МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Юсипов И.И.
Калякулина А.И.
Иванченко М.В.

# ВВЕДЕНИЕ В АЛГОРИТМЫ БИОИНФОРМАТИКИ

Учебное пособие (лекционный материал)

2018

УДК _____

ББК _____

П44

**Юсипов И.И., Калякулина А.И., Иванченко М.В.** Введение в алгоритмы биоинформатики: Учебное пособие (лекционный материал). Нижний Новгород: Нижегородский госуниверситет, 2018. 118 с.

Рецензент: д.ф.-м.н. Канаков О.И.

Предметом рассмотрения настоящего курса являются основные задачи биоинформатики, которые могут быть сформулированы как задачи поиска, выравнивания, а также другие модельные задачи на строках. Цель курса состоит в изучении соответствующих методов и алгоритмов биоинформатики. Особое внимание уделяется формированию у студентов навыков реализации и применения рассматриваемых методов к решению конкретных задач биоинформатики. Задачей дисциплины является теоретическое и практическое освоение вышеупомянутых методов и алгоритмов. Курс лекций предназначен для студентов института ИТММ ННГУ, специализирующихся в области математического моделирования.

Для успешного усвоения материала необходимо предварительное изучение дисциплины “Языки программирования” и “Основы объектно-ориентированного программирования”.

Сборник лекций основан на курсах, читаемых Н.И. Вяххи [1], и работах П.А. Певзнера [2, 3].

Ответственный за выпуск:

председатель методической комиссии ИИТММ ННГУ,

Грезина А.В., к.ф.-м.н., доцент

УДК _____

ББК _____

СОДЕРЖАНИЕ

## Лекция 1. Введение. Скрытые сообщения в источнике репликации.

Репликация генома является одной из важнейших задач, выполняемых в клетке. Прежде чем клетка сможет делиться, она должна сначала продублировать свой геном, чтобы каждая из двух дочерних клеток отнаследовала свою собственную копию. В 1953 году Джеймс Уотсон и Фрэнсис Крик завершили свою статью о двойной спирали ДНК [4] известной фразой:

*It has not escaped our notice that the specific pairing we have postulated immediately suggests a possible copying mechanism for the genetic material.*

От нас не ускользнуло наблюдение о том, что при определенном связывании, которое мы постулировали, сразу можно предложить возможный механизм копирования генетического материала.

Они предположили, что две нити родительской молекулы ДНК разматываются во время репликации, а затем каждая родительская нить действует как шаблон для синтеза новой нити. В результате процесс репликации начинается с одной пары комплементарных нитей ДНК, а заканчивается двумя парами комплементарных нитей, как показано на Рисунке 1.1.

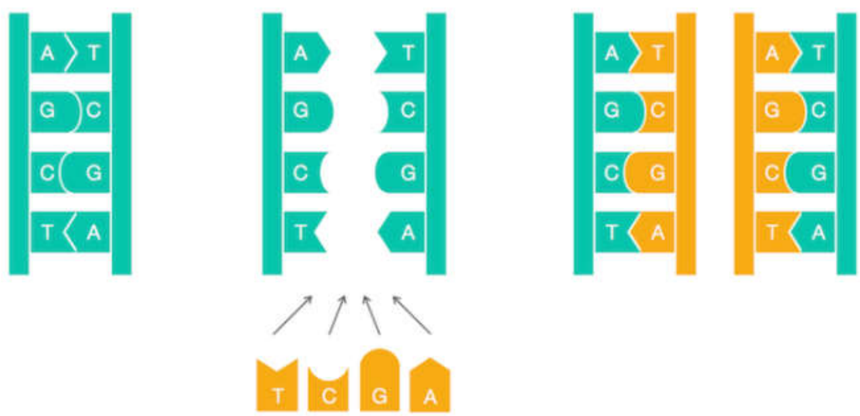



Рис. 1.1. Простейшее представление процесса репликации. Нуклеотиды аденин (A) и тимин (T), а также цитозин (C) и гуанин (G) являются комплементами друг друга. Дополнительные нуклеотиды связываются друг с другом в ДНК.

Представленный рисунок схематически моделирует репликацию ДНК, однако, детали репликации оказались намного более сложными, чем предполагали Уотсон и Крик; для обеспечения репликации ДНК требуется поразительное количество молекулярной логистики.

На первый взгляд, ученый в области компьютерных наук не может представить, что эти детали имеют какую-либо вычислительную значимость. Чтобы алгоритмически имитировать процесс в приведенном выше рисунке, нужно только взять строку, представляющую геном, и вернуть ее копию. Однако, если потратить время на изучение лежащего в основе биологического процесса, то будут получены новые алгоритмические знания по анализу репликации. Репликация начинается в геномной области, которая называется источником репликации (обозначается *oriC*) и выполняется молекулярными копировальными машинами, называемыми ДНК-полимеразами.

Поиск *oriC* представляет собой важную задачу не только для понимания того, как клетки реплицируются, но и для различных биомедицинских проблем. Например, некоторые методы генной терапии используют генетически модифицированные мини-геномы, которые называются вирусными векторами, потому что они способны проникать сквозь стенки клеток (как и настоящие вирусы). В сельском хозяйстве использовались вирусные векторы с искусственными генами, например, для изготовления морозостойких томатов и устойчивой к пестицидам кукурузы. В 1990 году генная терапия была впервые успешно испытана на людях, когда она спасла жизнь четырехлетней девочки, страдающей от тяжелого комбинированного иммунодефицитного расстройства; девочка была настолько уязвима к инфекциям, что вынуждена была жить в стерильной среде.

Идея генной терапии заключается в намеренном заражении пациента, у которого отсутствует ключевой ген, вирусным вектором, содержащим искусственный ген, который кодирует терапевтический белок. Внутри клетки вектор реплицируется и продуцирует много копий терапевтического белка, который, в свою очередь, лечит болезнь пациента. Чтобы гарантировать, что вектор действительно реплицируется внутри клетки, биологи должны знать, где именно *oriC* находится в геноме вектора, и гарантировать, что выполняемые ими генетические манипуляции не влияют на него.

В следующей задаче предполагается, что геном имеет только один *oriC* и представлен как цепочка ДНК, или последовательность нуклеотидов из четырехбуквенного алфавита {A, C, G, T}.

**Задача.** Поиск источника репликации.

Вход: последовательность ДНК.

Выход: расположение *oriC* в геноме.

Хотя проблема поиска *oriC* задает легитимный биологический вопрос, в ней нет четко определенной вычислительной проблемы. Действительно, биологи могли бы спланировать эксперимент по поиску *oriC*: например, они могли бы удалить различные короткие сегменты из генома, пытаясь найти сегмент, удаление которого прекращает репликацию. Ученые в области компьютерных наук, с другой стороны, потребовали бы больше информации, прежде чем начали бы решать задачу.

Вопрос в том, почему биологи должны заботиться о том, что думают учёные в области компьютерных наук? Вычислительные методы – единственный реальный способ ответить на многие вопросы современной биологии. Во-первых, эти методы намного быстрее, чем экспериментальные подходы; во-вторых, результаты многих экспериментов нельзя интерпретировать без вычислительного анализа. В частности, существующие экспериментальные подходы к прогнозированию *oriC* довольно трудоемки. В результате *oriC* с использованием только экспериментального подхода был найден в очень малом количестве видов. Таким образом, хотелось бы разработать вычислительный подход для поиска *oriC*, чтобы биологи могли тратить свое время и деньги на другие задачи.

Сосредоточимся на относительно простом поиске *oriC* в бактериальных геномах, большинство из которых состоят из одной кольцевой хромосомы. Исследования показали, что область бактериального генома, кодирующего *oriC*, обычно составляет несколько сотен нуклеотидов. План состоит в том, чтобы начать с бактерии, в которой известен *oriC*, а затем определить, что делает этот геномный регион особенным, чтобы разработать вычислительный подход для поиска *oriC* у других бактерий. Рассмотрим пример *Vibrio cholerae*, патогенной бактерии, вызывающей холеру; здесь представлена нуклеотидная последовательность, присутствующая в *oriC Vibrio cholerae*:

```
atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagatgatcaagagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtt
tccttaaccctctattttttacggaagaatgatcaagctgctgctcttgatcatcgtttc
```

Интересным является вопрос, как бактериальная клетка узнает, что нужно начать репликацию именно в этой короткой области гораздо большей хромосомы *Vibrio cholerae*, которая состоит из 1.108.250 нуклеотидов. В области *oriC* должно быть какое-то «скрытое сообщение», сигнализирующее клетке о необходимости начать репликацию. Действительно, известно, что инициирование репликации опосредуется *DnaA*, белком, который связывается с коротким сегментом в *oriC*, известным как *DnaA*-бокс. Можно представить себе *DnaA*-бокс как сообщение в последовательности ДНК, сигнализирующее белку *DnaA*: «Связывать здесь». Вопрос заключается в том, как найти это скрытое сообщение, не зная, как оно выглядит заранее. Другими словами, можно ли найти что-то, что выделяется в *oriC*. Это обсуждение мотивирует следующую проблему:

**Задача.** Поиск скрытого сообщения.

Вход: строка, представляющая источник генерации генома.

Выход: скрытое сообщение.

Хотя проблема скрытого сообщения представляет собой законный интуитивный вопрос, он снова не имеет никакого смысла для ученого в области компьютерных наук, потому что понятие «скрытого сообщения» точно не определено. Область *oriC Vibrio cholerae* в настоящее время так же озадачивает, как пергамент, обнаруженный Уильямом Леграном в истории Эдгара Аллана По «Золотой жук». Написанное на пергаменте имеет вид:

53‡‡†305))6·;4826)4‡.)4‡);806·;48†8^60))85;161;:‡·8<br>
†83(88)5·†;46(;88·96·?;8)·‡(;485);5·†2:·‡(;4956·2(5<br>
·—4)8^8·;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡1;48†85;4<br>
)485†528806·81(‡9;48;(88;4(‡?34;48)4‡;1‡(;:188;‡?;

Увидев пергамент, рассказчик замечает: «Если бы все сокровища Голконды были бы ценою открытия этой загадки, то я бы, вероятно, никогда не получил их». Легран отвечает: «Я сомневаюсь, чтобы человеческий ум мог придумать такой секрет, которого бы другой ум не открыл при достаточном старании». Он объясняет, что три последовательных символа **;48** появляются с удивительной частотой на пергаменте.

53‡‡†305))6·**;48**26)4‡.)4‡);806·**;48**†8^60))85;161;:‡·8<br>
†83(88)5·†;46(;88·96·?;8)·‡(**;48**5);5·†2:·‡(;4956·2(5<br>
·—4)8^8·;4069285);)6†8)4‡‡;1(‡9**;48**081;8:8‡1**;48**†85;4<br>
)485†528806·81(‡9**;48**;(88;4(‡?34**;48**)4‡;1‡(;:188;‡?;

Легранд выяснил, что пираты говорили по-английски; поэтому он предположил, что высокая частота сочетания **;48** подразумевает, что оно кодирует наиболее частое английское слово THE. Подставляя **;** для **T**, **4** для **H** и **8** для **E**, Легранд немного упростил текст (показано ниже), что в конечном итоге могло бы привести его к погребенным сокровищам.

53‡‡†305))6·**THE**26)**H**‡.)**H**‡)**TE**06·**THE**†**E**^60))**E**5**T**161**T:**‡·**E**<br>
†**E**3(**EE**)5·†**TH**6(**TEE**·96·?**TE**)·‡(**THE**5)**T**5·†2:·‡(**TH**956·2(5<br>
·—**H)E**^**E**·**TH**0692**E**5)**T**)6†**E**)**H**‡‡**T**1(‡9**THE**0**E**1**TE:E**‡1**THE**†**E**5**TH**<br>
)**HE**5†52**EE**06·**E**1(‡9**THET**(**EETH**(‡?3**HTHE**)**H**‡**T**1‡(**T:**1**EET**‡?**T**

Действуя в предположении, что ДНК – собственный язык, можно взять метод Легранда и посмотреть, найдутся ли какие-нибудь удивительно частые «слова» в *oriC Vibrio cholerae*. Здесь добавлен повод искать частые слова в *oriC*, потому что для различных биологических процессов некоторые нуклеотидные последовательности часто неожиданно появляются в небольших регионах генома. Например, **ACTAT** представляет собой частую подстроку ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT.

Термин *k*-мер используется в отношении строки длины *k* и *Count* (*Text, Pattern*) определяется как число раз, когда *k*-мер *Pattern* появляется как подстрока строки *Text*. Следуя приведенному выше примеру,

*Count* (ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT, ACTAT) = 3.

Стоит заметить, что *Count* (CG**ATATA**TCC**ATA**G, ATA) равен 3 (не 2), так как нужно учитывать перекрывающиеся вхождения *Pattern* в *Text*.

План вычисления *Count* (*Text*, *Pattern*), состоит в том, чтобы «сдвигать окно» по строке *Text*, проверяя, соответствует ли каждый *k*-мер текста подстроке *Pattern*. Поэтому *k*-мер, начинающийся с позиции *i* в строке *Text*, называется как *Text*(*i*, *k*). Во всем курсе индексирование начинается с 0. В этом случае строка *Text* начинается в позиции 0 и заканчивается в позиции (|*Text*| – 1), где |*Text*| обозначает количество символов в строке *Text*. Например, если *Text* = GACCATACTG, то *Text*(4, 3) = ATA. Следует обратить внимание, что последний *k*-мер строки *Text* начинается с позиции (|*Text*| – *k*), например, последний 3-мер GACCATACTG начинается с позиции 7. Это обсуждение приводит к следующему псевдокоду [5] для вычисления *Count* (*Text*, *Pattern)*.

```
PatternCount(Text, Pattern)
    count ← 0
    for i ← 0 to |Text| - |Pattern|
        if Text(i, |Pattern|) = Pattern
            count ← count + 1
    return count
```

**Задача 1.1.** Реализовать *PatternCount*.

Вход: строки *Text* и *Pattern*.

Выход: *Count* (*Text*, *Pattern*).

Пример входа:

GCGCG

GCG

Пример выхода:

2

Подстрока *Pattern* является наиболее частым *k*-мером в строке *Text*, если она максимизирует *Count* (*Text*, *Pattern*) среди всех *k*-меров. Можно видеть, что

**ACTAT** является наиболее частым 5-мером строки ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT, и **ATA** является наиболее частым 3-мером строки CG**ATATAT**CC**ATA**G.

Теперь поставлена строго определенная вычислительная задача.

**Задача.** Поиск наиболее часто встречающихся *k*-меров в строке.

Вход: строка *Text* и целое число *k*.

Выход: все наиболее часто встречающиеся *k*-меры в строке *Text*.

Прямой алгоритм поиска наиболее часто встречающихся *k*-меров в строке *Text* проверяет все *k*-меры, появляющиеся в этой строке (всего |*Text*| – *k* + 1 таких *k*-меров), а затем вычисляет, сколько раз каждый *k*-мер появляется в строке *Text*. Для реализации этого алгоритма, называемого *FrequentWords*, нужно сгенерировать массив *Count*, где *Count*(*i*) хранит *Count* (*Text*, *Pattern*) для *Pattern* = *Text* (*i*, *k*) (см. Рисунок 1.2).

| *Text* | **A** | **C** | **T** | G | **A** | **C** | **T** | C | C | C | A | C | C | C | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| COUNT | 2 | 1 | 1 | 1 | 2 | 1 | 1 | 3 | 1 | 1 | 1 | 3 | 3 | | |

Рис. 1.2. Массив *Count* для строки *Text* = ACTGACTCCCACCCC и *k* = 3. Например, *Count*(0) = *Count*(4) = 2, потому что ACT (выделено полужирным шрифтом) дважды появляется в тексте.

Ниже представлен псевдокод для *FrequentWords*.

```
FrequentWords(Text, k)
    FrequentPatterns ← an empty set
    for i ← 0 to |Text| - k
        Pattern ← the k-mer Text(i, k)
        Count(i) ← PatternCount(Text, Pattern)
    maxCount ← maximum value in array Count
    for i ← 0 to |Text| - k
        if Count(i) = maxCount
            add Text(i, k) to FrequentPatterns
    remove duplicates from FrequentPatterns
    return FrequentPatterns
```

**Задача 1.2.** Реализовать *FrequentWords*.

Вход: строка *Text* и целое число *k*.

Выход: все наиболее часто встречающиеся *k*-меры в строке *Text*.

Пример входа:

ACGTTGCATGTCGCATGATGCATGAGAGCT

4

Пример выхода:

CATG GCAT

Несмотря на то, что алгоритм *FrequentWords* находит наиболее частые *k*-меры, он не очень эффективен. Каждый вызов *PatternCount* (*Text*, *Pattern*) проверяет, появляется ли *k*-мер *Pattern* в позиции 0 текста, затем в позиции 1 текста и т.д. Поскольку каждый *k*-мер требует $(|Text| - k + 1)$ таких проверок, для каждого из которых требуется до *k* сравнений, общее количество шагов *PatternCount* (*Text*, *Pattern*) равно $(|Text| - k + 1) \cdot k$. Кроме того, *FrequentWords* должен вызывать *PatternCount* $(|Text| - k + 1)$ раз (один раз для каждого *k*-мера текста), так что его общее количество шагов $(|Text| - k + 1) \cdot (|Text| - k + 1) \cdot k$. Для упрощения выражения говорят, что время выполнения *FrequentWords* имеет верхнюю границу $(|Text|^2 \cdot k)$ шагов, и соответственно сложность алгоритма равна $O(|Text|^2 \cdot k)$.

Если $|Text|$ и *k* малы, как в случае поиска *DnaA*-боксов в типичном бактериальном *oriC*, то алгоритм с временем работы $O(|Text|^2 \cdot k)$ приемлем. Но как только найдется новое биологическое приложение, требующее решить задачу *FrequentWords* для очень длинного текста, возникнут проблемы.

| **k** | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|
| **count** | 25 | 12 | 8 | 8 | 5 | 4 | 3 |
| ***k*-mers** | tga | atga | gatca<br>tgatc | tgatca | atgatca | atgatcaa | atgatcaag<br>cttgatcat<br>tcttgatca<br>ctcttgatc |

Рис. 1.3. *Frequent Words* для *Vibrio cholerae*.

В таблице на Рисунке 1.3 приведены наиболее частые *k*-меры в области *oriC* для *Vibrio cholerae* (показано ниже), а также количество вхождений каждого *k*-мера.

Например, 9-мер **ATGATCAAG** появляется три раза в области *oriC Vibrio cholerae*.

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaag**ATGATCAAG**agaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat

tgataatgaatttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcatgtt
tccttaaccctctattttttacggaaga**ATGATCAAG**ctgctgctcttgatcatcgtttc

Рассматриваются, как правило, 9-меры, а не какое-либо другое значение *k*, потому что эксперименты показали, что бактериальные *DnaA*-боксы обычно имеют длину в девять нуклеотидов. Вероятность того, что существует 9-мер, появляющийся три или более раз в случайно сформированной цепочке ДНК длиной 500, составляет приблизительно 1/1300. Фактически, есть 4 разных 9-мера, встречающихся 3 или более раз в данной области: **ATGATCAAG**, CTTGATCAT, TCTTGATCA и CTCTTGATC. Маловероятность появления даже одного повторяющегося 9-мера в области *oriC Vibrio cholerae* приводит нас к рабочей гипотезе о том, что один из этих четырех 9-меров может представлять собой потенциальный *DnaA*-бокс, который при появлении несколько раз в короткой области, инициирует репликацию.

Было упомянуто, что вероятность того, что некоторый 9-мер появляется 3 или более раз в случайной цепочке ДНК длиной 500, составляет приблизительно 1/1300. Можно создать случайную строку, моделирующую цепочку ДНК, выбирая каждый нуклеотид для любой позиции с вероятностью 1/4. Построение случайных строк можно обобщить на произвольный алфавит с *A* символами, где каждый символ выбирается с вероятностью 1 / *A*.

Теперь задается вопрос: какова вероятность того, что конкретный *k*-мер *Pattern* появится (хотя бы один раз) как подстрока случайной строки длины *N*. К примеру, нужно найти вероятность того, что 01 появится в двоичной строке (*A* = 2) длины 4. Возможны такие строки:

```
0000  0001  0010  0011  0100  0101  0110  0111
1000  1001  1010  1011  1100  1101  1110  1111
```

Поскольку 01 является подстрокой 11 из этих 4-меров, и поскольку каждый 4-мер может быть сгенерирован случайным образом с вероятностью 1/16, вероятность того, что *Pattern* = 01 появится в случайном двоичном 4-мере, равна 11/16.

Изменение *Pattern* с 01 на 11 изменяет вероятность того, что *Pattern* появится в виде подстроки случайной двоичной строки. 11 появляется только в 8 бинарных 4-мерах:

```
0000  0001  0010  0011  0100  0101  0110  0111

1000  1001  1010  1011  1100  1101  1110  1111
```

В результате вероятность появления 11 в случайной двоичной строке длиной 4 равна 8/16 = 1/2.

Пусть Pr(*N*, *A*, *Pattern*, *t*) обозначает вероятность того, что строка *Pattern* появится *t* или более раз в случайной строке длины *N*, сформированной из алфавита, состоящего из *A* букв. Pr(4, 2, 01, 1) = 11/16, а Pr(4, 2, 11, 1) = 1/2. При увеличении t вероятность уменьшается. Например, вероятность нахождения 01 дважды (0101) в случайном двоичном 4-мере определяется Pr(4, 2 , 01, 2) = 1/16, потому что 0101 является единственным двоичным 4-мером, содержащим 01 дважды. Но Pr (4, 2, 11, 2) = 3/16, поскольку бинарные 4-меры 0111, 1110 и 1111 все имеют по крайней мере два появления 11.

Продемонстрировано, что разные *k*-меры имеют разные вероятности многократного появления подстроки случайной строки. В общем, это явление называется парадоксом перекрытия слов, потому что различные вхождения подстроки *Pattern* могут перекрывать друг друга для некоторых вариантов *Pattern*.

Например, в 1110 имеются два перекрывающихся вхождения 11, а также три перекрывающихся вхождения в 1111; но вхождения 01 никогда не могут пересекаться друг с другом, и поэтому 01 никогда не может встречаться более двух раз в двоичном 4-мере. Парадокс перекрытия слов делает вычисления Pr(*N*, *A*, *Pattern*, *t*) довольно сложной задачей, потому что эта вероятность сильно зависит от конкретного выбора *Pattern*. В свете усложнений, представленных парадоксом перекрытия слов, попытаемся аппроксимировать Pr(*N*, *A*, *Pattern*, *t*), а не вычислять точно.

