAGCCATTTCTGGGACTTTGCTGTGAAAAGCTGGGCGATGGTTGTGGACCTGC
    ",
"
    TCAGCACCATGACCGCCTGGCCACCAATCGCCCGTAACAAGCGGGACGTCCGCGACGACGCGTGCGCTAGCC
    "]
t = 10
k = 15
N = 100


def RandomizedMotifSearch(Dna, k, t):
    M = RandomMotifs(Dna, k, t)
    BestMotifs = M
    while True:
        Profile = ProfileWithPseudocounts(M)
        M = Motifs(Profile, Dna)
        if Score(M) < Score(BestMotifs):
            BestMotifs = M
        else:
            return BestMotifs


def RandomMotifs(Dna, k, t):
    random_motifs = []
    t = len(Dna)
    l = len(Dna[0])
    for i in range(t):
        random_index = random.randint(1, l-k)
        random_motifs.append(Dna[i][random_index:random_index+
            k])
    return random_motifs


def Motifs(Profile, Dna):
    kmer_list = []
```

As we can see, this algorithm finds a pretty good solution.

3. Gibbs Sampling

   Thanks to our previous experiments, we now consider random search algorithms as adequate tools for our problem. The caveat of our algorithm is that it discards all previous motifs in each iteration.

   A more cautious alternative is Gibbs Sampling, which discards a single *k-mer* from the current set of motifs at each iteration.

   a) Exercise: Normalize the probabilities

      The algorithm chooses randomly at each iteration which *k-mer* will be dropped, and it chooses the replacement of that *k-mer*.

      We use pseudocounts to generate the next profile matrix, which gives us some extravagant probabilities. We need to normalize them, so their sum equals 1.

```
def Normalize(Probabilities):
    d = {}
    for k,v in Probabilities.items():
        d[k] = Probabilities[k]/sum(Probabilities.values())
    return d
```

   b) Exercise: Simulate rolling a die

      Now that our probabilities are normalized, we can focus on simulating a weighted die.

```
import random

def WeightedDie(Probabilities):
  count = 0
    p = random.uniform(0,1)
    for k,v in Probabilities.items():
        count = count+v
        if p < count:
            return k
```

      Our function returns a single element, in the next step we'll make it a subroutine of a larger function.

   c) Exercise: Generate a k-mer

      We have all the necessary tools to generate a *k-mer* based on the profile matrix.

```python
import random


def Pr(Text, Profile):
    probability = 1
    k = len(Text)
    for i in range(k):
        probability *= Profile[Text[i]][i]
    return probability


def Normalize(Probabilities):
    d = {}
    for k,v in Probabilities.items():
        d[k] = Probabilities[k]/sum(Probabilities.values())
    return d


def WeightedDie(Probabilities):
  count = 0
    p = random.uniform(0,1)
    for k,v in Probabilities.items():
        count = count+v
        if p < count:
            return k


def ProfileGeneratedString(Text, profile, k):
    n = len(Text)
    probabilities = {}
    for i in range(0,n-k+1):
        probabilities[Text[i:i+k]] = Pr(Text[i:i+k], profile)
    probabilities = Normalize(probabilities)
    return WeightedDie(probabilities)
```

d) Exercise: Implement the Gibbs Sampling algorithm

Finally, we can implement the Gibbs Sampling algorithm:

```python
import random

def GibbsSampler(Dna, k, t, N):
    Motifs = RandomMotifs(Dna, k ,t)
    BestMotifs = Motifs[:]
    for i in range(N):
        p = random.randint(0,t-1)
        del Motifs[p]
        profile = ProfileWithPseudocounts(Motifs)
        Motifs.insert(p, ProfileGeneratedString(Dna[p],
            profile, k))
        if Score(Motifs) < Score(BestMotifs):
            BestMotifs = Motifs
    return BestMotifs


def RandomMotifs(Dna, k, t):
    random_motifs = []
    t = len(Dna)
    l = len(Dna[0])
    for i in range(t):
        random_index = random.randint(1, l-k)
        random_motifs.append(Dna[i][random_index:random_index+
            k])
    return random_motifs


def ProfileWithPseudocounts(Motifs):
    t = len(Motifs) + 4
    profile = CountWithPseudocounts(Motifs)
    for key, v in profile.items():
        v[:] = [x / t for x in v]
    return profile


def CountWithPseudocounts(Motifs):
    k = len(Motifs[0])
    count = {'A': [1]*k, 'C': [1]*k, 'G': [1]*k, 'T': [1] * k}
    t = len(Motifs)
    for i in range(t):
        for j in range(k):
            symbol = Motifs[i][j]
            count[symbol][j] += 1
    return count


def Pr(Text, Profile):
    probability = 1
    k = len(Text)
    for i in range(k):
        probability *= Profile[Text[i]][i]
    return probability


def Normalize(Probabilities):
    d = {}
    for k,v in Probabilities.items():
        d[k] = Probabilities[k]/sum(Probabilities.values())
    return d


```

### Vocabulary

- k-mer: subsquences of length *k* in a biological sequence
- Frequency map: sequence –> frequency of the sequence