Для аппроксимации Pr(*N*, *A*, *Pattern*, *t*) будем предполагать, что *k*-мер *Pattern* не перекрывается. В качестве примера, скажем, что мы хотим сосчитать количество тернарных строк (*A* = 3) длины 7, которые содержат 01 не реже двух раз. Помимо двух вхождений 01, имеется три оставшихся символа в строке. Предположим, что все эти символы – 2. Два вхождения 01 могут быть вставлены в 222 десятью разными способами, чтобы сформировать 7-мер, как показано ниже.

**0101**222 **01**2**01**22 **01**22**01**2 **01**222**01** 2**0101**22
2**01**2**01**2 2**01**22**01** 22**0101**2 22**01**2**01** 222**0101**

Эти два вхождения 01 вставлены в 222, но можно было бы вставить их в любой другой тернарный 3-мер. Поскольку существует $3^3 = 27$ тернарных 3-меров, получаем приближение $10 \cdot 27 = 270$ для общего числа тернарных 7-меров, содержащих два или более вхождения 01. Поскольку существует $3^7 = 2187$ всего тернарных 7-меров, оценим вероятность Pr(7, 3, 01, 2) как 270/2187.

Чтобы обобщить этот метод для аппроксимации Pr(*N*, *A*, *Pattern*, *t*) для произвольных значений параметров, рассмотрим строку *Text* длины *N*, имеющую по крайней мере *t* вхождений *k*-мера *Pattern*. Если выбрать ровно *t* этих вхождений, можно рассматривать *Text* как последовательность из *n* = *N* -

$t \cdot k$ символов, прерванных $t$ вставками $k$-мера *Pattern*. Если зафиксировать эти $n$ символов, то можно подсчитать количество различных строк в *Text*, которые могут быть сформированы путем вставки $t$ вхождений *Pattern* в строку, образованную этими $n$ символами.

Например, рассмотрим еще раз вопрос о вставке двух вхождений 01 в 222 ($n$ = 3) и добавим пять экземпляров буквы X под каждым 7-мером.

**0101**222   **01**2**01**22   **01**22**01**2   **01**222**01**   2**0101**22
**X X** XXX   **X** X**X** XX   **X** XX**X** X   **X** XXX**X**   X**X X** XX

2**01**2**01**2   2**01**22**01**   22**0101**2   22**01**2**01**   222**0101**
X**X** X**X** X   X**X** XX**X**   XX**X X** X   XX**X** X**X**   XXX**X X**

Что означает X? Вместо того, чтобы подсчитывать количество способов вставки двух вхождений 01 в 22, можно подсчитать количество способов выбрать два из пяти X для выделения жирным шрифтом.

**XX**XXX   **X**X**X**XX   **X**XX**X**X   **X**XXX**X**   X**XX**XX
X**X**X**X**X   X**X**XX**X**   XX**XX**X   XX**X**X**X**   XXX**XX**

Другими словами, подсчитывается количество способов выбрать 2 из 5 объектов, которые могут быть посчитаны биномиальным коэффициентом $\binom{5}{2} = 10$. В общем случае биномиальный коэффициент $\binom{m}{k}$ представляет собой количество способов выбора $k$ из $m$ объектов и равен $m!/[k!\,(m-k)!]$.

Для аппроксимации Pr($N$, $A$, *Pattern*, $t$) можно подсчитать количество способов вставки $t$ экземпляров $k$-мера *Pattern* в строку фиксированной длины $n = N - t \cdot k$. Поэтому будем иметь $n + t$ вхождений $X$, из которых нужно выбрать $t$ мест размещения *Pattern*, что дает общее число $\binom{n+t}{t}$. Затем нужно умножить $\binom{n+t}{t}$ на общее число строк длины $n$, в которое можно вставить $t$ экземпляров *Pattern*, чтобы получить приблизительное общее число $\binom{n+t}{t} \cdot A^n$ (фактическое число будет меньше). Разделение на общее число строк длины $N$ дает искомое приближение:

$$\Pr(N, A, Pattern, t) \approx \frac{\left[\binom{n+t}{t} \cdot A^n\right]}{A^n} = \frac{\binom{N-t(k-1)}{t}}{A^{tk}}$$

Теперь вычислим вероятность того, что конкретный 5-мер ACTAT будет входить по крайней мере $t$ = 3 раза в случайную цепочку ДНК ($A$ = 4) длины $N$ = 30. Поскольку $n = N - t \cdot k = 15$, оценка вероятности:

$$\Pr(30, 4, ACTAT, 3) \approx \frac{\binom{30-3\cdot 4}{3}}{4^{15}} \approx 7.599 \cdot 10^{-7}$$

Точная вероятность близка к $7{,}572 \cdot 10^{-7}$, что свидетельствует о том, что полученное приближение является относительно точным для неперекрывающихся шаблонов. Однако оно становится неточным для перекрывающихся шаблонов, например, Pr(30, 4, AAAAA, 3) ≈ 1.148 · $10^{-3}$.

Вероятность нахождения ACTAT в случайной цепочке ДНК длиной 30 достаточно мала. Однако стоит помнить, что первоначальная цель состояла в том, чтобы аппроксимировать вероятность того, что существует 5-мер, появляющийся три или более раз. В общем случае вероятность того, что некоторый $k$-мер появляется $t$ или более раз в случайной строке длины $N$, сформированной над алфавитом из $A$ букв, записывается Pr($N, A, k, t$).

Pr($N, A, k, t$) было аппроксимировано как

$$\Pr(N, A, Pattern, t) \approx \frac{\binom{N-t(k-1)}{t}}{A^{tk}}$$

Примерная вероятность того, что *Pattern* не появится $t$ или более раз равна $1-p$. Таким образом, вероятность того, что все $A^k$ шаблонов появятся меньше $t$ раз в случайной строке длины $N$, может быть приблизительно равна:

$$\Pr(N, A, Pattern, t) \approx 1 - (1-p)^{A^k}$$

Чтобы избежать ошибок округления, предположим, что $p$ примерно одинакова для любого *Pattern*, тогда можно аппроксимировать Pr($N, A, k, t$), умножив $p$ на общее число $A^k$ всех $k$-меров:

$$\Pr(N, A, k, t) \approx p \cdot A^k = \frac{\binom{N-t(k-1)}{t}}{A^{tk} \cdot A^k} = \frac{\binom{N-t(k-1)}{t}}{A^{(t-1)k}}$$

Это приближение также является грубым упрощением, поскольку вероятность Pr($N, A, Pattern, t$) варьируется в зависимости от разных вариантов $k$-меров и предполагает, что вхождения разных $k$-меров являются независимыми событиями. Например, если аппроксимировать Pr(500, 4, 9, 3), приведенная выше формула приводит к следующему приближению:

$$\Pr(500, 4, 9, 3) \approx \frac{\binom{500-3\cdot 8}{3}}{4^{(3-1)\cdot 9}} = \frac{17861900}{68719476736} \approx \frac{1}{3847}$$

Из-за перекрывающихся строк это приближение отклоняется от истинного значения, которое ближе к 1/1300. Более точные оценки представлены в [6].

## Лекция 2. Скрытые сообщения. Репликация ДНК.

Нуклеотиды A и T являются дополнениями друг друга, также как G и C. Имея одну нить и источник «свободных плавающих» нуклеотидов, можно представить себе синтез комплементарной цепи на шаблонной нити. Эта модель репликации была строго подтверждена Мезельсоном и Шталем в 1958 году [7].

Эксперимент Мезельсона-Шталя, проведенный в 1958 году Мэтью Мезельсоном и Франклином Шталем, иногда называют «самым красивым экспериментом в биологии». В конце 1950-х годов биологи обсудили три противоречивые модели репликации ДНК, проиллюстрированные на рисунке 2.1. Полуконсервативная гипотеза предполагала, что каждая родительская цепь действует как шаблон для синтеза дочерней цепи. В результате каждая из двух дочерних молекул содержит одну родительскую нить и одну вновь синтезированную нить. Консервативная гипотеза предполагала, что целая двухцепочечная родительская молекула ДНК служит в качестве матрицы для синтеза новой дочерней молекулы, в результате чего одна молекула была с двумя родительскими цепями, а другая – с двумя вновь синтезированными цепями. Дисперсионная гипотеза предполагала, что какой-то механизм разрушает основную цепь ДНК на кусочки и соединяет интервалы синтезированной ДНК, так что каждая из дочерних молекул представляет собой смесь старой и новой двухцепочечной ДНК.

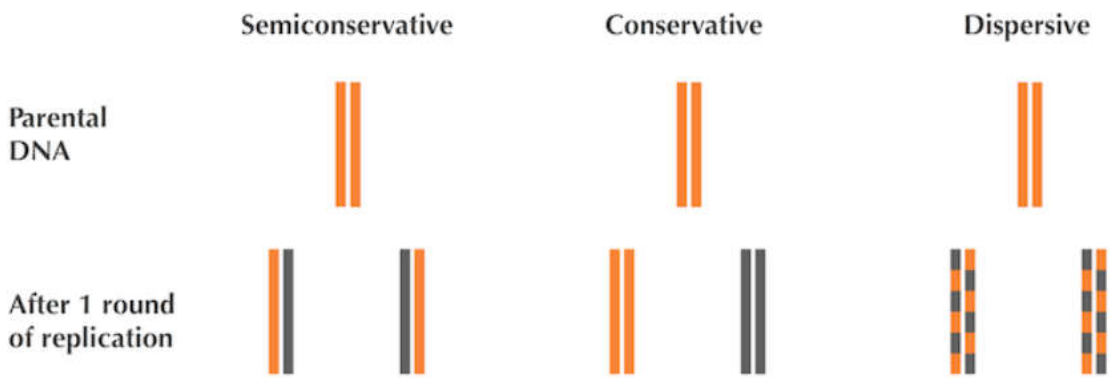



Рис. 2.1. Три модели: родительская ДНК окрашена в желтый цвет, а вновь синтезированная ДНК окрашена в черный цвет.

Понимание Мезельсона и Шталя основывалось на том факте, что один изотоп азота, азот-14 ($^{14}N$), легче и более распространен, чем азот-15 ($^{15}N$). Зная, что молекулярная структура ДНК содержит $^{14}N$, Мезельсон и Шталь выращивали *E.coli* для многих эпизодов репликации в среде $^{15}N$, что заставляло бактерии набирать вес, поскольку они поглощали более тяжелый изотоп в ДНК.

Когда они были уверены, что бактериальная ДНК была насыщена $^{15}N$, они переносили эти тяжелые клетки *E.coli* в менее плотную среду $^{14}N$.

Красота эксперимента Мезельсона-Шталя заключается в том, что вся новая синтезированная ДНК будет содержать исключительно $^{14}N$, а три существующих гипотезы для репликации ДНК предсказали разные результаты для того, как этот изотоп $^{14}N$ будет включен в ДНК. В частности, после одного эпизода репликации консервативная модель предсказала, что половина ДНК

*E.coli* будет по-прежнему иметь только $^{15}N$ и, следовательно, будет более тяжелой, тогда как другая половина будет иметь только $^{14}N$ и будет легче. Однако, когда они пытались разделить ДНК *E.coli* по массе с помощью центрифуги после одного эпизода репликации, вся ДНК имела одинаковую плотность. Именно так они опровергли консервативную гипотезу раз и навсегда.

К сожалению, этот эксперимент не смог устранить ни одну из двух других моделей, так как видно, что как дисперсионные, так и полуконсервативные гипотезы предсказывали, что вся ДНК после одного эпизода репликации будет иметь одинаковую плотность.

Рассмотрим сначала дисперсионную модель, в которой говорится, что каждая дочерняя нить ДНК образуется наполовину разделенными кусками родительской нити и наполовину новой ДНК. Если бы эта гипотеза была верна, то после двух циклов репликации любая дочерняя цепь ДНК должна содержать около 25% $^{15}N$ и около 75% $^{14}N$. Другими словами, вся ДНК должна иметь одинаковую плотность. И все же, когда Мезельсон и Шталь открыли центрифугу после двух эпизодов репликации *E.coli*, они этого не наблюдали.

Вместо этого они обнаружили, что ДНК разделена на две разные плотности. Это то, что предсказала полуконсервативная модель: через один цикл каждая клетка должна обладать одной нитью $^{14}N$ и одной нитью $^{15}N$; после двух циклов половина молекул ДНК должна иметь одну нить $^{14}N$ и одну нить $^{15}N$, тогда как другая половина должна иметь две нити $^{14}N$, создавая две разные плотности, которые они заметили.

Мезельсон и Шталь отвергли консервативную и дисперсионную гипотезы о репликации, и все же они хотели убедиться, что полуконсервативная гипотеза подтверждается дальнейшей репликацией *E.coli*. Эта модель предсказала, что после трех эпизодов репликации одна четверть молекул ДНК должна по-прежнему иметь нить $^{15}N$, в результате чего 25% ДНК будет иметь промежуточную плотность, тогда как остальные 75% должны быть легче, имея только $^{14}N$. Это действительно то, что Мезельсон и Шталь наблюдали в лаборатории, а полуконсервативная гипотеза остается в силе и по сей день.

Начало и конец нити ДНК обозначаются 5' и 3' соответственно. Сахарный компонент нуклеотида имеет кольцо из пяти атомов углерода, которые обозначены как 1', 2', 3 ', 4' и 5' на рисунке 2.2. 5' атом соединен с фосфатной группой в нуклеотиде и, в итоге, с 3' концом соседнего нуклеотида. 3' атом соединен с другим соседним нуклеотидом в цепи нуклеиновой кислоты. В результате два конца нуклеотида называются 5' концом и 3' концом.

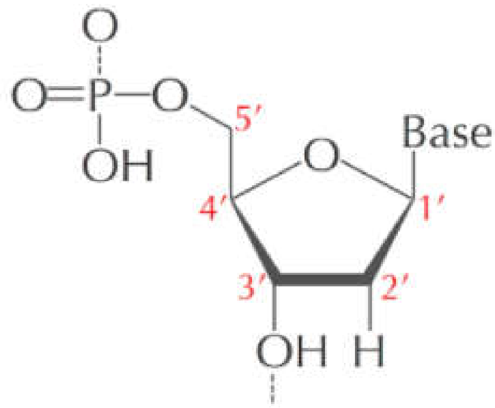



Рис. 2.2. Схема нуклеотида.

Если уменьшить масштаб до уровня двойной спирали, можно видеть на рисунке 2.3, что любой фрагмент ДНК ориентирован 3' атомом на одном конце и 5' атомом на другом конце. В качестве стандарта нить ДНК всегда считывается в направлении 5'→ 3'. Обратите внимание, что ориентации противоположны друг другу в комплементарных цепях.

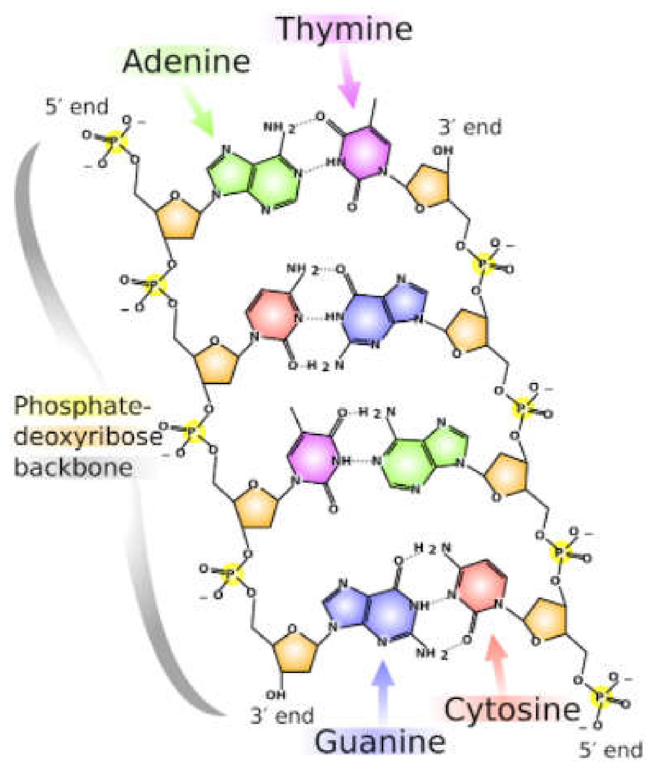



Рис. 2.3. Схема спирали ДНК.

На Рисунке 2.4 показана последовательность нуклеотидов AGTCGCATAGT и ее дополнительная последовательность ACTATGCGACT.

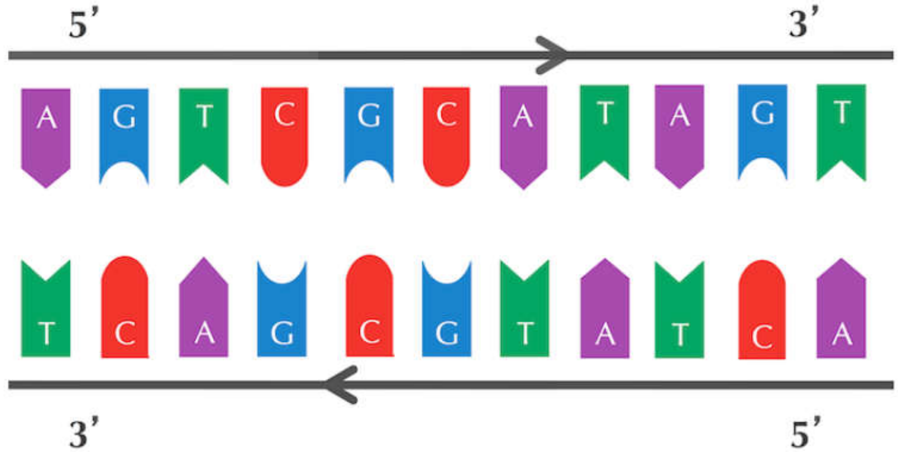



Рис. 2.4. Дополнительные нити проходят в противоположных направлениях. Каждая нить считывается в направлении 5'→ 3'.

На этом этапе можно подумать, что допущена ошибка, поскольку дополнительная цепочка на этом рисунке читается слева направо как TCAGCGTATCA, а не ACTATGCGACT. Но нет: каждая цепочка ДНК имеет направление, а дополнительная цепь проходит в противоположном направлении к матричной нити, как показано стрелками на рисунке. Каждая нить считывается в направлении 5'→ 3'.

Имея нуклеотид $p$, обозначим его комплементарный нуклеотид как $\bar{p}$. Обратно комплементарной для строки $Pattern = p_1 \dots p_n$ будет строка $\overline{Pattern} = \overline{p_n} \dots \overline{p_1}$, образованная путем взятия дополнения каждого нуклеотида в *Pattern*, а затем обращения результирующей строки.

**Задача 2.1.** Найти обратно комплементарную цепочку ДНК.

Вход: строка ДНК *Pattern*.

Выход: строка $\overline{Pattern}$, обратно комплементарная к *Pattern*.

Пример входа:

AAAACCCGGT

Пример выхода:

ACCGGGTTTT

Интересно отметить, что среди четырех наиболее часто встречающихся 9-меров в *oriC Vibrio cholerae* ATGATCAAG и CTTGATCAT являются обратно комплементарными друг к другу, шесть вхождений этих строк показаны ниже.

```
atcaatgatcaacgtaagcttctaagcATGATCAAGgtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca
cggaaagATGATCAAGagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt
```

```
acgaaagcatgatcatggctgttgttctgtttatcttgttttgactgagacttgttagga
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctCTTGATCATcgatccgattgaag
atcttcaattgttaattctcttgcctcgactcatagccatgatgagctCTTGATCATgtt
tccttaaccctctattttttacggaagaATGATCAAGctgctgctCTTGATCATcgtttc
```

Найти 9-мер, который появляется шесть раз (либо сам, либо его обратное дополнение) в цепочке ДНК длиной 500, гораздо более удивительно, чем найти 9-мер, который появляется три раза (сам). Это наблюдение приводит к рабочей гипотезе о том, что ATGATCAAG и его обратное дополнение CTTGATCAT действительно представляют собой *DnaA*-боксы в *Vibrio cholerae*. Этот вычислительный вывод имеет биологический смысл, поскольку белок *DnaA*, который связывается с *DnaA*-боксами и инициирует репликацию, не заботится о том, с какой из двух нитей он связывается. Таким образом, для наших целей как ATGATCAAG, так и CTTGATCAT представляют собой *DnaA*-боксы.

Однако прежде чем заключить, что *DnaA*-бокс для *Vibrio cholerae* найден, нужно проверить, имеются ли в геноме *Vibrio cholerae* другие короткие области, демонстрирующие множественные вхождения ATGATCAAG (или CTTGATCAT). Возможно, что эти строки встречаются на протяжении всего генома *Vibrio cholerae*, а не только в области *oriC*. Для этого необходимо решить следующую задачу *PatternMatching*.

**Задача 2.2.** Найти все вхождения шаблона в строку.

Вход: две строки, *Pattern* и *Genome*.

Выход: набор целых чисел, разделенных пробелами, указывающих все начальные позиции в *Genome*, где *Pattern* входит в виде подстроки.

Пример входа:

ATAT

GATATATGCATATACTT

Пример выхода:

1 3 9

Нельзя делать вывод о том, что ATGATCAAG / CTTGATCAT является скрытым сообщением для всех бактериальных геномов без предварительной проверки, даже если оно появляется в известных регионах *oriC* других бактерий. Возможно, что эффект множественного появления ATGATCAAG / CTTGATCAT в области *oriC Vibrio cholerae* – это статистический случай, который не имеет ничего общего с репликацией. Или, может быть, у разных бактерий есть разные *DnaA*-боксы.

Проверим предложенную область *oriC* для *Thermotoga petrophila*, бактерии, для которой чрезвычайно жаркие условия являются наиболее комфортными; она была впервые найдена в воде под нефтяными пластами, где температура может превышать 80° C, что отражено в названии.

```
aactctatacctcctttttgtcgaatttgtgtgatttatagagaaaatcttattaactga
aactaaaatggtaggtttggtggtaggttttgtgtacattttgtagtatctgatttttaa
ttacataccgtatattgtattaaattgacgaacaattgcatggaattgaatatatgcaaa
acaaacctaccaccaaactctgtattgaccattttaggacaacttcagggtggtaggttt
ctgaagctctcatcaatagactattttagtctttacaaacaatattaccgttcagattca
agattctacaacgctgttttaatgggcgttgcagaaaacttaccacctaaaatccagtat
ccaagccgatttcagagaaacctaccacttacctaccacttacctacccgggtggta
agttgcagacattattaaaaacctcatcagaagcttgttcaaaaatttcaatactcgaaa
cctaccacctgcgtcccctattatttactactactaataatagcagtataattgatctga
```

Эта область не содержит ни одного вхождения ATGATCAAG или CTTGATCAT. Таким образом, различные бактерии могут использовать разные *DnaA*-боксы как «скрытые сообщения» для белка *DnaA*.

Применение *FrequentWords* в области *oriC* выше показывает, что следующие шесть 9-меров появляются в этой области 3 или более раз:

AACCTACCA AAACCTACC ACCTACCAC

CCTACCACC GGTAGGTTT TGGTAGGTT

Здесь должно происходить нечто своеобразное, так как крайне маловероятно, что шесть разных 9-меров будут встречаться так часто в одной и той же короткой области случайной строки. Можно проконсультироваться с Ori-Finder [8], программным инструментом для поиска источников репликации в последовательностях ДНК. Это программное обеспечение выбирает CCTACCACC (вместе с обратно комплементарным GGTGGTAGG) в качестве рабочей гипотезы для *DnaA*-бокса в *Thermotoga petrophila*. Вместе эти два взаимодополняющих 9-мера появляются пять раз в источнике репликации:

aactctatacctcctttttgtcgaatttgtgtgatttatagagaaaatcttattaactga<br>
aactaaaatggtaggttt**GGTGGTAGG**ttttgtgtacattttgtagtatctgatttttaa<br>
ttacataccgtatattgtattaaattgacgaacaattgcatggaattgaatatatgcaaa<br>
acaaa**CCTACCACC**aaactctgtattgaccattttaggacaacttcag**GGTGGTAGG**ttt<br>
ctgaagctctcatcaatagactattttagtctttacaaacaatattaccgttcagattca<br>
agattctacaacgctgttttaatgggcgttgcagaaaacttaccacctaaaatccagtat<br>
ccaagccgatttcagagaaacctaccacttacctaccactta**CCTACCACC**cgggtggta<br>
agttgcagacattattaaaaacctcatcagaagcttgttcaaaaatttcaatactcgaaa<br>
**CCTACCACC**tgcgtcccctattatttactactactaataatagcagtataattgatctga

Теперь попытаемся найти *oriC* в новом секвенированном бактериальном геноме. Поиск скоплений либо ATGATCAAG / CTTGATCAT, либо CCTACCACC / GGTGGTAGG вряд ли поможет, поскольку этот новый геном может использовать совершенно другое скрытое сообщение. Изменим вычислительную цель: вместо того, чтобы находить скопления конкретного *k*-мера, попробуем найти все *k*-меры, которые образуют скопления в геноме. Их расположение может пролить свет на местоположение *oriC*.

План состоит в том, чтобы с помощью скользящего вдоль генома окна фиксированной длины *L* искать область, где *k*-мер появляется несколько раз подряд. Значение параметра *L* = 500 отражает типичную длину *oriC* в бактериальных геномах.

Определим *k*-мер как «кламп» (clump), если он появляется много раз в течение короткого интервала генома. Более формально: для заданных целых чисел *L* и *t*, *k*-мер *Pattern* формирует (*L*, *t*)-кламп внутри (более длинной) строки *Genome*, если в строке *Genome* есть интервал длины *L*, в котором этот *k*-мер появляется как минимум *t* раз. (Это определение предполагает, что *k*-мер полностью входит в этот интервал.) Например, TGCA формирует (25,3)-кламп в следующем геноме:

gatcagcataagggtccC**TGCA**A**TGCA**TGACAAGCC**TGCA**GTtgttttac

В предыдущих примерах областей *oriC* ATGATCAAG образует (500,3)-кламп в геноме *Vibrio cholerae*, а CCTACCACC образует (500,3)-кламп в геноме *Thermotoga petrophila*. Теперь можно сформулировать следующую задачу.

**Задача.** Поиск шаблонов, формирующих клампы в строке.

Вход: строка *Genome* и целые числа *k, L* и *t*.

Выход: все *k*-меры, формирующие (*L*, *t*)-клампы в строке *Genome*.

Можно решить задачу *ClumpFinding*, применив алгоритм для задачи *FrequentWords* к каждому окну длины *L* в геноме. Однако, если алгоритм для задачи *FrequentWords* не очень эффективен, то такой подход может быть непрактичным. Напомним, что *FrequentWords* имеет время работы $O(L^2 \cdot k)$. Применение этого алгоритма к каждому окну длины *L* в строке *Genome* приведет к алгоритму с временем работы $O(L^2 \cdot k \cdot |Genome|)$. Более того, даже если использовать более быстрый алгоритм для задачи *FrequentWords*, время работы остается высоким даже при попытке проанализировать бактериальный, не говоря уже о человеческом геноме.

**Задача 2.3.** Найти все шаблоны, формирующие клампы в строке.

Вход: строка *Genome* и целые числа *k, L* и *t*.

Выход: все различные *k*-меры, формирующие (*L*, *t*)-клампы в строке *Genome*.

Пример входа:

CGGACTCGACAGATGTGAAGAACGACAATGTGAAGACTCGACAC GACAGAGTGAAGAGAAGAGGAAACATTGTAA

5 50 4

Пример выхода:

CGACA GAAGA

В геноме бактерии *E.coli* можно найти сотни разных 9-меров, образующих (500,3)-клампы, и совершенно неясно, какой из этих 9-меров может представлять собой *DnaA*-бокс в области *oriC* бактерии.

Можно рассмотреть процесс репликации более подробно. Как показано на рисунке 2.5, две взаимодополняющие нити ДНК, проходящие в противоположных направлениях вокруг круговой хромосомы, начинаются с *oriC*. Когда нити разворачиваются, они создают две вилки репликации, которые расширяются в обоих направлениях вокруг хромосомы до тех пор, пока нити полностью не разделятся на конце репликации (обозначается *terC*). Окончание репликации расположено примерно напротив *oriC* в хромосоме.

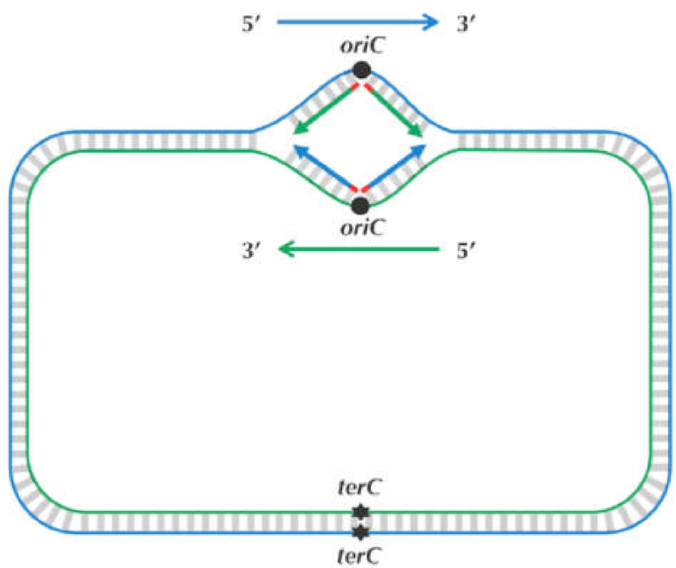



Рис. 2.5. Четыре ДНК-полимеразы при работе с репликацией хромосомы в качестве репликационных вилок при прохождении от *oriC* до *terC*. Синяя нить направлена по часовой стрелке; зеленая нить направлена против часовой стрелки.

Важно знать о репликации то, что ДНК-полимераза не дожидается полного разделения двух родительских нитей до начала репликации; вместо этого она начинает копировать, пока нити распутываются, как показано на рисунке 2.5. Таким образом, всего четыре ДНК-полимеразы, каждая из которых отвечает за одну половину нити, могут начинать с *oriC* и реплицировать всю хромосому. Чтобы начать репликацию, ДНК-полимеразе нужен праймер, короткий комплементарный сегмент (показан красным), который связывается с родительской цепью, и начинает работу ДНК-полимеразы. После начала разделения нитей каждая из четырех ДНК-полимераз начинает репликацию путем добавления нуклеотидов, начиная с праймера, и проходит вокруг хромосомы от *oriC* до *terC* либо по часовой стрелке, либо против часовой стрелки.

Когда все четыре ДНК-полимеразы достигнут *terC*, ДНК хромосомы будет полностью реплицирована, что приведет к появлению двух пар дополнительных нитей, как показано на рисунке 2.6, и клетка будет готова к делению.

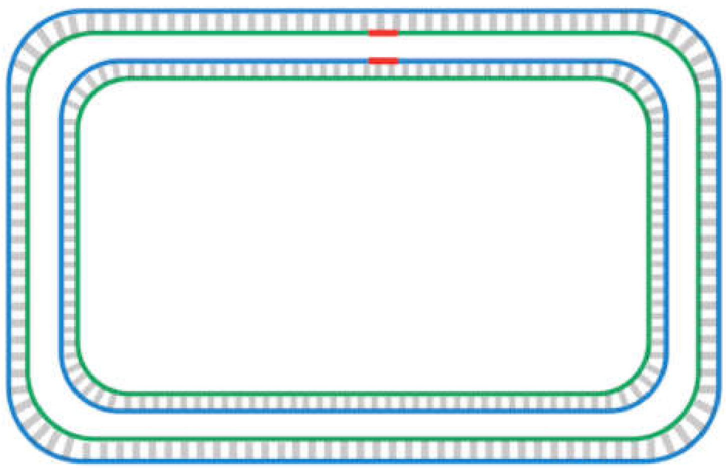


Рис. 2.6. Четыре ДНК-полимеразы реплицировали хромосому.

Однако, в данном представлении есть серьезный недостаток; процесс репликации описан простейшим образом, для лучшего понимания дальнейшего процесса.

Проблема такого описания заключается в предположении, что ДНК-полимеразы могут копировать ДНК в любом направлении вдоль нити (то есть, как 5'→ 3', так и 3'→ 5'). Однако, ДНК-полимеразы являются однонаправленными. Это означает, что они могут пересекать цепочку ДНК только в направлении 3'→ 5', противоположном направлению 5'→ 3'.

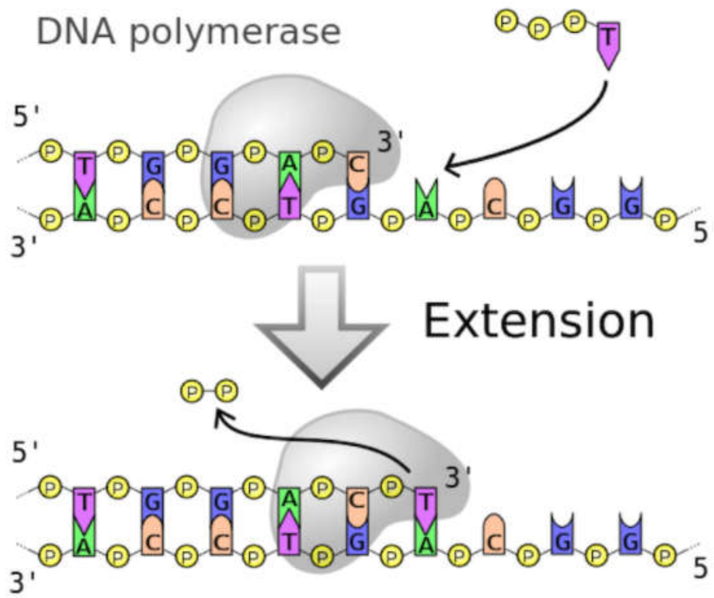



Рис. 2.7. ДНК-полимераза копирует цепочку в направлении 3'→ 5'. Стоит обратить внимание, что она создает дочернюю нить в направлении 5'→ 3'.

Однонаправленность ДНК-полимеразы требует серьезного пересмотра представленной наивной модели репликации. Если требуется пройти по ДНК от *oriC* до *terC*, то существует четыре разных половины нитей родительской ДНК, соединяющих *oriC* с *terC*, как показано на рисунке 2.8. Две из этих полунитей пересекаются от *oriC* до *terC* в направлении 5'→ 3' и называются прямыми полунитями (представлены тонкой синей и зеленой линиями на рисунке 2.8). Остальные две половины пересекаются от *oriC* до *terC* в направлении 3'→ 5' и, таким образом, называются обратными полунитями (представлены толстой синей и зеленой линиями на рисунке 2.8).

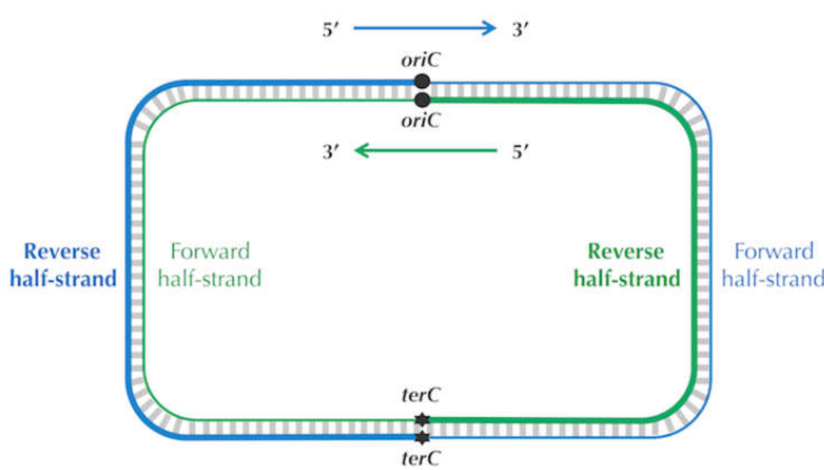



Рис. 2.8. Комплементарные нити ДНК с прямыми и обратными полунитями, показанные тонкими и толстыми линиями, соответственно.

Поскольку ДНК-полимераза может двигаться только в обратном направлении (3'→ 5'), она может копировать нуклеотиды без остановки от *oriC* до *terC* вдоль обратных полунитей, как показано на рисунке 2.9. Однако, репликация на прямых полунитях отличается, поскольку ДНК-полимераза не может двигаться в прямом направлении (5'→ 3'); на этих полунитях ДНК-полимераза должна осуществлять репликацию назад, в сторону *oriC*.

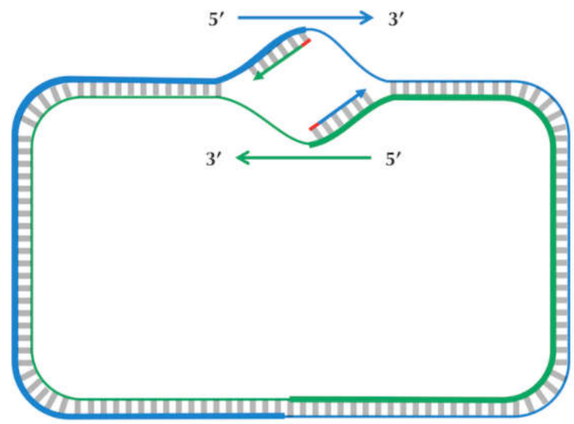



Рис. 2.9. Репликация начинается с *oriC* (праймеры, показанные красным) с синтезом фрагментов на обратных полунитях (показаны толстыми линиями). ДНК-полимераза ждет, пока вилка репликации не откроется на некоторый (небольшой) интервал до того, как она начнет копировать прямые полуниты (показаны тонкими линиями) назад к *oriC*.

На прямой полунити, чтобы реплицировать ДНК, ДНК-полимераза должна ждать, когда откроется вилка репликации (приблизительно на 2000 нуклеотидов), пока в конце вилки не образуется новый праймер; после этого ДНК-полимераза начинает реплицировать небольшой кусок ДНК, начиная с этого праймера, перемещаясь назад в направлении *oriC*. Когда две ДНК-полимеразы на прямых полунитях достигают *oriC*, получим ситуацию, показанную на рисунке 2.10.

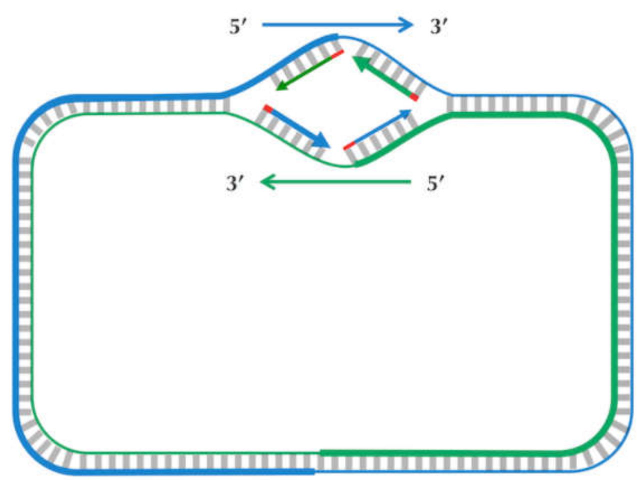



Рис. 2.10. Дочерние фрагменты теперь синтезируются (с некоторой задержкой) на передних полунитях (показаны тонкими линиями).

После этого момента репликация на каждой обратной полунити прогрессирует непрерывно; однако, ДНК-полимераза на прямой полунити не имеет другого выбора, кроме как ждать снова, пока вилка репликации не откроется еще на (примерно) 2000 нуклеотидов. Затем требуется, чтобы новый праймер начал синтезировать другой фрагмент назад к *oriC*. В целом, репликация на прямой полунити требует периодической остановки и перезапуска, что приводит к синтезу коротких фрагментов Оказаки (названы в честь Рэйдзи Оказаки, впервые описавшего их в 1968 г.) из нескольких праймеров, которые дополняют интервалы на прямых полунитях. Эти фрагменты можно увидеть на рисунке 2.11.

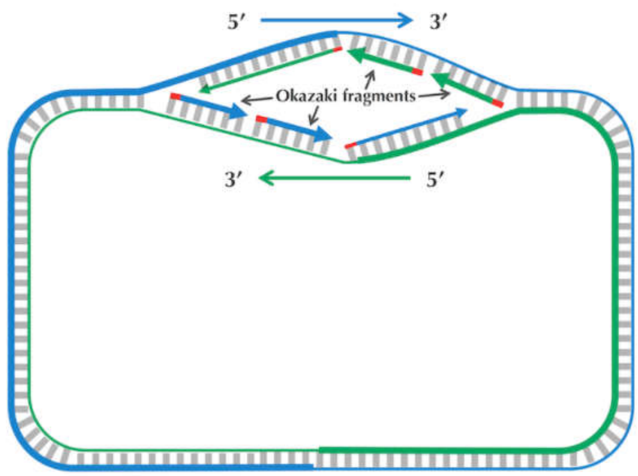



Рис. 2.11. Репликационная вилка продолжает расти. Для каждой из обратных полунитей (показаны толстыми линиями) требуется один праймер, а передние полунити (показаны тонкими линиями) требуют нескольких праймеров для синтеза фрагментов Оказаки. Два из этих праймеров показаны красным цветом на каждой прямой полунити.

Когда вилка репликации достигает *terC*, процесс репликации почти завершен, поскольку вся ДНК синтезирована. Однако, между несоединенными фрагментами Оказаки все еще остаются пробелы, как показано на рисунке 2.12 сверху.

Затем последовательные фрагменты Оказаки соединяются ферментом, называемым ДНК-лигазой, в результате чего образуются две интактные дочерние хромосомы, каждая из которых состоит из одной родительской цепи и одной недавно синтезированной дочерней цепи. В действительности, ДНК-лигаза не ждет, пока все фрагменты Оказаки не будут воспроизведены, чтобы начать соединять их.

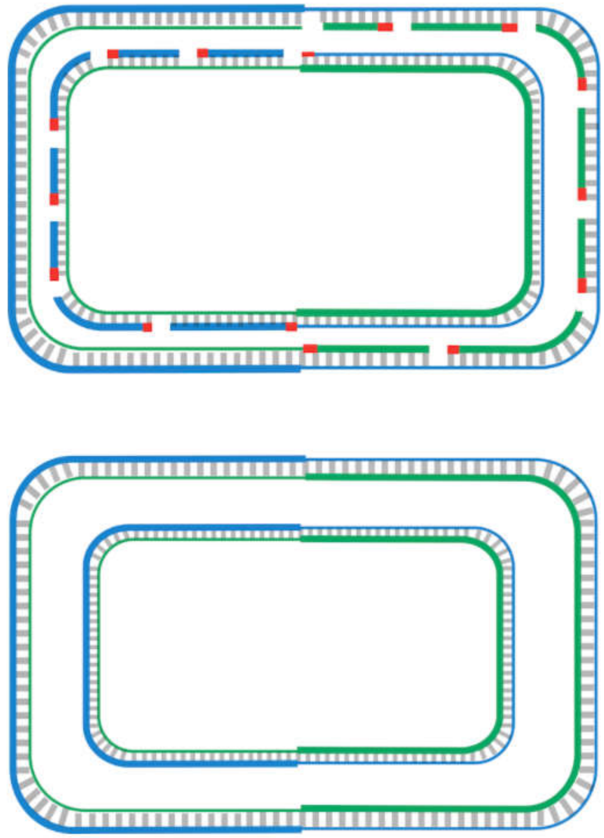


Рис. 2.12. Соединение фрагментов Оказаки.

Биологи называют обратную полунить (толстые линии) ведущей полунитью, так как единственная ДНК-полимераза беспрепятственно пересекает эту полунить, а прямую полунить (тонкие линии) – отстающей полунитью, поскольку она используется в качестве шаблона многими полимеразами ДНК, останавливающимися и начинающими репликацию.

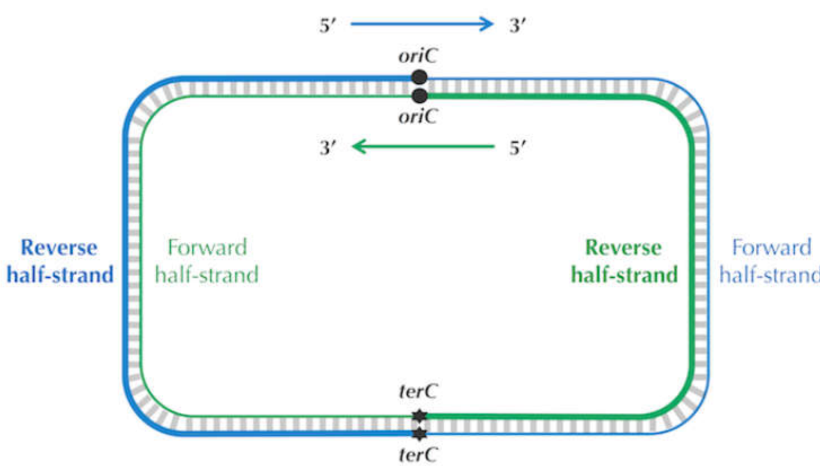



Рис. 2.13. Прямая и обратная полунити.

## Лекция 3. Прямая и обратная полунити. Открытые проблемы поиска источника репликации.

Ранее было показано, что по мере расширения вилки репликации ДНК-полимераза быстро синтезирует ДНК на обратной полунити, но происходят задержки на прямой полунити. Рассмотрим асимметрию репликации ДНК для разработки нового алгоритма поиска *oriC*.

Обратите внимание, что, поскольку репликация обратной полунити протекает быстро, она является двухцепочечной большую часть жизни. И наоборот, прямая полунить является одноцепочечной больший объем своей жизни, ожидая использования в качестве шаблона для репликации. Это несоответствие между прямыми и обратными полунитями важно, поскольку одноцепочечная ДНК имеет гораздо более высокую скорость мутации, чем двухцепочечная ДНК. В частности, если один из четырех нуклеотидов в одноцепочечной ДНК имеет большую тенденцию, чем другие нуклеотиды, мутировать в одноцепочечной ДНК, то должен быть замечен недостаток этого нуклеотида на прямой полунити.

Следуя этой идее, сравним количество нуклеотидов обратной и прямой полунитей. Если они существенно различаются, то можно будет разработать алгоритм, который попытается выявить эти различия в геномах, для которых *oriC* неизвестен. Количество нуклеотидов для прямой и обратной полунитей генома *Thermotoga petrophila* показано в таблице на рисунке 3.1.

| | C | G | A | T |
|---|---|---|---|---|
| **Entire strand** | 427419 | 413241 | 491488 | 491363 |
| **Reverse half-strand** | 219518 | 201634 | 243963 | 246641 |
| **Forward half-strand** | 207901 | 211607 | 247525 | 244722 |
| **Difference** | +11617 | -9973 | -3562 | +1919 |

Рис. 3.1. Количество нуклеотидов генома *Thermotoga petrophila*.

Хоть частоты A и T практически одинаковы на двух полунитях, C более часто встречается на обратной полунити, чем на прямой полунити, что приводит к разнице в 219518 – 207901 = +11617. Его комплементарный нуклеотид G реже встречается на обратной полунити, чем на прямой полунити, что приводит к разнице в 201634 – 211607 = –9973.

Оказывается, эти расхождения наблюдаются, поскольку цитозин (C) имеет тенденцию мутировать в тимин (T) через процесс, называемый дезаминированием. Скорость дезаминирования возрастает в 100 раз, когда ДНК является одноцепочечной, что приводит к уменьшению цитозина на прямой полунити. Кроме того, поскольку пары оснований C-G в конечном итоге меняются на пары оснований T-A, дезаминирование приводит к наблюдаемому уменьшению гуанина (G) на обратной полунити (прямая родительская полунить синтезирует обратную дочернюю полунить и наоборот).

Посмотрим, можно ли воспользоваться этими статистическими данными, вызванными дезаминированием, чтобы найти *oriC*. Как показывает таблица, разница между общим количеством гуанина и общим количеством цитозина отрицательна на обратной полунити (201634 – 219518 = –17884) и положительна на прямой полунити (211607 – 207901 = 3706). Таким образом, идея состоит в том, чтобы пройти весь геном, сохраняя текущую общую разницу между значениями G и C. Если эта разница начнет увеличиваться, то предполагается, что мы находимся на прямой полунити; с другой стороны, если эта разница начинает уменьшаться, то предполагается, что мы находимся на обратной полунити.

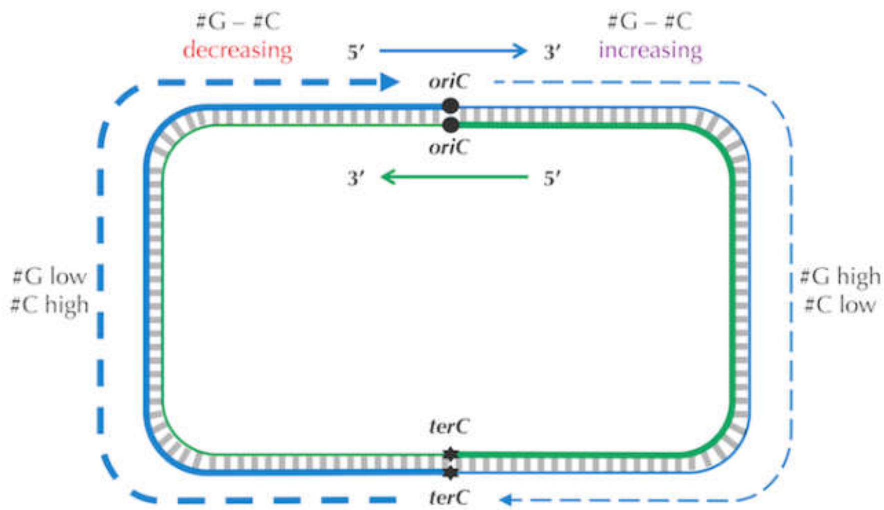



Рис. 3.2. Изменение G и C при движении по прямой и обратной полунити.

Поскольку местоположение *oriC* в циклическом геноме неизвестно, линеаризуем его (т.е. выберем произвольную позицию и постулируем, что в ней начинается геном), в результате чего получается линейная цепочка *Genome*. Определим $Skew_i(Genome)$ как разность между общим числом вхождений G и общим числом вхождений C в первые $i$ нуклеотидов *Genome*. Диаграмма *Skew* определяется как график $Skew_i(Genome)$ ($i$ изменяется от 0 до $|Genome|$), где $Skew_0(Genome)$ устанавливается равным нулю. На рисунке 3.3 показана диаграмма *Skew* для строки ДНК CATGGGCATCGGCCATACGCC.

Следует обратить внимание, что можно вычислить $Skew_{i+1}(Genome)$ по $Skew_i(Genome)$ в соответствии с нуклеотидом в положении $i$ строки *Genome*. Если этот нуклеотид равен G, то $Skew_{i+1}(Genome) = Skew_i(Genome) + 1$; если этот нуклеотид равен C, то $Skew_{i+1}(Genome) = Skew_i(Genome) – 1$; в противном случае $Skew_{i+1}(Genome) = Skew_i(Genome)$.

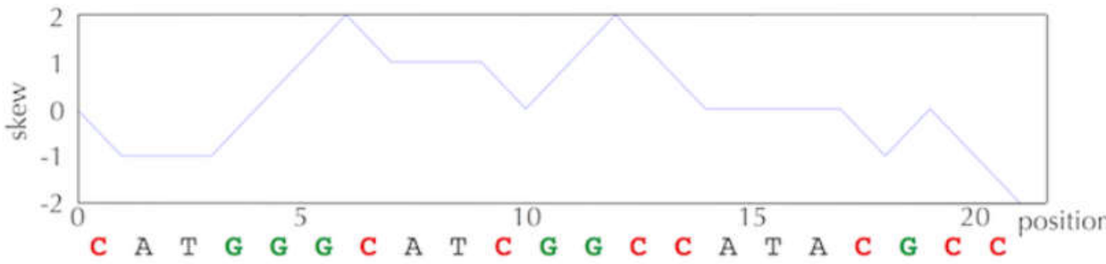



Рис. 3.3. Диаграмма *Skew* для строки ДНК CATGGGCATCGGCCATACGCC.

Теперь рассмотрим диаграмму *Skew* для линеаризованного генома *E.coli* (рисунок 3.4). Заметен очень четкий шаблон. Оказывается, что диаграмма *Skew* для многих бактериальных геномов имеет сходную характерную форму.

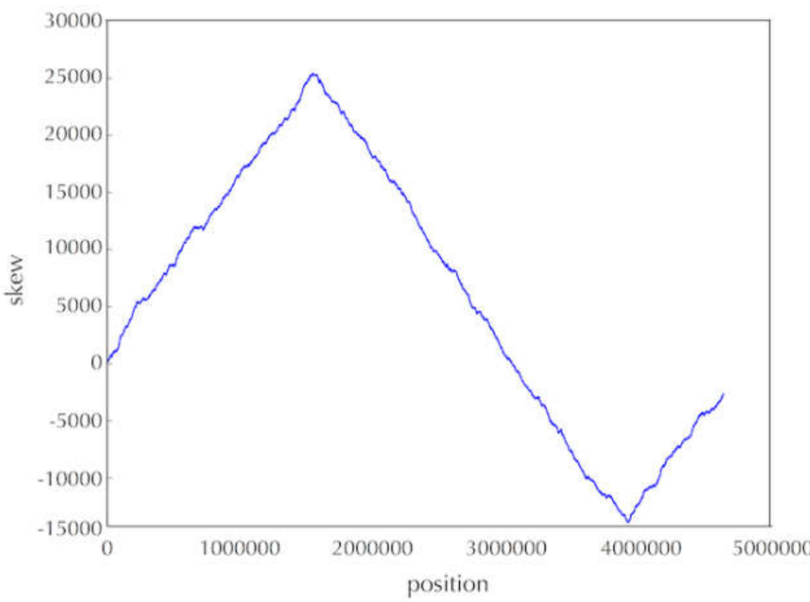



Рис. 3.4. Диаграмма *Skew* для линеаризованного генома *E.coli*.

Будем следовать в направлении 5'→ 3' ДНК и идти по хромосоме сначала от *terC* до *oriC* (вдоль обратной полунити), а затем от *oriC* до *terC* (вдоль прямой полунити). Ранее было показано, что *Skew* убывает вдоль обратной полунити и возрастает вдоль прямой полунити. Таким образом, *Skew* должен достигать минимума в том месте, где начинается обратная полунить и начинается прямая полунить, которая является местом расположения *oriC*.

Таким образом, получается новый алгоритм поиска *oriC*: он должен быть там, где *Skew* достигает минимума.

**Задача 3.1.** Найти позицию в геноме, где диаграмма *Skew* достигает минимума.

Вход: последовательность ДНК *Genome*.

Выход: все целочисленные *i*, минимизирующие $Skew_i(Genome)$ для всех значений *i* (от 0 до |*Genome*|).

Пример входа:

TAAAGACTGCCGAGAGGCCAACACGAGTGCTAGAACGAGGGGCG TAAACGCGGGTCCGAT

Пример выхода:

11 24

Решение задачи *MinimumSkew* теперь дает приблизительное местоположение *oriC* в позиции 3923620 *E.coli*. Чтобы подтвердить эту гипотезу, посмотрим на скрытое сообщение, представляющее потенциальный *DnaA*-бокс рядом с этим местом. Решение задачи *FrequentWords* в окне длины 500, начинающемся с позиции 3923620 (показано ниже), не показывает 9-меров (вместе с их обратными дополнениями), которые появляются три или более раз. Даже если обнаружен *oriC* в *E.coli*, кажется, что до сих пор не обнаружены *DnaA*-боксы, которые начинают репликацию в этой бактерии.

```
aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgc
ataacgcggtatgaaaatggattgaagcccgggccgtggattctactcaa
ctttgtcggcttgagaaagacctgggatcctgggtattaaaaagaagatc
tatttatttagagatctgttctattgtgatctcttattaggatcgcactg
ccctgtggataacaaggatccggcttttaagatcaacaacctggaaagga
tcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcag
aatgaggggttatacacaactcaaaaactgaacaacagttgttctttgga
taactaccggttgatccaagcttcctgacagagttatccacagtagatcg
cacgatctgtatacttatttgagtaaattaacccacgatcccagccattc
ttctgccggatcttccggaatgtcgtgatcaagaatgttgatcttcagtg
```

Рассмотрим *oriC Vibrio cholerae* еще раз, чтобы узнать, дает ли он какие-либо сведения о том, как изменить алгоритм, чтобы найти *DnaA*-боксы в *E.coli*. В дополнение к трем вхождениям ATGATCAAG и трем вхождениям его обратного комплемента CTTGATCAT, *Vibrio cholerae oriC* содержит дополнительные вхождения ATGATCAAC и CATGATCAT, которые отличаются от ATGATCAAG и CTTGATCAT только одним нуклеотидом:

atca**ATGATCAAC**gtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatccacaac<br>
ctgagtggatgacatcaagataggtcgttgtatctccttcctctcgtactctcatgacca<br>
cggaaag**ATGATCAAG**agaggatgatttcttggccatatcgcaatgaatacttgtgactt<br>
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtgacggagcgggatt<br>
acgaaag**CATGATCAT**ggctgttgttctgtttatcttgttttgactgagacttgttagga<br>
tagacggtttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat<br>
tgataatgaatttacatgcttccgcgacgatttacct**CTTGATCAT**cgatccgattgaag<br>
atcttcaattgttaattctcttgcctcgactcatagccatgatgagct**CTTGATCAT**gtt<br>
tccttaaccctctatttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc

Нахождение восьми похожих вхождений целевого 9-мера и его обратного дополнения в короткую область еще более статистически неожиданно, чем поиск шести точных вхождений ATGATCAAG и его обратного комплемента CTTGATCAT, которые были найдены ранее. Кроме того, обнаружение данных приблизительных 9-меров имеет биологический смысл, поскольку *DnaA* может связывать не только «идеальные» *DnaA*-боксы, но и их небольшие вариации.

Говорят, что положение $i$ в $k$-мерах $p_1 \dots p_k$ и $q_1 \dots q_k$ является промахом, если $p_i \neq q_i$. Например, CGAAT и CGGAC имеют два промаха. Число промахов между строками $p$ и $q$ называется расстоянием Хэмминга между этими строками и обозначается как *HammingDistance*($p$, $q$).

**Задача 3.2.** Найти расстояние Хэмминга между двумя строками.

Вход: две строки одинаковой длины.

Выход: расстояние Хэмминга между данным строками.

Пример входа:

GGGCCGTTGGT

GGACCGTTGAC

Пример выхода:

3

Говорят, что $k$-мер *Pattern* появляется как подстрока строки *Text* с не более чем $d$ несоответствиями, если есть некоторый $k$-мер *Pattern'* строки *Text*, имеющий $d$ или меньше промахов с *Pattern*, т.е. *HammingDistance*(*Pattern*, *Pattern'*) $\leq d$, Наблюдение, что *DnaA*-бокс может иметь небольшие вариации, приводит к следующему обобщению задачи *PatternMatching*.

**Задача 3.3.** Найти все примерные вхождения шаблона в строку.

Вход: строки *Pattern* и *Text*, целое число $d$.

Выход: набор целых чисел, разделенных пробелами, указывающих все начальные позиции в *Text*, где *Pattern* входит в виде подстроки с не более, чем $d$ промахами.

Пример входа:

ATTCTGGA

CGCCCGAATCCAGAACGCATTCCCATATTTCGGGACCACTGGCCT CCACGGTACGGACGTCAATCAAAT

3

Пример выхода:

6 7 26 27

Теперь цель – изменить предыдущий алгоритм для задачи *FrequentWords*, чтобы найти *DnaA*-боксы путем нахождения наиболее частых *k*-меров, возможно, с промахами. Для заданных строк *Text* и *Pattern*, а также целого числа $d$ $Count_d$(*Text*, *Pattern*) определяется как общее количество вхождений *Pattern* в *Text* с не более чем $d$ промахами.

Например, $Count_1$ (AACAAGCTGATAAACATTTAAAGAG, AAAAA) = 4, потому что AAAAA появляется четыре раза в этой строке с не более чем одним промахом: AACAA, ATAAA, AAACA и AAAGA. Два из этих совпадений перекрываются.

Вычисление $Count_d$(*Text*, *Pattern*) требует вычисления расстояния Хэмминга между *Pattern* и каждым *k*-мером *Text*. Ниже представлен псевдокод, осуществляющий поиск числа вхождений всех примерных шаблонов в строку.

```
ApproximatePatternCount(Text, Pattern, d)
    count ← 0
    for i ← 0 to |Text| − |Pattern|
        Pattern′ ← Text(i , |Pattern|)
        if HammingDistance(Pattern, Pattern′) ≤ d
            count ← count + 1
    return count
```

**Задача 3.4.** Реализовать *ApproximatePatternCount*.

Вход: строки *Pattern* и *Text*, целое число $d$.

Выход: $Count_d$(*Text*, *Pattern*).

Пример входа:

GAGG

TTTAGAGCCTTCAGAGG

2

Пример выхода:

4

Наиболее частым *k*-мером с не более чем $d$ промахами в *Text* является строка *Pattern*, максимизирующая $Count_d$(*Text*, *Pattern*) для всех *k*-меров. Обратите внимание: *Pattern* не обязательно должен являться подстрокой *Text*; например, как было показано ранее, AAAAA является наиболее частым 5-мером с 1 несоответствием в AACAAGCTGATAAACATTTAAAGAG, хотя его нет в этой строке.

**Задача 3.5.** Найти наиболее часто встречающиеся $k$-меры с учетом промахов в строке.

Вход: строка *Text*, целые числа $k$ и $d$ (предполагается $k \leq 12$, $d \leq 3$).

Выход: все наиболее часто встречающиеся $k$-меры с не более чем $d$ промахами в строке *Text*.

Пример входа:

ACGTTGCATGTCGCATGATGCATGAGAGCT

4 1

Пример выхода:

GATG ATGC ATGT

Один из способов решения вышеуказанной задачи состоит в том, чтобы сгенерировать все $4^k$ $k$-меров *Pattern*, вычислить *ApproximatePatternCount* (*Text*, *Pattern*, *d*) для каждого $k$-мера *Pattern*, а затем вывести $k$-меры с максимальным количеством примерных вхождений. На практике этот подход является неэффективным, поскольку многие из $4^k$ $k$-меров, которые анализирует этот метод, не следует рассматривать, так как ни они, ни их мутированные версии (с точностью до $d$ промахов) не появляются в тексте.

Переопределим задачу *FrequentWords*, чтобы учесть и несоответствия, и обратные комплементы. Напомним, что $\overline{Pattern}$ является обратным комплементом к $Pattern$.

**Задача 3.6.** Найти наиболее часто встречающиеся $k$-меры с учетом промахов и обратной комплементарности в строке.

Вход: строка ДНК *Text*, целые числа $k$ и $d$.

Выход: все $k$-меры *Pattern*, максимизирующие сумму $Count_d(Text, Pattern) + Count_d(Text, \overline{Pattern})$ по всем возможным $k$-мерам.

Пример входа:

ACGTTGCATGTCGCATGATGCATGAGAGCT

4 1

Пример выхода:

ATGT ACAT

Можно предпринять попытку найти *DnaA*-боксы в *E.coli* путем поиска наиболее частых 9-меров с промахами и обратными комплементами в области, предложенной минимумом диаграммы *Skew* как *oriC*. Несмотря на то, что минимум диаграммы для *E.coli* находится в позиции 3923620, не нужно предполагать, что *oriC* находится именно в этом положении ввиду случайных колебаний в *Skew*. Чтобы устранить эту проблему, можно было бы выбрать больший размер окна (например, $L$ = 1000), но расширение окна приведет к риску обнаружить другие 9-меры, которые не представляют собой *DnaA*-боксы, но чаще истинного *DnaA*-бокса появляются в этом окне. Имеет смысл рассмотреть маленькое окно, начинающееся, заканчивающееся или центрированное в позиции минимума *Skew*.

Определим наиболее частые 9-меры (с 1 допустимым промахом и обратными комплементами) в окне длиной 500, начиная с положения 3923620 генома *E.coli*. Экспериментально подтвержденный *DnaA*-бокс в *E.coli* (TTATCCACA) действительно является наиболее частым 9-мером с 1 промахом, а также его обратным комплементом TGTGGATAA:

aatgatgatgacgtcaaaaggatccggataaaacat**ggtgattgcctcgc**<br>
**ataacgcggtatgaaaatggattgaagcccgggccgtggattctactcaa**<br>
**ctttgtcggcttgagaaagacctgggatcctgggtattaaaaagaagatc**<br>
**tatttatttagagatctgttctattgtgatctcttattaggatcgcactg**<br>
**cccTGTGGATAAcaaggatccggcttttaagatcaacaacctggaaagga**<br>
**tcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcag**<br>
**aatgaggggTTATACACAactcaaaaactgaacaacagttgttcTTTGGA**<br>
**TAActaccggttgatccaagcttcctgacagagTTATCCACAgtagatcg**<br>
cacgatctgtatacttatttgagtaaattaacccacgatcccagccattc<br>
ttctgccggatcttccggaatgtcgtgatcaagaatgttgatcttcagtg

Выделенная значительная часть этой последовательности является экспериментально подтвержденной областью *oriC E.coli*, которая начинается через 37 нуклеотидов (выделены красным) после положения 3923620, где *Skew* достигает своего минимального значения.

В данном случае *DnaA*-боксы *E.coli* попадают в окно, которое было выбрано. Более того, несмотря на то, что TTATCCACA представляет собой наиболее частый 9-мер с 1 промахом и обратным комплементом в этом окне из 500 нуклеотидов, он не является единственным: GGATCCTGG, GATCCCAGC, GTTATCCAC, AGCTGGGAT и CTGGGATCA также появляются 4 раза с 1 промахом и обратным комплементом.

Неизвестно, с какой целью (если таковая имеется) эти другие 9-меры располагаются в геноме *E.coli*, но известно, что в геномах существует много разных типов скрытых сообщений; эти скрытые сообщения имеют тенденцию кластеризоваться в геноме, и большинство из них не имеют никакого отношения к репликации. Важным является то, что существующие подходы к прогнозированию *oriC* остаются несовершенными. Однако, предоставление биологам даже небольшого набора претендентов на *DnaA*-боксы – большая помощь.

Несмотря на то, что вычислительные предсказания бывают убедительны, биоинформатики сотрудничают с биологами для проверки расчетных прогнозов или улучшения этих прогнозов.

Ранее было рассмотрено три генома и обнаружено три различных гипотетических 9-мера: ATGATCAAG в *Vibrio cholerae*, CCTACCACC в *Thermotoga petrophila* и TTATCCACA в *E.coli*. На самом деле, поиск *oriC* часто более сложный, чем в трех приведенных примерах. У некоторых бактерий еще меньше *DnaA*-боксов, чем у *E.coli*, что затрудняет их идентификацию. Область *terC* часто расположена не прямо напротив *oriC*, а может быть значительно сдвинута, что приводит к обратным и прямым полунитям, имеющим существенно различную длину. Положение минимума диаграммы *Skew* часто является грубой оценкой положения *oriC* – это заставляет исследователей расширять окна при поиске *DnaA*-боксов, что вносит посторонние повторяющиеся подстроки. Наконец, диаграммы *Skew* не всегда выглядят так же хорошо, как у *E.coli*; например, диаграмма *Skew* для *Thermotoga petrophila* показана на рисунке 3.5.

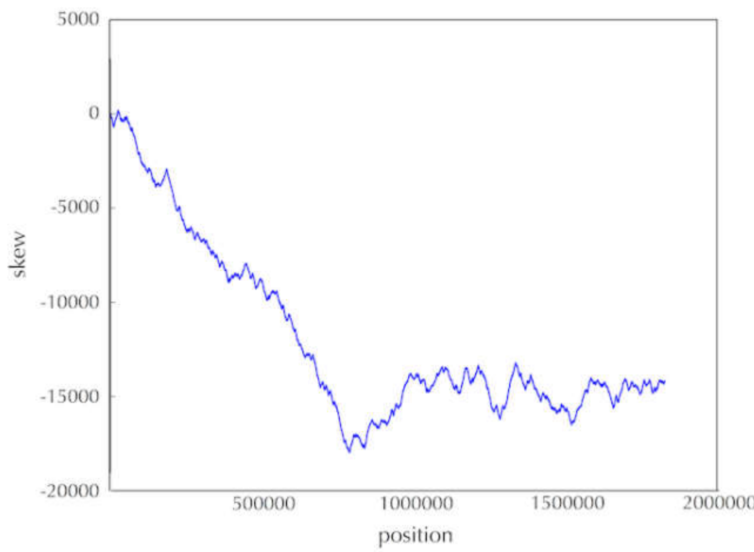



Рис. 3.5. Диаграмма *Skew* для генома *Thermotoga petrophila*.

Долгое время биологи полагали, что каждая бактериальная хромосома имеет только один *oriC*. Wang et al. [9] генетически модифицировали *E. coli*, вставив синтетический *oriC* на расстоянии в миллион нуклеотидов от известного *oriC* бактерии. К их удивлению, *E.coli* продолжали работать как обычно, начав репликацию в обоих местах.

После публикации этой статьи начались поиски бактерий с несколькими *oriC*. Xia, 2012 [10] усомнился в отношении постулата «единственного *oriC*» и привел примеры бактерий с очень необычными *Skew*. Фактически, наличие более одного *oriC* имеет смысл в свете эволюции: если геном длинный, а репликация медленная, то множественные источники репликации уменьшат количество времени, которое бактерия должна провести, реплицируя свою ДНК.

Например, *Wigglesworthia glossinidia*, симбиотическая бактерия, живущая в кишечнике мухи цеце, имеет нетипичную диаграмму *Skew*, показанную на рисунке 3.6. Поскольку эта диаграмма имеет по крайней мере два выраженных локальных минимума, Xia утверждал, что эта бактерия может иметь две или более области *oriC*.

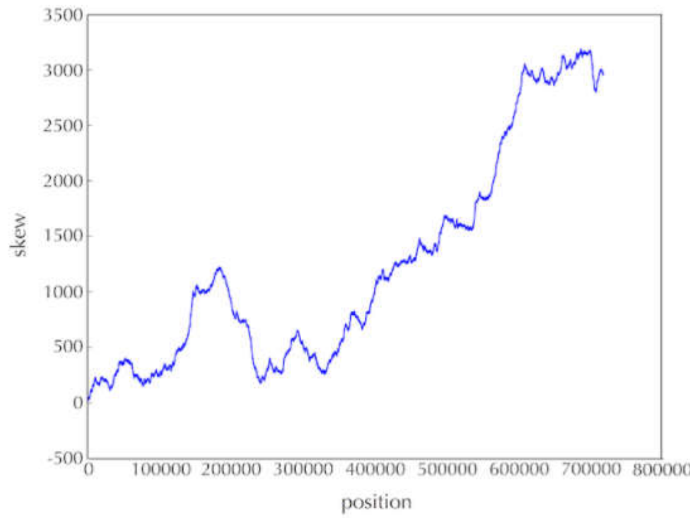



Рис. 3.6. Диаграмма *Skew* для генома *Wigglesworthia glossinidia*.

Нужно быть осторожными с гипотезой о том, что эта бактерия имеет два *oriC*, так как могут быть альтернативные объяснения множественных локальных минимумов в *Skew*. Например, перестановки передвигают гены в геноме и часто переставляют их из прямой в обратную полунить и наоборот, что приводит к неровностям на диаграмме *Skew*. Одним из примеров перестановки генома является обращение, которое переворачивает сегмент хромосомы и переключает его на противоположную нить; на рисунке 3.7 показано, что происходит с диаграммой *Skew* после разворота.

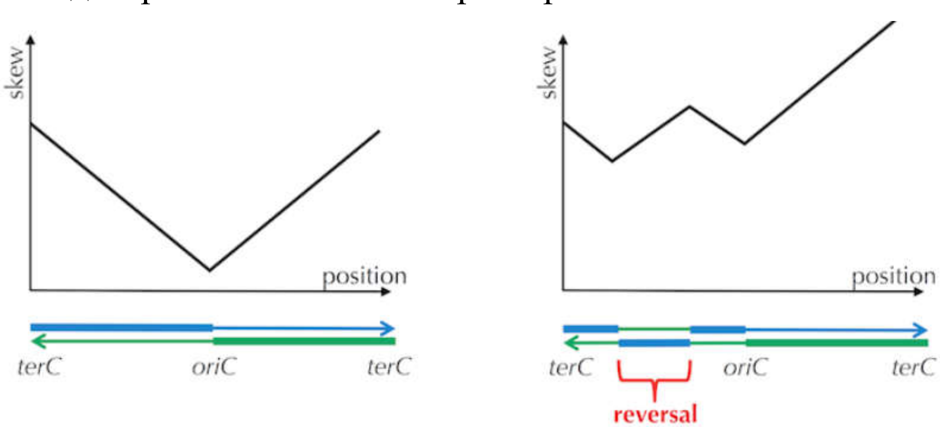



Рис. 3.7. Изменение диаграммы *Skew* при развороте генома.

Другая сложность заключается в том, что различные виды бактерий могут обмениваться генетическим материалом (это явление называется горизонтальным переносом генов). Если ген прямой полунити одной бактерии переносится на противоположную полунить другой (или наоборот), то будет наблюдаться нерегулярность в диаграмме *Skew*. В результате, вопрос о числе областей *oriC Wigglesworthia glossinidia* остается нерешенным.

Археи – это одноклеточные организмы, настолько отличные от других форм жизни, что биологи поставили их отдельно от бактерий и эукариот. Хотя археи визуально похожи на бактерии, у них есть некоторые геномные признаки, которые более тесно связаны с эукариотами. В частности, механизм репликации археи больше похож на эукариотов, чем на бактерии. Однако, археи используют гораздо большее количество источников энергии, чем эукариоты, питающиеся аммиаком, металлами или даже газообразным водородом.

На рисунке 3.8 показана диаграмма *Skew Sulfolobus solfataricus*, вида археи, растущей в кислых вулканических источниках при температурах выше 80° C. В диаграмме можно видеть по меньшей мере три локальных минимума, представленных глубокими ямами, в дополнение ко многим более мелким ямам.

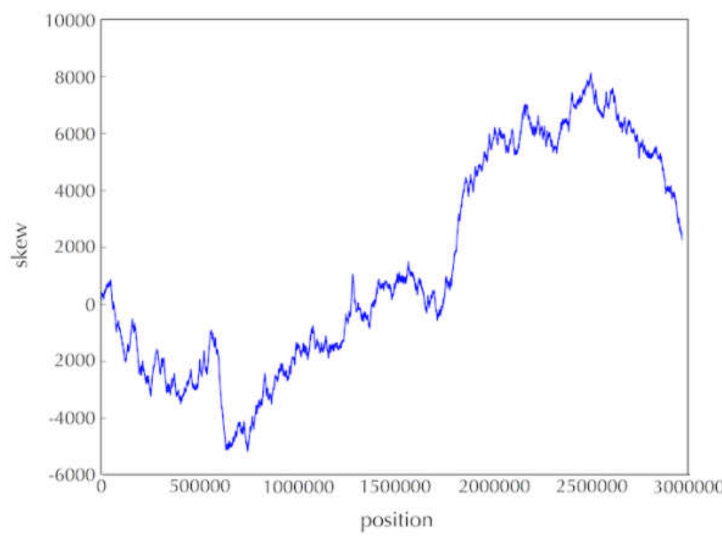



Рис. 3.8. Диаграмма *Skew* для генома *Sulfolobus solfataricus*.

В 2004 году Lundgren et al. [12] экспериментально продемонстрировали, что *Sulfolobus solfataricus* действительно имеет три *oriC*. С тех пор множественные *oriC* были найдены во многих других археях. Однако, не был разработан точный вычислительный подход для идентификации нескольких *oriC* в новом секвенированном геноме археи. Например, метанообразующая архея *Methanococcus jannaschii* считается основоположником генома археи, но ее *oriC* по-прежнему остаются неизвестными. Её диаграмма *Skew* (показана на рисунке 3.9) предполагает, что она может иметь несколько *oriC*.

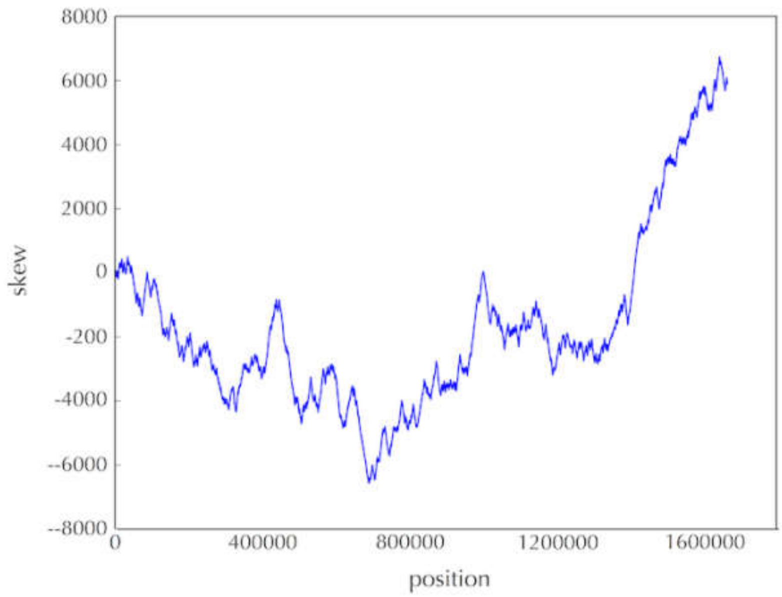



Рис. 3.9. Диаграмма *Skew* для генома *Methanococcus jannaschii*.

Обнаружение источника репликации у бактерий является сложной проблемой, но еще более нетривиальной задачей является поиск источников репликации в более сложных организмах, таких как дрожжи или даже люди, у которых есть сотни источников репликации. Среди различных видов дрожжей почкообразные дрожжи *Saccharomyces cerevisiae* имеют наилучшее описание источников репликации. Они имеют приблизительно 400 различных *oriC*, многие из которых могут использоваться во время репликации любой отдельной дрожжевой клетки.

Наличие большого количества *oriC* приводит к тому, что десятки репликационных вилок движутся друг к другу из разных мест в геноме такими способами, которые еще не полностью поняты. Тем не менее, исследователи обнаружили, что истоки репликации *Saccharomyces cerevisiae* имеют несколько варьирующийся шаблон, называемый ARS Consensus Sequence (ACS). ACS является сайтом связывания для так называемого Origin Recognition Complex, который инициирует загрузку дополнительных белков, необходимых для начала работы. Многие ACS соответствуют следующему каноническому тиминовому паттерну длины 11:

**TTTAT(G/A)TTT(T/A)(G/T)**

Обозначение (X / Y) указывает на то, что или нуклеотид X, или нуклеотид Y могут появиться в этом положении.

Тем не менее, различные ACS могут отличаться от этого канонического шаблона, причем длины варьируются от 11 до 17 нуклеотидов. Например, 11-

нуклеотидный паттерн, показанный выше, часто является частью 17-нуклеотидного шаблона:

`(T/A)(T/A)(T/A)(T/A)`**`TTTAT(G/A)TTT(T/A)(G/T)`**`(T/G)(T/C)`

В последнее время был достигнут определенный прогресс в исследовании ACS у нескольких других видов дрожжей. У некоторых видов, таких как *S.bayanus*, ACS почти тождественен по сравнению с *S.cerevisiae*, тогда как в других, таких как *K.lactis*, он очень отличается по длине (50), а также в нуклеотидной последовательности. У некоторых видов дрожжей, таких как *S.pombe*, Origin Recognition Complex связывается определенными областями, богатыми AT, что делает невозможным поиск источников репликации только на основе анализа последовательности.

## Лекция 4. Алгоритмы полного перебора.

В августе 1928 года, перед отъездом в отпуск, шотландский микробиолог Александр Флеминг поместил свои культуры вызывающих инфекцию бактерий *Staphylococcus* на лабораторный стенд. Когда он вернулся на работу через несколько недель, Флеминг заметил, что одна культура была заражена грибом *Penicillium*, и что колония *Staphylococcus*, окружающая его, была уничтожена. Флеминг назвал вещество, убивающее бактерии, пенициллином, и он предположил, что его можно использовать для лечения бактериальных инфекций у людей.

Когда Флеминг опубликовал свое открытие в 1929 году, его статья не имела немедленного отклика. Последующие эксперименты пытались выделить антибиотический агент (т.е. соединение, которое фактически убило бактерии) из гриба. В результате Флеминг в конце концов пришел к выводу, что пенициллин не может быть применен на практике для лечения бактериальных инфекций и отказался от исследований по антибиотикам.

В процессе поиска новых лекарств для лечения раненых солдат, американские и британские правительства активизировали исследования антибиотиков после начала Второй мировой войны; однако проблема массового производства антибиотиков осталась. В марте 1942 года половина общего количества пенициллина, принадлежащего фармацевтическому гиганту Merck, использовалась для лечения одного инфицированного пациента.

Также в 1942 году российские биологи Георгий Гауз и Мария Бражникова заметили, что бактерия *Bacillus brevis* убила патогенную бактерию *Staphylococcus aureus*. В отличии от усилий Флеминга с пенициллином, они успешно изолировали соединение антибиотиков из *Bacillus brevis* и назвали его

*Gramicidin Soviet*. Через год этот антибиотик был распространен по советским военным госпиталям.

Между тем, американские ученые исследовали различные продовольственные рынки на наличие гнилых продуктов и нашли заплесневелую канталупу в штате Иллинойс с высокой концентрацией пенициллина. Это открытие позволило Соединенным Штатам произвести 2 миллиона доз пенициллина во время вторжения союзников в Нормандию в 1944 году, тем самым спася тысячи раненых солдат.

Гауз продолжил свое исследование *Gramicidin Soviet* после Второй мировой войны, но не смог прояснить его химическую структуру. Эстафету у Гауза принял английский биохимик Ричард Синге, который изучил *Gramicidin Soviet* и множество других антибиотиков, произведенных *Bacillus brevis*. Через несколько лет после окончания Второй мировой войны он продемонстрировал, что они представляют собой короткие аминокислотные последовательности (то есть мини-белки), называемые пептидами. Гауз получил Сталинскую премию в 1946 году, а Синге выиграл Нобелевскую премию в 1952 году. Первая награда оказалась более ценной, поскольку она защищала Гауза от казни в период Лысенкоизма, советской кампании против «буржуазных» генетиков, которая усилилась в послевоенную эпоху.

Массовое производство антибиотиков инициировало эволюционную гонку вооружений между фармацевтическими компаниями и патогенными бактериями. Первые работали над разработкой новых антибиотиков, в то время как вторые вырабатывали резистентность к этим препаратам. Хотя современная медицина выигрывала все битвы в течение шести десятилетий, последние десять лет стали свидетелями тревожного роста устойчивых к антибиотикам бактериальных инфекций, которые нельзя лечить даже самыми мощными антибиотиками. В частности, бактерия *Staphylococcus aureus*, которую Гауз изучил в 1942 году, мутировала в резистентный штамм, известный как устойчивый к метициллину *Methicillin-resistant Staphylococcus aureus* (*MRSA*). *MRSA* сейчас является основной причиной смерти от инфекций в больницах, превышающей даже смертность от СПИДа в Соединенных Штатах.

С появлением *MRSA*, разработка новых антибиотиков представляет собой главный вызов современной медицине. Трудной задачей в исследованиях антибиотиков является секвенирование недавно обнаруженных антибиотиков или определение порядка аминокислот, составляющих антибиотический пептид.

Рассмотрим *Tyrocidine B1*, один из многих антибиотиков, производимых бактерией *Bacillus brevis*. *Tyrocidine B1* определяется последовательностью в 10 аминокислот, показанной ниже (с использованием однобуквенных и трехбуквенных обозначений для аминокислот). Цель – выяснить, как *Bacillus brevis* могла сделать этот антибиотик.

```
Val-Lys-Leu-Phe-Pro-Trp-Phe-Asn-Gln-Tyr
 V   K   L   F   P   W   F   N   Q   Y
```

Центральная догма молекулярной биологии утверждает, что «ДНК делает РНК делает белок». Согласно центральной догме, ген из генома сначала транскрибируется в нить РНК, состоящей из четырех рибонуклеотидов: аденина, гуанина, цитозина и урацила. Нить РНК может быть представлена как строка РНК, образованная над четырехбуквенным алфавитом {A, C, G, U}. Затем транскрипт РНК транслируется в аминокислотную последовательность белка.

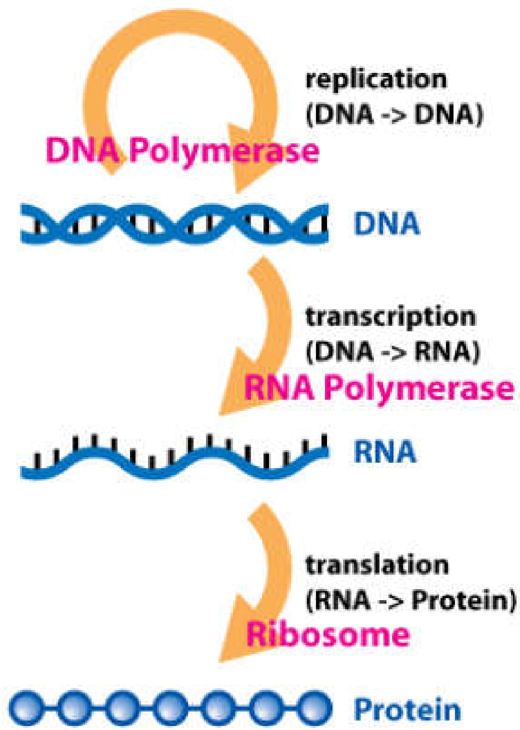



Рис. 4.1. Схематическое представление Центральной догмы молекулярной биологии.

Как и для репликации, химические механизмы, лежащие в основе транскрипции и трансляции с вычислительной точки зрения достаточно просты. Транскрипция преобразует ДНК-цепочку в строку РНК, заменяя все вхождения T на U. Полученная цепь РНК транслируется в аминокислотную последовательность следующим образом: 1) во время трансляции цепь РНК разделяется на неперекрывающиеся 3-меры, называемыми кодонами; 2) каждый кодон превращается в одну из 20 аминокислот в соответствии с генетическим кодом (результирующая последовательность может быть представлена в виде аминокислотной строки, образованной над 20-буквенным алфавитом). Как показано на рисунке 4.2, каждый из 64 кодонов РНК кодирует свою собственную аминокислоту (некоторые кодоны кодируют одну и ту же аминокислоту), за исключением трех стоп-кодонов, которые не кодируют аминокислоты, а служат для остановки процесса трансляции. Например, цепочка ДНК TATACGAAA преобразуется в строку РНК UAUACGAAA, которая, в свою очередь, преобразуется в аминокислотную последовательность Y-T-K.

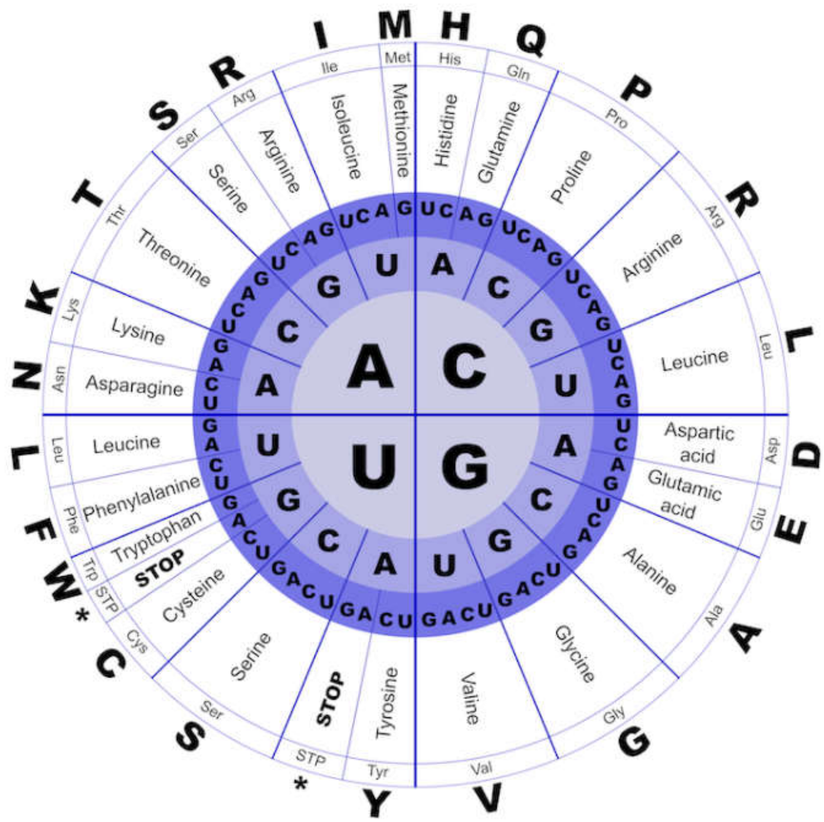



Рис. 4.2. Генетический код описывает трансляцию РНК из 3-меров (кодонов) в одну из 20 различных аминокислот. Первые три круга, при движении изнутри, представляют собой 1-й, 2-й и 3-й нуклеотиды кодона. 4-й, 5-й и 6-й круги определяют переведенную аминокислоту тремя способами: полное имя аминокислоты, ее трехбуквенную аббревиатуру и ее однобуквенную аббревиатуру. Три из 64 общих кодонов РНК представляют собой стоп-кодоны, которые останавливают трансляцию.

В 1961 году Сидней Бреннер и Фрэнсис Крик установили правило «один кодон, одна аминокислота» в процессе трансляции белка. Они отметили, что удаление одного нуклеотида или двух последовательных нуклеотидов в гене резко изменило белковый продукт. Как ни парадоксально, удаление трех последовательных нуклеотидов приводило к незначительным изменениям в белке. Например, фраза

**THE · SLY · FOX · AND · THE · SHY · DOG**

теряет всякий смысл после удаления одной буквы:

THE · SYF · OXA · NDT · HES · HYD · OG

или после удаления двух букв:

THE · SFO · XAN · DTH · ESH · YDO · G

но имеет смысл после удаления трех букв:

THE · FOX · AND · THE · SHY · DOG

В 1964 году Чарльз Янофски показал, что ген и протеин, который он производит, являются коллинеарными, что означает, что первый кодон кодирует первую аминокислоту в белке, второй код кодирует вторую аминокислоту и т.д. В течение следующих тринадцати лет, биологи полагали, что белок кодировался длинной последовательностью смежных нуклеотидных триплетов. Однако открытие разделенных генов в 1977 году показало иное и потребовало решения вычислительной проблемы прогнозирования местоположения генов с использованием только геномной последовательности.

Будем представлять генетический код как массив *GeneticCode*, содержащий 64 элемента, как показано на рисунке 4.3.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | AAA | K | 16 | CAA | Q | 32 | GAA | E | 48 | UAA | * |
| 1 | AAC | N | 17 | CAC | H | 33 | GAC | D | 49 | UAC | Y |
| 2 | AAG | K | 18 | CAG | Q | 34 | GAG | E | 50 | UAG | * |
| 3 | AAU | N | 19 | CAU | H | 35 | GAU | D | 51 | UAU | Y |
| 4 | ACA | T | 20 | CCA | P | 36 | GCA | A | 52 | UCA | S |
| 5 | ACC | T | 21 | CCC | P | 37 | GCC | A | 53 | UCC | S |
| 6 | ACG | T | 22 | CCG | P | 38 | GCG | A | 54 | UCG | S |
| 7 | ACU | T | 23 | CCU | P | 39 | GCU | A | 55 | UCU | S |
| 8 | AGA | R | 24 | CGA | R | 40 | GGA | G | 56 | UGA | * |
| 9 | AGC | S | 25 | CGC | R | 41 | GGC | G | 57 | UGC | C |
| 10 | AGG | R | 26 | CGG | R | 42 | GGG | G | 58 | UGG | W |
| 11 | AGU | S | 27 | CGU | R | 43 | GGU | G | 59 | UGU | C |
| 12 | AUA | I | 28 | CUA | L | 44 | GUA | V | 60 | UUA | L |
| 13 | AUC | I | 29 | CUC | L | 45 | GUC | V | 61 | UUC | F |
| 14 | AUG | M | 30 | CUG | L | 46 | GUG | V | 62 | UUG | L |
| 15 | AUU | I | 31 | CUU | L | 47 | GUU | V | 63 | UUU | F |

Рис. 4.3. Массив *GeneticCode* содержит 64 элемента, каждый из которых представляет собой аминокислоту или стоп-кодон (представлен символом *).

Следующая задача требует нахождения перевода строки РНК в цепочку аминокислот.

**Задача 4.1.** Перевести строку РНК в строку аминокислот.

Вход: строка РНК *Pattern* и массив *GeneticCode*.

Выход: перевод *Pattern* в аминокислотную последовательность *Peptide*.

Пример входа:

AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUA
ACGGGUGA

Пример выхода:

MAMAPRTEINSTRING

Тысячи различных 30-меров ДНК могли бы кодировать *Tyrocidine B1*, и хотелось бы знать, какой из них появляется в геноме *Bacillus brevis*. Существует три разных способа разделить цепочку ДНК на кодоны для трансляции, начиная с каждой из первых трех начальных позиций строки. Эти различные способы деления строки ДНК на кодоны называются рамками считывания. Поскольку ДНК двухцепочечна, у генома есть шесть рамок считывания (по три для каждой нити), как показано на рисунке 4.4.

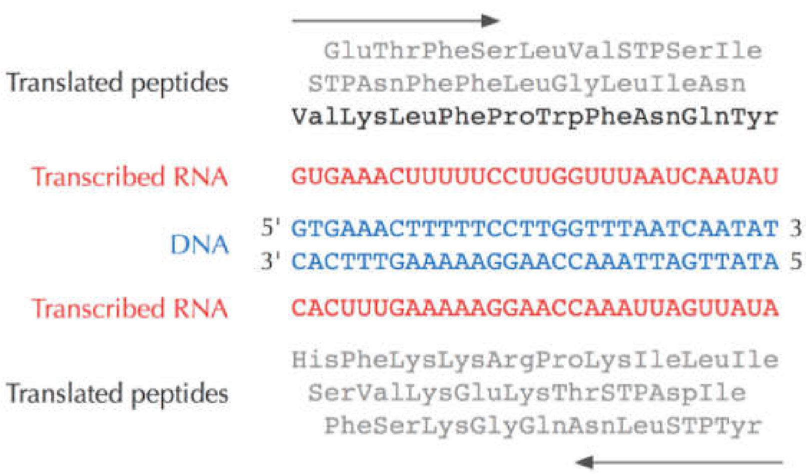



Рис. 4.4. Шесть рамок считывания дают шесть разных способов транскрибирования и транслирования одного и того же фрагмента ДНК. Три верхние аминокислотные строки считываются слева направо, а три нижние строки считываются справа налево. Выделенная аминокислотная строка описывает последовательность *Tyrocidine B1*. Стоп-кодоны представлены как STP.

Строка ДНК *Pattern* кодирует аминокислотную последовательность *Peptide*, если строка РНК, транскрибируемая либо с помощью *Pattern*, либо его обратного комплемента $\overline{Pattern}$, транслируется в *Peptide*. Например, ДНК-последовательность GAAACT транскрибируется в GAAACU и транслируется в ET. Обратный комплемент этой ДНК-строки, AGTTTC, транскрибируется в AGUUUC и транслируется в SF. Таким образом, GAAACT кодирует как ET, так и SF.

**Задача 4.2.** Найти подстроки генома, кодирующие заданную аминокислотную последовательность (*PeptideEncoding*).

Вход: строка ДНК *Text*, аминокислотная строка *Peptide* и массив *GeneticCode*.

Выход: все подстроки строки *Text*, кодирующие *Peptide* (если такие подстроки существуют).

Пример входа:

ATGGCCATGGCCCCCAGAACTGAGATCAATAGTACCCGTATTAAC
GGGTGA

MA

Пример выхода:

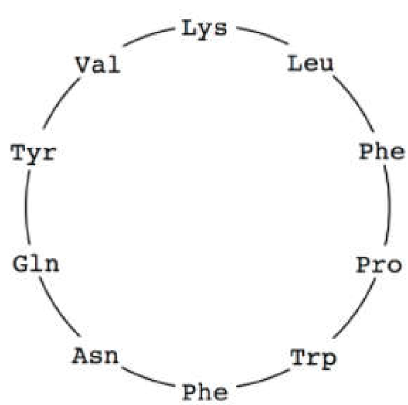
ATGGCC

GGCCAT

ATGGCC

После решения задачи *PeptideEncoding* для *Tyrocidine B1* можно попробовать найти 30-мер в геноме *Bacillus brevis*, кодирующий *Tyrocidine B1*, но такой 30-мер не существует.

Ни Гауз, ни Синге не знали об этом, но тироцидины и грамицидины на самом деле являются циклическими пептидами; циклическое представление для *Tyrocidine B1* показано слева на рисунке 4.5. Таким образом, *Tyrocidine B1* имеет десять различных линейных представлений, и нужно запустить задачу *PeptideEncoding* на каждой из этих последовательностей, чтобы найти потенциальный 30-мер, кодирующий *Tyrocidine B1*. Тем не менее, если попытаться решить задачу *PeptideEncoding* для каждой из десяти строк справа на рисунке 4.5, не обнаружится 30-мера в геноме *Bacillus brevis*, кодирующего *Tyrocidine B1*.




1 Val-Lys-Leu-Phe-Pro-Trp-Phe-Asn-Gln-Tyr
2 Lys-Leu-Phe-Pro-Trp-Phe-Asn-Gln-Tyr-Val
3 Leu-Phe-Pro-Trp-Phe-Asn-Gln-Tyr-Val-Lys
4 Phe-Pro-Trp-Phe-Asn-Gln-Tyr-Val-Lys-Leu
5 Pro-Trp-Phe-Asn-Gln-Tyr-Val-Lys-Leu-Phe
6 Trp-Phe-Asn-Gln-Tyr-Val-Lys-Leu-Phe-Pro
7 Phe-Asn-Gln-Tyr-Val-Lys-Leu-Phe-Pro-Trp
8 Asn-Gln-Tyr-Val-Lys-Leu-Phe-Pro-Trp-Phe
9 Gln-Tyr-Val-Lys-Leu-Phe-Pro-Trp-Phe-Asn
10 Tyr-Val-Lys-Leu-Phe-Pro-Trp-Phe-Asn-Gln

Рис. 4.5. *Tyrocidine B1* представляет собой циклический пептид (слева), и поэтому он имеет десять различных линейных представлений (справа).

Центральная догма молекулярной биологии подразумевает, что все пептиды должны кодироваться геномом, однако, в данном случае это не так. Нобелевский лауреат Эдвард Татум был озадачен, и в 1963 году он провел гениальный эксперимент. Трансляция белков осуществляется молекулярной машиной, называемой рибосомой, и поэтому Татум предполагал, если он ингибирует рибосому, продуцирование белка в *Bacillus brevis* должно прекратиться. К его изумлению, все белки действительно перестали производиться – за исключением тироцидинов и грамицидинов. Этот эксперимент заставил Татума предположить, что какой-то еще неизвестный нерибосомальный механизм должен собирать эти пептиды.

В 1969 году Фриц Липманн (еще один лауреат Нобелевской премии) продемонстрировал, что тироцидины и грамицидины представляют собой нерибосомальные пептиды (*NRP*), синтезированные не рибосомой, а гигантским белком под названием *NRP-synthetase*. Этот фермент объединяет антибиотические пептиды без какой-либо зависимости от РНК или генетического кода. Известно, что каждая *NRP-synthetase* собирает пептиды путем присоединения к ним по одной аминокислоте за раз, как показано на рисунке 4.6.

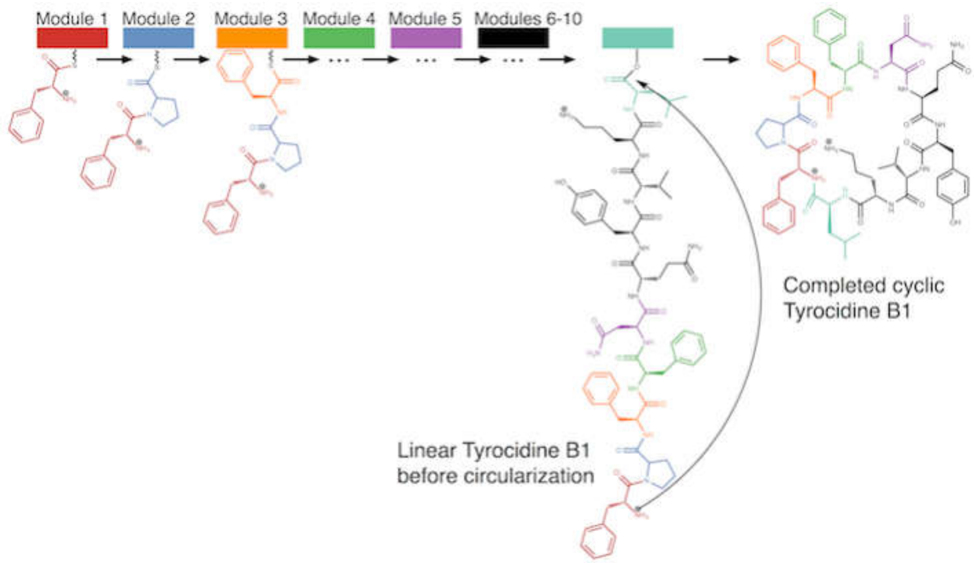



Рис. 4.6. *NRP-synthetase* представляет собой гигантский многомодульный белок, который последовательно собирает циклический пептид, по одной аминокислоте за раз. Каждый из десяти различных модулей (показаны разными цветами) добавляет одну аминокислоту в пептид, который на рисунке является одним из многих тироцидинов, продуцируемых *Bacillus brevis*. На последней стадии пептид является циклическим.

Причина, по которой многие *NRP* имеют фармацевтические применения, состоит в том, что они были разработаны эрами эволюции как «молекулярные пули», которые используют бактерии и грибы, чтобы убить своих врагов. Если эти враги оказываются патогенами, исследователи стремятся заимствовать эти пули в качестве антибактериальных препаратов. Однако *NRP* не ограничиваются антибиотиками: многие из них представляют собой противоопухолевые агенты и иммуноподавители, тогда как другие используют бактерии для связи с другими клетками.

Традиционное представление о том, что бактерии действуют как одиночки и мало взаимодействует с остальной их колонией, было сломано открытием коммуникационного метода, называемого чувством кворума. Это открытие показало, что бактерии способны координировать активность при миграции к лучшему питанию или образованию биопленки для защиты в агрессивных средах. «Язык», используемый при чувстве кворума, часто основан на обмене пептидами (а также другими молекулами), называемых бактериальными феромонами. Характер связи между бактериями может быть дружественным или состязательным.

Когда одна бактерия высвобождает феромоны в окружающую среду, их концентрация часто бывает слишком низкой, чтобы ее можно было обнаружить; однако, как только плотность населения увеличивается, концентрации феромонов достигают порогового уровня, который позволяет бактериям активировать определенные гены в ответ.

Например, *Burkholderia cepacia* – патоген, поражающий людей с кистозным фиброзом. Большинство пациентов, колонизированных *B.cepacia*, также инфицируются *Pseudomonas aeruginosa*. Соотношение двух штаммов у этих пациентов привело к тому, что биологи предположили, что межвидовая связь с *P.aeruginosa* может помочь *B.cepacia* повысить свою собственную патогенность. Действительно, добавление *P.aeruginosa* к клонам *B.cepacia* приводит к значительному увеличению синтеза протеаз (т.е. ферментов, необходимых для разрушения белков), что указывает на наличие чувствительности кворума – *B.cepacia* может извлечь выгоду из сделанных другим видом феромонов, чтобы улучшить свои шансы на выживание.

Поскольку *NRP* не придерживаются Центральной Догмы, их не получится вывести из генома. Что еще более затрудняет секвенирование этих пептидов, так это то, что многие *NRP* (включая тироцидины и грамицидины) являются циклическими. Таким образом, стандартные инструменты для секвенирования линейных пептидов не применимы к анализу *NRP*.

Основу пептидного секвенирования представляет масс-спектрометр, дорогой молекулярный прибор, который разрушает молекулы на части, а затем взвешивает полученные фрагменты. Масс-спектрометр измеряет массу молекулы в дальтонах (Да); 1 Да приблизительно равен массе одной ядерной частицы (т.е. протона или нейтрона).

Аппроксимируем массу молекулы, считая количество протонов и нейтронов, найденных в атомах молекулы, что даст общую массу молекулы. Например, аминокислота глицин, имеющая химическую формулу $C_2H_3ON$, имеет целую массу 57, так как $2 \cdot 12 + 3 \cdot 1 + 1 \cdot 16 + 1 \cdot 14 = 57$. Однако 1 Да не точно равен массе протона / нейтрона, поэтому может потребоваться учитывать различные природные изотопы каждого атома при взвешивании молекулы. В результате аминокислоты обычно имеют нецелые массы (например, глицин имеет общую массу, равную приблизительно 57,02 Да); для простоты, однако, будем работать с таблицей целых масс, приведенной на рисунке 4.7.

| G | A | S | P | V | T | C | I | L | N |
|---|---|---|---|---|---|---|---|---|---|
| 57 | 71 | 87 | 97 | 99 | 101 | 103 | 113 | 113 | 114 |
| **D** | **K** | **Q** | **E** | **M** | **H** | **F** | **R** | **Y** | **W** |
| 115 | 128 | 128 | 129 | 131 | 137 | 147 | 156 | 163 | 186 |

Рис. 4.7. Таблица масс аминокислот.

*Tyrocidine B1*, который представлен последовательностью VKLFPWFNQY, имеет общую массу 1322 Да (99 + 128 + 113 + 147 + 97 + 186 + 147 + 114 + 128 + 163 = 1322).

Масс-спектрометр разрушает каждую молекулу *Tyrocidine B1* на два линейных фрагмента и анализирует образцы, которые могут содержать миллиарды идентичных копий пептида, причем каждая копия разделена по-своему. Один экземпляр может разбиться на LFP и WFNQYVK (с соответствующими массами 357 и 965), тогда как другой может разделиться на PWFN и QYVKLF. Цель – использовать массы этих и других фрагментов для секвенирования пептида. Набор всех масс фрагментов, создаваемых масс-спектрометром, называется экспериментальным спектром.

Будем считать, что масс-спектрометр разрушает копии циклического пептида по всем возможным парам связей, так что полученный экспериментальный спектр содержит массы всех возможных линейных фрагментов пептида, называемых субпептидами. Например, циклический пептид NQEL имеет 12 субпептидов: N, Q, E, L, NQ, QE, EL, LN, NQE, QEL, ELN и LNQ. Также предположим, что субпептиды могут встречаться более одного раза, если аминокислота встречается несколько раз в пептиде (например, ELEL также имеет 12 субпептидов: E, L, E, L, EL, LE, EL, LE, ELE, LEL, ELE и LEL).

**Задача 4.3.** Найти количество субпептидов для циклического пептида заданной длины.

Вход: длина циклического пептида *N*.

Выход: количество субпептидов для циклического пептида длины *N*.

Пример входа:

31315

Пример выхода:

980597910

Теоретический спектр циклического пептида *Peptide*, обозначенного *Cyclospectrum*(*Peptide*), представляет собой совокупность всех масс его субпептидов, а также масса 0 и масса всего пептида. Массы упорядочены от наименьшей к наибольшей. Будем считать, что теоретический спектр может содержать повторяющиеся элементы, как это имеет место для NQEL (показано на рисунке 4.8), где NQ и EL имеют одинаковую массу.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 113 | 114 | 128 | 129 | 227 | **242** | **242** | 257 | 355 | 356 | 370 | 371 | 484 |
| | L | N | Q | E | LN | **NQ** | **EL** | QE | LNQ | ELN | QEL | NQE | NQEL |

Рис. 4.8. Теоретический спектр NQEL.

**Задача 4.4.** Сформировать теоретический спектр циклического пептида.

Вход: аминокислотная строка *Peptide*.

Выход: *Cyclospectrum*(*Peptide*).

Пример входа:

LEQN

Пример выхода:

0 113 114 128 129 227 242 242 257 355 356 370 371 484

Формирование теоретического спектра известного пептида проста, но цель – решить обратную задачу восстановления неизвестного пептида из его экспериментального спектра. Начнем с предположения, что биолог создал идеальный спектр, который совпадает с теоретическим спектром пептида.

**Задача.** Для заданного идеального спектра найти циклический пептид, теоретический спектр которого соответствует экспериментальному спектру.

Вход: коллекция (возможно повторяющихся) целых чисел *Spectrum*, соответствующая идеальному спектру.

Выход: аминокислотная последовательность *Peptide*, для которой *Cyclospectrum*(*Peptide*)=*Spectrum* (если такая последовательность существует).

Будем работать непосредственно с аминокислотными массами, представляя пептид с помощью последовательности целых чисел, обозначающих массы составляющих аминокислот. Например, представим NQEL как 114-128-129-113 и *Tyrocidine B1* (VKLFPWFNQY) как 99-128-113-147-97-186-147-114-128-163. Поэтому заменим алфавит из 20 аминокислот алфавитом из 18 целых чисел, так как две пары аминокислот имеют одну и ту же целую массу (рисунок 4.9).

| G | A | S | P | V | T | C | I/L | N | D | K/Q | E | M | H | F | R | Y | W |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 57 | 71 | 87 | 97 | 99 | 101 | 103 | **113** | 114 | 115 | **128** | 129 | 131 | 137 | 147 | 156 | 163 | 186 |

Рис. 4.9. Массы аминокислот.

Заметим, что в общем случае (без ограничения аминокислотным алфавитом) задача секвенирования циклопептидов может иметь несколько решений. Например, «пептиды» 1-1-3-3 и 1-2-1-4 имеют одинаковый теоретический спектр $\{1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7\}$.

### Алгоритм полного перебора для секвенирования циклопептида

Рассмотрим простой алгоритм перебора для проблемы секвенирования циклопептидов. Для этого сначала обозначим общую массу аминокислотной последовательности *Peptide* как *Mass*(*Peptide*). В экспериментах, проводимых с помощью масс-спектрометрии, в то время как пептид, сгенерировавший *Spectrum*, неизвестен, масса пептида обычно известна и обозначается *ParentMass*(*Spectrum*). Для простоты будем считать, что для всех экспериментальных спектров *ParentMass*(*Spectrum*) равен наибольшей массе в *Spectrum*.

Алгоритм полного перебора для секвенирования циклопептида *BFCyclopeptideSequencing* генерирует все возможные пептиды, масса которых равна *ParentMass*(*Spectrum*), а затем проверяет, какой из этих пептидов имеет теоретический спектр, соответствующий *Spectrum*.

```
BFCyclopeptideSequencing(Spectrum)
    for every peptide with Mass(Peptide) equal to ParentMass(Spectrum)
        if Spectrum = Cyclospectrum(Peptide)
            output Peptide
```

*BFCyclopeptideSequencing* решает задачу *CyclopeptideSequencing*. Однако, нужно рассмотреть его время работы: сколько пептидов имеет массу, равную *ParentMass*(*Spectrum*).

**Задача 4.5.** Вычислить количество пептидов с заданной массой.

Вход: целое число *m*.

Выход: количество линейных пептидов, имеющих массу *m*.

Пример входа:

1024

Пример выхода:

14712706211

Оказывается, что триллионы пептидов имеют ту же целую массу (1322), что и *Tyrocidine B1* (рисунок 4.10). Поэтому алгоритм *BFCyclopeptideSequencing* непрактичен.

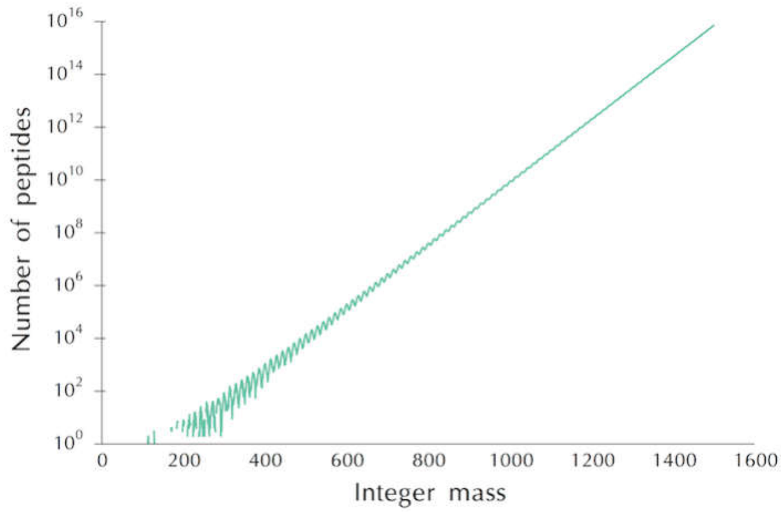



Рис. 4.10. Количество пептидов с заданной целой массой растет экспоненциально.

## Лекция 5. Алгоритм ветвей и границ.

Несмотря на то, что рассмотренный алгоритм перебора потерпел неудачу, это не означает, что все алгоритмы перебора обречены.

Вместо того, чтобы проверять все циклические пептиды с заданной массой, новым подходом к решению проблемы секвенирования циклопептидов будет «выращивание» потенциальных линейных пептидов, теоретические спектры которых «согласуются» с экспериментальным спектром.

Для заданного экспериментального спектра *Spectrum* сформируем набор *Peptides* потенциальных линейных пептидов, первоначально состоящих из пустого пептида, который представляет собой просто пустую строку (обозначенную "") с массой 0. На следующем шаге будем расширять *Peptides*, чтобы они содержали все линейные пептиды длиной 1. Продолжим этот процесс, создавая 18 новых пептидов длиной $k$+1 для каждой аминокислотной последовательности *Peptide* длиной $k$ в *Peptides*, добавляя все возможные аминокислотные массы до конца *Peptide*.

Чтобы количество пептидов-кандидатов не увеличивалось экспоненциально, каждый раз, когда расширяется *Peptides*, будем обрезать данный массив, сохраняя только те линейные пептиды, которые остаются совместимыми с экспериментальным спектром. Проверяем, имеет ли какой-либо из этих новых линейных пептидов массу, равную *Mass*(*Spectrum*). Если так, зациклим этот пептид и проверим, обеспечивает ли он решение *CyclopeptideSequencing*.

В более общем плане алгоритмы перебора, которые проверяют всех кандидатов, но отбрасывают большие подмножества безнадежных кандидатов с использованием различных условий согласованности, называются алгоритмами ветвей и границ. Каждый такой алгоритм состоит из шага ветвления для увеличения количества решений-кандидатов, за которым следует шаг удаления безнадежных кандидатов. В рассмотренном алгоритме ветвей и границ для задачи *CyclopeptideSequencing* шаг ветвления будет расширять каждый пептид-кандидат длиной $k$ до 18 пептидов длиной $k$+1, а шаг ограничения будет устранять несогласованные пептиды из рассмотрения.

Спектр линейного пептида не содержит также много масс, как спектр циклического пептида с той же аминокислотной последовательностью. Например, теоретический спектр циклического пептида NQEL содержит 14 масс (соответствующих "", N, Q, E, L, LN, NQ, QE, EL, ELN, LNQ, NQE, QEL и NQEL). Теоретический спектр линейного пептида NQEL, показанный на рисунке 5.1, не содержит масс, соответствующих LN, LNQ или ELN, поскольку эти субпептиды «оборачивают» конец линейного пептида.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 113 | 114 | 128 | 129 | 242 | 242 | 257 | 370 | 371 | 484 |
| "" | L | N | Q | E | NQ | EL | QE | QEL | NQE | NQEL |

Рис. 5.1. Теоретический спектр линейного пептида NQEL.

Для заданного экспериментального спектра *Spectrum* циклического пептида, линейный пептид согласуется со *Spectrum*, если каждая масса в его теоретическом спектре содержится в *Spectrum*. Если масса появляется в теоретическом спектре линейного пептида более одного раза, то она должна появляться, по крайней мере, столько же раз в *Spectrum*, чтобы линейный пептид соответствовал *Spectrum*. Например, линейный пептид может все же соответствовать теоретическому спектру NQEL, если спектр пептида содержит 242 дважды. Но он не может соответствовать теоретическому спектру NQEL, если его спектр содержит 113 дважды.

Ключом к новому алгоритму является то, что каждый линейный субпептид циклического пептида *Peptide* согласуется с *Cyclospectrum*(*Peptide*). Таким образом, для решения проблемы секвенирования циклопептида для *Spectrum* можно безопасно запретить все пептиды, которые не соответствуют *Spectrum*, из растущего набора, что обеспечивает ограничивающий шаг, описанный выше.

Например, линейный пептид VKF (со спектром {0, 99, 128, 147, 227, 275, 374}) будет запрещен, поскольку он не согласуется с спектром *Tyrocidine B1*. Линейный пептид VKY не будет запрещен, потому что каждая масса в его теоретическом спектре ({0, 99, 128, 163, 227, 291, 390}) присутствует в спектре *Tyrocidine B1*.

Для данного текущего набора линейных пептидов *Peptides*, определим *Expand*(*Peptides*) как новый набор, содержащий все возможные расширения пептидов в *Peptides* при добавлении одной массы аминокислоты. Теперь можно предоставить псевдокод для алгоритма ветвей и границ, называемого *CyclopeptideSequencing*.

```
CYCLOPEPTIDESEQUENCING(Spectrum)
    Peptides ← a set containing only the empty peptide
    while Peptides is nonempty
        Peptides ← Expand(Peptides)
        for each peptide Peptide in Peptides
            if Mass(Peptide) = ParentMass(Spectrum)
                if Cyclospectrum(Peptide) = Spectrum
                    output Peptide
                remove Peptide from Peptides
            else if Peptide is not consistent with Spectrum
                remove Peptide from Peptides
```

**Задача 5.1.** Найти все линейные пептиды, теоретический спектр которых соответствует экспериментальному спектру.

Вход: коллекция (возможно повторяющихся) целых чисел *Spectrum*.

Выход: набор линейных последовательностей *Peptides*, согласующихся со *Spectrum*.

Пример входа:

0 113 128 186 241 299 314 427

Пример выхода:

186-128-113 186-113-128 128-186-113 128-113-186 113-186-128 113-128-186

Трудно представить себе худший сценарий, в котором *CyclopeptideSequencing* работает долго, но никто не может гарантировать, что этот алгоритм не будет генерировать огромное количество неправильных *k*-меров на промежуточных этапах. Алгоритм *BFCyclopeptideSequencing* является экспоненциальным, и, хотя на практике *CyclopeptideSequencing* намного быстрее, этот алгоритм не доказал свою полиномиальность. Таким образом, с точки зрения курса вводных алгоритмов, ориентированных на теоретические компьютерные науки, практический алгоритм *CyclopeptideSequencing* столь же неэффективен, как и *BFCyclopeptideSequencing*, поскольку время работы ни одного из представленных алгоритмов не может быть ограничено полиномом.

Можно запустить *CyclopeptideSequencing* на следующем *Spectrum*:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 97 | 97 | 99 | 101 | 103 | 196 | 198 | 198 | 200 | 202 |
| 295 | 297 | 299 | 299 | 301 | 394 | 396 | 398 | 400 | 400 | 497 |

*CyclopeptideSequencing* сначала расширяет набор *Peptides* до множества всех 1-меров, соответствующих *Spectrum*:

| | | | |
|---|---|---|---|
| 97 | 99 | 101 | 103 |
| P | V | T | C |

Затем алгоритм добавляет каждую из 18 аминокислотных масс к каждому из 1-меров выше. Полученное множество *Peptides*, содержащее 4·18=72 пептидов длины 2, обрезают, чтобы сохранить только десять пептидов, которые соответствуют *Spectrum*:

| | | | | |
|---|---|---|---|---|
| 97-99 | 97-101 | 97-103 | 99-97 | 99-101 |
| PV | PT | PC | VP | VT |
| | | | | |
| 99-103 | 101-97 | 101-99 | 103-97 | 103-99 |
| VC | TP | TV | CP | CV |

После расширения и обрезания, на следующей итерации набор пептидов содержит 15 последовательных 3-меров:

| | | | | |
|---|---|---|---|---|
| 97-99-103 | **97-99-101** | 97-101-97 | **97-101-99** | 97-103-99 |
| PVC | **PVT** | PTP | **PTV** | PCV |
| | | | | |
| **99-97-103** | 99-97-101 | **99-101-97** | 99-103-97 | 101-97-99 |
| **VPC** | VPT | **VTP** | VCP | TPV |
| | | | | |
| 101-97-103 | **101-99-97** | 103-97-101 | **103-97-99** | 103-99-97 |
| TPC | **TVP** | CPT | **CPV** | CVP |

После еще одной итерации набор *Peptides* содержит десять последовательных 4-меров. Шесть 3-меров, выделенных красным, не смогли расширить ни одного 4-мера ниже, поэтому *CyclopeptideSequencing* может генерировать некоторые неправильные *k*-меры на промежуточных итерациях.

| | | | | |
|---|---|---|---|---|
| 97-99-103-97 | 97-101-97-99 | 97-101-97-103 | 97-103-99-97 | 99-97-101-97 |
| PVCP | PTPV | PTPC | PCVP | VPTP |
| 99-103-97-101 | 101-97-99-103 | 101-97-103-99 | 103-97-101-97 | 103-99-97-101 |
| VCPT | TPVC | TPCV | CPTP | CVPT |

На последней итерации генерируется 10 5-меров:

| | | | | |
|---|---|---|---|---|
| 97-99-103-97-101 | 97-101-97-99-103 | 97-101-97-103-99 | 97-103-99-97-101 | 99-97-101-97-103 |
| PVCPT | PTPVC | PTPCV | PCVPT | VPTPC |
| 99-103-97-101-97 | 101-97-99-103-97 | 101-97-103-99-97 | 103-97-101-97-99 | 103-99-97-101-97 |
| VCPTP | TPVCP | TPCVP | CPTPV | CVPTP |

Все эти линейные пептиды соответствуют одному и тому же циклическому пептиду PVCPT, таким образом, решая проблему *CyclopeptideSequencing*.

Несмотря на то, что алгоритм *CyclopeptideSequencing* успешно реконструировал *Tyrocidine B1*, он работает только в случае идеального спектра, т.е. когда экспериментальный спектр пептида точно совпадает с его теоретическим спектром. Эта негибкость *CyclopeptideSequencing* представляет собой практический барьер, поскольку масс-спектрометры генерируют «шумные» спектры, которые далеки от идеала – они характеризуются наличием ложных и недостающих масс. Ложная масса присутствует в экспериментальном спектре, но отсутствует в теоретическом; недостающая масса присутствует в теоретическом спектре, но отсутствует в экспериментальном.

Например, можно сравнить следующие теоретические и (имитируемые) экспериментальные спектры циклического пептида NQEL (рисунок 5.2). Массы, отсутствующие в экспериментальном спектре, показаны синим цветом, а ложные массы в экспериментальном спектре показаны зеленым цветом.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Theoretical:** | 0 | | 113 | 114 | 128 | 129 | 227 | 242 | 242 | 257 | | 355 | 356 | 370 | 371 | 484 |
| **Experimental:** | 0 | 99 | 113 | 114 | 128 | | 227 | | | 257 | 299 | 355 | 356 | 370 | 371 | 484 |

Рис. 5.2. Теоретический и экспериментальный спектр циклического пептида NQEL.

Особенно показательно в этом примере то, что масса аминокислоты E (129) отсутствует, а масса аминокислоты V (99) ложна; в результате первый шаг *CyclopeptideSequencing* установил бы {V, L, N, Q} как аминокислотный состав кандидатов-пептидов, что неверно. Фактически, любая недостающая или ложная масса вызовет выбрасывание алгоритмом *CyclopeptideSequencing* правильного пептида, поскольку его теоретический спектр отличается от экспериментального.

Для обобщения проблемы секвенирования циклопептидов для обработки зашумленных спектров, необходимо отменить требование, чтобы теоретический спектр кандидата пептида точно соответствовал экспериментальному спектру и вместо этого включить функцию подсчёта, которая будет выбирать пептид, теоретический спектр которого лучше всего соответствует данному экспериментальному спектру. Для заданного циклического пептида *Peptide* и спектра *Spectrum* определим *Score*(*Peptide*, *Spectrum*) как количество масс между *Cyclospectrum*(*Peptide*) и *Spectrum*. Вспоминая пример выше, если *Spectrum* = {0, **99**, 113, 114, 128, 227, 257, **299**, 355, 356, 370, 371, 484}, то *Score*(NQEL, *Spectrum*) = 11.

Функция подсчета должна учитывать множественность общих масс, т.е. сколько раз они встречаются в каждом спектре. Например, предположим, что *Spectrum* является теоретическим спектром NQEL; для этого спектра масса 242 имеет кратность 2. Если 242 имеет кратность 1 в теоретическом спектре *Peptide*, то 242 вносит 1 в *Score*(*Peptide*, *Spectrum*). Если 242 имеет большую множественность в теоретическом спектре *Peptide*, то 242 вносит 2 в *Score*(*Peptide*, *Spectrum*).

**Задача 5.2.** Вычислить оценку циклического пептида по спектру.

Вход: аминокислотная строка *Peptide* и набор целых чисел *Spectrum*.

Выход: *Score*(*Peptide*, *Spectrum*).

Пример входа:

NQEL

0 99 113 114 128 227 257 299 355 356 370 371 484

Пример выхода:

11

Теперь можно переопределить проблему секвенирования циклопептидов для шумных спектров.

**Задача.** Найти циклический пептид, имеющий максимальный *Score* по сравнению с экспериментальным спектром.

Вход: набор целых чисел *Spectrum*.

Выход: циклический пептид *Peptide*, максимизирующий *Score*(*Peptide*, *Spectrum*) по всем пептидам *Peptide* с массой, равной *ParentMass*(*Spectrum*).

Цель – адаптировать алгоритм *CyclopeptideSequencing*, чтобы найти пептид с максимальным *Score*. Этот алгоритм имел строгий ограничивающий шаг, в котором все потенциальные линейные пептиды, имеющие несогласованные спектры, выбрасывались. Например, линейный пептид VKF не согласуется с теоретическим спектром циклического пептида *Tyrocidine B1*. Однако, возможно, не следует запрещать VKF в случае экспериментальных спектров, так как они могут иметь недостающие массы. Таким образом, нужно пересмотреть этот шаг, чтобы включить больше потенциальных линейных пептидов, следя при этом за тем, что количество рассматриваемых пептидов не выходит из-под контроля.

Чтобы ограничить количество рассматриваемых линейных пептидов-кандидатов, заменим *Peptides* на *Leaderboard*, который содержит *N* кандидатов с наивысшим *Score* для дальнейшего расширения. На каждом шаге будем расширять все пептиды-кандидаты, найденные в *Leaderboard*, затем удалять те пептиды, чьи недавно вычисленные *Score* недостаточно высоки, чтобы оставлять их в *Leaderboard*. Эта идея похожа на понятие «*cut*» в турнире по гольфу; после *cut* в следующем раунде могут играть только *N* лучших игроков, поскольку только у них есть разумные шансы на победу.

Следует отметить, что в *cut* должны быть все, кто связан с конкурентом за *N*-е место. Таким образом, *Leaderboard* следует обрезать до *N* линейных пептидов с максимальным *Score*, включая «хвосты», которые могут включать более *N* пептидов. Для заданного списка пептидов *Leaderboard*, спектра *Spectrum* и целого числа *N*, определим *Trim*(*Leaderboard*, *Spectrum*, N*)* как набор лучших *N* линейных пептидов с наивысшим *Score* в *Leaderboard* (включая «хвосты») по отношению к *Spectrum*.

*Score*(*Peptide*, *Spectrum*) в настоящее время оценивает *Peptide* по *Spectrum*, если *Peptide* является циклическим. Однако, чтобы обобщить эту функцию оценки, когда *Peptide* является линейным, можно исключить те субпептиды *Peptide*, которые оборачивают конец строки, в результате чего возникает функция *LinearScore*(*Peptide*, *Spectrum*). Например, если *Spectrum* является экспериментальным спектром NQEL, то можно проверить, что *LinearScore*(NQEL, *Spectrum*) = 8.

Введем *LeaderboardCyclopeptideSequencing*.

```
LEADERBOARDCYCLOPEPTIDESEQUENCING(Spectrum, N)
    Leaderboard ← set containing only the empty peptide
    LeaderPeptide ← empty peptide
    while Leaderboard is non-empty
        Leaderboard ← Expand(Leaderboard)
        for each Peptide in Leaderboard
            if Mass(Peptide) = ParentMass(Spectrum)
                if Score(Peptide, Spectrum) > Score(LeaderPeptide, Spectrum)
                    LeaderPeptide ← Peptide
            else if Mass(Peptide) > ParentMass(Spectrum)
                remove Peptide from Leaderboard
        Leaderboard ← Trim(Leaderboard, Spectrum, N)
    output LeaderPeptide
```

**Задача 5.3.** Реализовать *LeaderboardCyclopeptideSequencing*.

Вход: целое число *N* и набор целых чисел *Spectrum*.

Выход: *LeaderPeptide* после запуска *LeaderboardCyclopeptideSequencing*.

Пример входа:

10

0 71 113 129 147 200 218 260 313 331 347 389 460

Пример выхода:

113-147-71-129

Поскольку линейный пептид, приводящий к циклическому пептиду с наивысшим *Score*, может быть удален из таблицы лидеров, алгоритм *LeaderboardCyclopeptideSequencing* – это эвристика, не гарантирующая правильного решения проблемы *CyclopeptideSequencing*.

Рассмотрим смоделированный спектр $Spectrum_{10}$ *Tyrocidine B1*, показанный ниже, который содержит приблизительно 10% **недостающих** / **ложных** масс. Синие массы фактически не находятся в спектре, но они отображены, чтобы было ясно, какие массы отсутствуют.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 97 | 99 | **113** | 114 | **128** | 128 | 147 | 147 | 163 | 186 | 227 | 241 | 242 |
| 244 | 260 | 261 | 262 | 283 | 291 | 333 | 340 | 357 | **385** | **388** | 389 | 390 | 390 |
| 405 | 430 | 430 | 447 | 485 | 487 | 503 | 504 | 518 | 543 | 544 | 552 | 575 | 577 |
| 584 | **631** | 632 | 650 | 651 | 671 | 672 | 690 | 691 | 738 | 745 | 747 | 770 | 778 |
| 779 | 804 | 818 | 819 | **820** | 835 | 837 | 875 | **892** | 892 | 917 | 932 | 932 | 933 |
| 934 | 965 | 982 | 989 | **1030** | **1031** | 1039 | 1060 | 1061 | 1062 | 1078 | 1080 | 1081 | 1095 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1136 | 1159 | 1175 | 1175 | 1194 | 1194 | 1208 | 1209 | 1223 | 1225 | 1322 |

Применение *LeaderboardCyclopeptideSequencing* к этому спектру (с $N = 1000$) приводит к правильному циклическому пептиду VKLFPWFNQY, который имеет *Score* 86.

До сих пор *LeaderboardCyclopeptideSequencing* работал достаточно хорошо, но по мере увеличения числа ошибок, вероятность того, что этот алгоритм вернет неправильный пептид, увеличивается. Проверим этот алгоритм на более шумном моделированном спектре; ниже показан $Spectrum_{25}$ для *Tyrocidine B1*, который имеет 25% **недостающих** / **ложных** масс.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 97 | 99 | 113 | 114 | **115** | 128 | 128 | 147 | 147 | 163 | 186 | 227 | 241 |
| 242 | **244** | 244 | **256** | 260 | 261 | 262 | 283 | 291 | **309** | **330** | 333 | 340 | **347** |
| **357** | **385** | 388 | 389 | 390 | 390 | 405 | **430** | **430** | **435** | 447 | 485 | 487 | 503 |
| 504 | 518 | **543** | 544 | 552 | 575 | 577 | 584 | **599** | **608** | 631 | 632 | 650 | 651 |
| **653** | **671** | 672 | 690 | 691 | **717** | 738 | 745 | **747** | 770 | **778** | 779 | 804 | 818 |
| 819 | **827** | 835 | 837 | 875 | 892 | 892 | 917 | 932 | 932 | 933 | 934 | 965 | 982 |
| 989 | **1031** | 1039 | 1060 | **1061** | 1062 | 1078 | 1080 | 1081 | 1095 | 1136 | 1159 | 1175 | 1175 |
| 1194 | 1194 | 1208 | 1209 | 1223 | **1225** | 1322 | | | | | | | |

При запуске на $Spectrum_{25}$, *LeaderboardCyclopeptideSequencing* (с $N = 1000$) идентифицирует VKLFP**AD**FNQY (*Score*: 83) как лучший циклический пептид вместо правильного пептида VKLFP**W**FNQY (*Score*: 82). Эти два пептида схожи – это связано с тем, что объединенная масса A (71) и D (115) равна массе W (186).

Хоть правильный и неправильный пептиды схожи, их аминокислотные наборы отличаются. Если бы можно было выяснить аминокислотный состав *Tyrocidine B1* из его спектра и запустить *LeaderboardCyclopeptideSequencing* на этом меньшем алфавите (а не на алфавите из всех аминокислот), то можно было бы исключить неправильный пептид VKLFP**AD**FNQY из рассмотрения.

До сих пор предполагалось, что только 20 аминокислот образуют строительные блоки белков; эти строительные блоки называются протеиногенными аминокислотами. На самом деле существуют две дополнительные протеиногенные аминокислоты, селеноцистеин и пирролизин, которые встраиваются в белки специальными механизмами биосинтеза.

Селеноцистеин является протеиногенной аминокислотой, которая существует во всех царствах жизни как строительный блок особого класса белков, называемых селенопротеинами. В отличие от других аминокислот, селеноцистеин непосредственно не кодируется в генетическом коде. Вместо этого он кодируется особым образом кодоном UGA (который обычно является стоп-кодоном) через механизм, известный как трансляционная перекодировка.

Пирролизин представляет собой протеиногенную аминокислоту, которая существует в некоторых археях и метанобразующих бактериях. В организмах, содержащих пирролизин, эта аминокислота кодируется UAG, которая также обычно действует как стоп-кодон.

Помимо 22 протеиногенных аминокислот, *NRP* содержат непротеиногенные аминокислоты, которые расширяют количество возможных строительных блоков для пептидов антибиотиков от 20 до более 100.

Увеличение размера аминокислотных алфавитов является проблемой для текущего подхода к секвенированию циклопептидов. Правильный пептид теперь должен «конкурировать» со многими более неправильными за места в таблице лидеров, увеличивая вероятность того, что правильный пептид будет удален в процессе работы.

Например, *Tyrocidine B1* содержит только протеиногенные аминокислоты, а его близкий родственник, *Tyrocidine B* (Val-**Orn**-Leu-Phe-Pro-Trp-Phe-Asn-Gln-Tyr), содержит непротеиногенную аминокислоту, называемую орнитином (**Orn**).

При применении *LeaderboardCyclopeptideSequencing* для расширенного алфавита к $Spectrum_{10}$, одним из наиболее высокочувствительных пептидов является VKLFPWFNQ**XZ**, где X имеет массу 98, а Z имеет массу 65. Скорее всего, нестандартные аминокислоты конкурировали со стандартными аминокислотами за ограниченное количество позиций в таблице лидеров, в результате чего VKLFPWFNQ**XZ** выигрывает у правильного пептида VKLFPWFNQ**Y**. Поскольку алгоритм *LeaderboardCyclopeptideSequencing* не может идентифицировать правильный пептид даже с 10% ложных и отсутствующих масс, заявленная цель ранее теперь еще более важна. Нужно определить аминокислотный состав пептида по его спектру, чтобы можно было использовать *LeaderboardCyclopeptideSequencing* на этом небольшом алфавите аминокислот.

Одним из способов определения аминокислотного состава пептида из его экспериментального спектра могло быть взятие наименьших масс, присутствующих в спектре (от 57 до 200 Да). Однако, если отсутствует хотя бы одна аминокислотная масса, этот подход не сможет восстановить аминокислотную композицию пептида.

Рассмотрим другой подход. Скажем, что экспериментальный спектр содержит массы субпептидов NQE и NQ. Если вычесть эти две массы, то получим массу E, даже если она не будет присутствовать в экспериментальном спектре. Если исходный пептид NQEL, то также можно найти массу E, вычитая массы QE и Q или NQEL и LNQ.

Следуя этому примеру, определим свертку спектра как все положительные разности масс в спектре. Таблицы далее (рисунки 5.3, 5.4) показывают свертки теоретических и имитируемых спектров NQEL.

| | "" 0 | L 113 | N 114 | Q 128 | E 129 | LN 227 | NQ 242 | EL 242 | QE 257 | LNQ 355 | ELN 356 | QEL 370 | NQE 371 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | | | | | | | | | | | | | |
| **113** | 113 | | | | | | | | | | | | |
| **114** | 114 | 1 | | | | | | | | | | | |
| **128** | 128 | 15 | 14 | | | | | | | | | | |
| **129** | 129 | 16 | 15 | 1 | | | | | | | | | |
| **227** | 227 | 114 | 113 | 99 | 98 | | | | | | | | |
| **242** | 242 | 129 | 128 | 114 | 113 | 15 | | | | | | | |
| **242** | 242 | 129 | 128 | 114 | 113 | 15 | | | | | | | |
| **257** | 257 | 144 | 143 | 129 | 128 | 30 | 15 | 15 | | | | | |
| **355** | 355 | 242 | 241 | 227 | 226 | 128 | 113 | 113 | 98 | | | | |
| **356** | 356 | 243 | 242 | 228 | 227 | 129 | 114 | 114 | 99 | 1 | | | |
| **370** | 370 | 257 | 256 | 242 | 241 | 143 | 128 | 128 | 113 | 15 | 14 | | |
| **371** | 371 | 258 | 257 | 243 | 242 | 144 | 129 | 129 | 114 | 16 | 15 | 1 | |
| **484** | 484 | 371 | 370 | 356 | 355 | 257 | 242 | 242 | 227 | 129 | 128 | 114 | 113 |

Рис. 5.3. Спектральная свертка для теоретического спектра NQEL. Наиболее частыми элементами свертки между 57 и 200 являются (кратности в круглых скобках): 113 (8), 114 (8), 128 (8), 129 (8).

Как было предсказано, некоторые из значений в этих таблицах появляются чаще, чем другие. Например, 113 (масса L) имеет кратность 8; 113 имеет кратность 8. Шесть из восьми вхождений 113 в приведенной выше таблице соответствуют парам субпептидов, различающимся по L: L и ""; LN и N; EL и E; LNQ и NQ; QEL и QE; NQEL и NQE.

| | "" 0 | false 99 | L 113 | N 114 | Q 128 | LN 227 | QE 257 | false 299 | LNQ 355 | ELN 356 | QEL 370 | NQE 371 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | | | | | | | | | | | | |
| **99** | 99 | | | | | | | | | | | |
| **113** | 113 | 14 | | | | | | | | | | |
| **114** | 114 | 15 | 1 | | | | | | | | | |
| **128** | 128 | 29 | 15 | 14 | | | | | | | | |
| **227** | 227 | 128 | 114 | 113 | 99 | | | | | | | |
| **257** | 257 | 158 | 144 | 143 | 129 | 30 | | | | | | |
| **299** | 299 | 200 | 186 | 185 | 171 | 72 | 42 | | | | | |
| **355** | 355 | 256 | 242 | 241 | 227 | 128 | 98 | 56 | | | | |
| **356** | 356 | 257 | 243 | 242 | 228 | 129 | 99 | 57 | 1 | | | |
| **370** | 370 | 271 | 257 | 256 | 242 | 143 | 113 | 71 | 15 | 14 | | |
| **371** | 371 | 272 | 258 | 257 | 243 | 144 | 114 | 72 | 16 | 15 | 1 | |
| **484** | 484 | 385 | 371 | 370 | 356 | 257 | 227 | 185 | 129 | 128 | 114 | 113 |

Рис. 5.4. Спектральная свертка для моделируемого спектра NQEL. Наиболее частыми элементами свертки между 57 и 200 являются (кратности в круглых скобках): 113 (4), 114 (4), 128 (4), 99 (3), 129 (3).

Интересно, что 129 (масса E) появляется три раза в приведенной выше свертке имитируемого спектра, хотя 129 отсутствовал в самом спектре.

В моделированном спектре для NQEL наиболее частыми элементами свертки в диапазоне от 57 до 200 являются (кратности в круглых скобках): 113 (4), 114 (4), 128 (4), 99 (3), 129 (3).

Обратите внимание, что эти наиболее частые элементы захватывают все четыре аминокислоты в NQEL.

**Задача 5.4.** Вычислить свертку спектра.

Вход: набор целых чисел *Spectrum*.

Выход: список элементов в свертке *Spectrum*. Если элемент имеет кратность *k*, он должен появиться ровно *k* раз.

Пример входа:

0 137 186 323

Пример выхода:

137 137 186 186 323 49

*LeaderboardCyclopeptideSequencing* не удалось восстановить *Tyrocidine B1* из $Spectrum_{10}$ при использовании расширенного алфавита аминокислот. Десять наиболее частых элементов его спектральной свертки в диапазоне от 57 до 200 (с кратностями в круглых скобках):

| | | | | |
|---|---|---|---|---|
| **147** (35) | **128** (31) | **97** (28) | **113** (28) | **114** (26) |
| **186** (23) | **57** (21) | **163** (21) | **99** (18) | **145** (18) |

Этот список отражает все восемь различных аминокислотных масс, составляющих *Tyrocidine B1*, которые окрашены в зеленый в списке выше. На рисунке 5.5 показана свертка $Spectrum_{10}$.

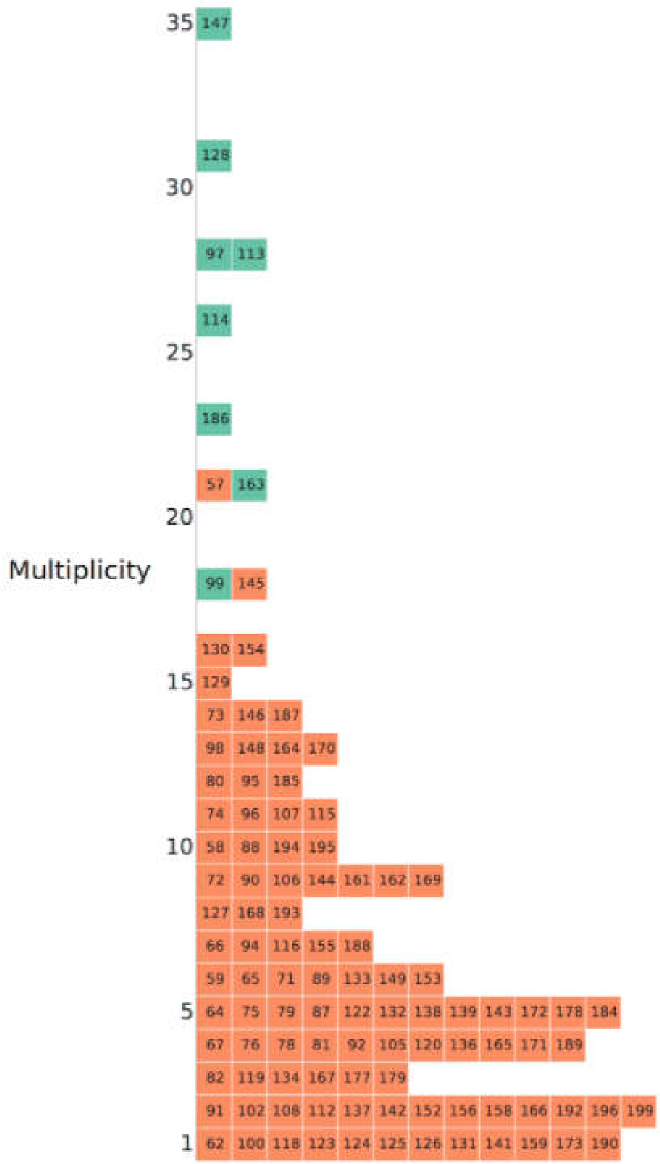



Рис. 5.5. Спектральная свертка экспериментального спектра $Spectrum_{10}$ для *Tyrocidine B1*. Для каждого элемента спектральной свертки (показанного как число внутри клетки) ее y-координата представляет количество раз, когда элемент появляется в спектральной свертке. Зеленые клетки представляют массу аминокислот *Tyrocidine B1*.

Теперь есть схема нового алгоритма секвенирования циклопептидов. Для данного экспериментального спектра сначала вычисляется свертка экспериментального спектра. Затем выбирается *M* наиболее частых элементов между 57 и 200 в свертке для формирования расширенного алфавита кандидатов аминокислотных масс («с хвостами»). Затем запустить алгоритм *LeaderboardCyclopeptideSequencing*, где массы аминокислот ограничены этим алфавитом. Этот алгоритм называется *ConvolutionCyclopeptideSequencing*.

**Задача 5.5.** Реализовать *ConvolutionCyclopeptideSequencing*.

Вход: целое число *M*, целое число *N* и набор (возможно повторяющихся) целых чисел *Spectrum*.

Выход: циклический пептид *LeaderPeptide* с аминокислотами, взятыми только из верхних *M* элементов (и хвостов) свертки *Spectrum*, которые находятся между 57 и 200, и где размер *Leaderboard* ограничен величиной *N* (и хвостами).

Пример входа:

20

60

57 57 71 99 129 137 170 186 194 208 228 265 285 299 307 323 356 364 394 422 493

Пример выхода:

99-71-137-57-72-57

*ConvolutionCyclopeptideSequencing* (с $N = 1000$ и $M = 20$) теперь корректно восстанавливает *Tyrocidine B1* из $Spectrum_{10}$. Истинный тест этого алгоритма заключается в том, будет ли он работать в более шумном спектре. Напомним, что предыдущий алгоритм не смог идентифицировать правильный пептид для $Spectrum_{25}$. Однако, *ConvolutionCyclopeptideSequencing* (с $N = 1000$ и $M = 20$) теперь правильно идентифицирует *Tyrocidine B1* из этого спектра.

## Лекция 6. Открытые проблемы секвенирования циклических пептидов.

Ранее рассматривались моделированные спектры, которые относительно легко упорядочиваются (даже те, у которых есть ложные и отсутствующие массы). Масс-спектрометр не просто взвешивает крошечные пептидные фрагменты по одному за раз, а сначала превращает субпептиды в ионы (то есть заряженные частицы). Ионизация частиц помогает масс-спектрометру сортировать ионы с помощью электромагнитного поля; ионы разделены не массой, а соотношением массы / заряда. Если фрагмент иона NQY (целая масса: 114 + 128 + 163 = 405) имеет заряд +1, то он содержит один дополнительный протон, в результате чего получается полная целая масса 406 и отношение массы / заряда 406/1 = 406. Точнее, моноизотопная масса NQY составляет приблизительно 114,043 + 128,058 + 163,063 = 405,164, а масса протона равна 1,007 Да, что делает массовый заряд / отношение более близким к (405,164 + 1,007) / 1 = 406,171.

Масс-спектрометр выводит коллекцию пиков, которые показаны на рисунке 6.1 для реального спектра *Tyrocidine B1*. X-координата каждого пика представляет собой отношение массы / заряда иона, а его высота представляет собой интенсивность (то есть относительное количество) ионов, имеющих данное отношение массы / заряда. Например, в экспериментальном спектре *Tyrocidine B1*, показанного на рисунке 6.1, можно найти небольшой пик с отношением массы / заряда 406,30, что соответствует фрагментному иону NQY, имеющему отношение массы / заряда 406,171, с погрешностью приблизительно 0,13 Да.

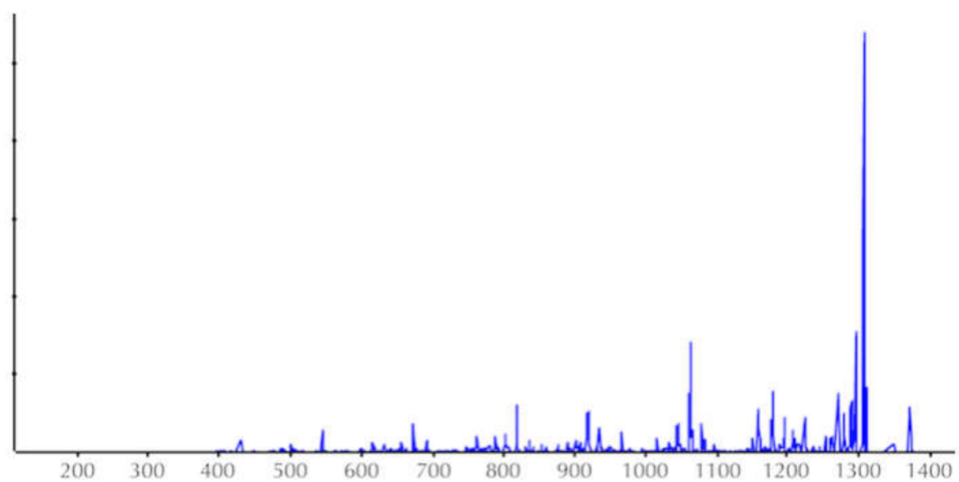



Рис. 6.1. Результаты масс-спектрометрии для *Tyrocidine B1*.

Для анализа реальных спектров нужно преодолеть несколько практических барьеров. Во-первых, заряд каждого пика неизвестен, что заставляет исследователей пробовать все возможные заряды от 1 до некоторого параметра *maxCharge*, где конкретный выбор *maxCharge* зависит от используемой технологии фрагментации. Эта процедура генерирует *maxCharge* массы для каждого пика, так что чем больше значение *maxCharge*, тем больше ложных масс в спектре.

Во-вторых, реальный спектр на рисунке 6.1 имеет почти 1000 пиков, большинство из которых являются ложными пиками, что означает, что их отношение массы / заряда не соответствует отношению массы / заряда субпептида (для любой величины заряда). Ложные пики обычно имеют низкую интенсивность, что требует предварительного этапа обработки, который удаляет пики низкой интенсивности перед применением алгоритма. Ниже приведен список из 95 отношений масса / заряд для пиков, которые «выжили» на этапе предварительной обработки спектра *Tyrocidine B1*. Их интенсивности могут, тем не менее, меняться на 2-3 порядка; например, интенсивность пика с отношением массы / заряда 372,2 в 300 раз меньше, чем интенсивность пика с отношением массы / заряда 1306,5.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 372.2 | 397.2 | 402.0 | **406.3** | 415.1 | **431.2** | **448.3** | 449.3 | 452.2 | 471.3 |
| **486.3** | **488.2** | 500.5 | **505.3** | 516.1 | 536.1 | **544.2** | **545.3** | 562.5 | 571.3 |
| 599.2 | 614.4 | 615.4 | 616.4 | 618.2 | **632.0** | 655.5 | 656.3 | **672.5** | **673.3** |
| 677.3 | **691.4** | **692.4** | 712.1 | 722.3 | **746.5** | 760.4 | 761.6 | 762.5 | **771.6** |
| 788.4 | 802.3 | 803.3 | 818.5 | **819.4** | 831.4 | **836.3** | 853.3 | 875.5 | **876.5** |
| 901.5 | 915.9 | 916.5 | 917.8 | **918.4** | **933.4** | **934.7** | **935.5** | 949.4 | **966.2** |
| 995.4 | 1015.6 | 1027.5 | 1029.5 | 1031.5 | 1044.5 | 1046.5 | **1061.5** | **1063.4** | **1079.2** |
| 1083.7 | 1088.4 | 1093.5 | **1096.5** | 1098.4 | 1158.5 | 1159.5 | **1176.6** | 1177.7 | 1178.6 |
| 1192.7 | **1195.4** | 1207.5 | **1210.4** | **1224.6** | 1252.5 | 1270.5 | 1271.5 | 1278.6 | 1279.6 |
| 1295.6 | 1305.6 | 1306.5 | 1307.5 | 1309.6 | | | | | |

Только 31 из этих 95 отношений масса / заряд (выделенных полужирным шрифтом) могут быть сопоставлены с субпептидами *Tyrocidine B1* (с *maxCharge* = 1 и максимально допустимым массовым расхождением 0,3 Да):

| Mass | Subpeptide | Mass | Subpeptide | Mass | Subpeptide |
|---|---|---|---|---|---|
| 406.2 | NQY | 431.2 | FPW | 448.2 | WFN |
| 486.2 | KLFP | 488.2 | VKLF | 505.2 | NQYV |
| 544.2 | LFPW | 545.2 | PWFN | 632.3 | QYVKL |
| 672.3 | KLFPW | 673.3 | PWFNQ | 691.3 | LFPWF |
| 692.3 | FPWFN | 746.3 | NQYVKL | 771.3 | VKLFPW |
| 819.4 | KLFPWF | 836.4 | PWFNQY | 876.4 | QYVKLFP |
| 918.4 | VKLFPWF | 933.4 | LFPWFNQ | 934.4 | YVKLFPW |
| 935.4 | PWFNQYV | 966.4 | WFNQYVK | 1061.5 | KLFPWFNQ |

| Mass | Subpeptide | Mass | Subpeptide | Mass | Subpeptide |
|---|---|---|---|---|---|
| 1063.5 | PWFNQYVK | 1079.5 | WFNQYVKL | 1096.5 | LFPWFNQY |
| 1176.5 | NQYVKLFPW | 1195.6 | LFPWFNQYV | 1210.6 | FPWFNQYVK |
| 1224.6 | KLFPWFNQY | | | | |

Теперь можно видеть, что секвенирование *Tyrocidine B1* из реального спектра, для которого две трети всех масс ложны, представляет гораздо более сложную проблему, чем секвенирование этого пептида из имитируемого $Spectrum_{25}$.

В случае алфавита из произвольных целых чисел, задача секвенирования циклопептидов соответствует проблеме в компьютерной науке, известной как Beltway Problem. Beltway Problem требует найти набор точек на окружности таким образом, чтобы расстояния между всеми парами точек (где расстояние измеряется по окружности) соответствуют заданному набору целых чисел.

Аналог проблемы Beltway в случае, когда точки лежат вдоль отрезка, а не на круге, называется Turnpike Problem. Термины «Beltway» (кольцевая дорога) и «Turnpike» (магистраль) возникают по аналогии с выходами на круговые и линейные дороги, соответственно. В случае *n* точек на окружности и линии, входы для Beltway Problem и Turnpike Problem состоят из $n\ (n - 1) + 2$ и $n\ (n - 1) / 2 + 2$ расстояний соответственно (эти формулы включают в себя расстояние 0, а также длину всего отрезка).

Попытки разработать полиномиальные алгоритмы для Beltway Problem и Turnpike Problem (или доказать их неразрешимость) потерпели неудачу. Однако существует псевдополиномиальный алгоритм для Turnpike Problem. В отличие от действительно полиномиального алгоритма, который может быть ограничен многочленом от длины входа, псевдополиномиальный алгоритм для задачи магистрали является полиномиальным по общей длине отрезка. Например, если *n* точек разделены огромными расстояниями, скажем, порядка $2^{100}$, то полиномиальный алгоритм будет по-прежнему быстрым, тогда как псевдополиномиальный алгоритм будет чрезмерно медленным. Обратите внимание: хоть сами расстояния будут огромными, каждое расстояние может быть сохранено с использованием только 100 бит.

Если $A = (a_1 = 0, a_2, \dots, a_n)$ – набор *n* точек на отрезке, расположенных в порядке возрастания ($a_1 < a_2 < \cdots < a_n$), то $\Delta A$ обозначает совокупность всех попарных разностей между точками в $A$. Например, если $A = (0, 2, 4, 7)$, то

$\Delta A$=(−7,−5,−4,−3,−2,−2,0,0,0,0,2,2,3,4,5,7).

Проблема Turnpike требует восстановления $A$ из $\Delta A$.

**Задача 6.1.** Для заданных попарных расстояний между точками на отрезке, восстановить позиции этих точек.

Вход: набор целых чисел $L$.

Выход: набор целых чисел $A$ такой, что $\Delta A = L$.

Пример входа:

-10 -8 -7 -6 -5 -4 -3 -3 -2 -2 0 0 0 0 0 2 2 3 3 4 5 6 7 8 10

Пример выхода:

0 2 4 7 10

Рассмотрим подход к решению проблемы Turnpike, который является полиномом по длине отрезка. Для заданного набора целых чисел $A = (a_1 < a_2 < \cdots < a_n)$, производящей функцией $A$ является полином

$A(x) = \sum_{i=1}^{n} x^{a_i}$.

Например, если $A$ = (0, 2, 4, 7, 10), то

$$A(x) = x^0 + x^2 + x^4 + x^7$$

$$\Delta A(x) = x^{-7} + x^{-5} + x^{-4} + x^{-3} + 2x^{-2} + 4x^0 + 2x^2 + x^3 + x^4 + x^5 + x^7$$

Можно проверить, что производящая функция для $\Delta A(x)$ равна $A(x) \cdot A(x\text{-}1)$. Таким образом, проблема Turnpike сводится к проблеме полиномиальной факторизации. Подобно тому, как целое число можно разбить на его простые множители, полином с целыми коэффициентами можно разложить на «простые» многочлены с целыми коэффициентами. Если можно определить $\Delta A(x)$ и определить, какие основные факторы вносят вклад в $A(x)$ и $A(x\text{-}1)$, то будет известно $A(x)$ и, следовательно, $A$. В 1982 году Розенблатт и Сеймур [12] описали метод представления $A(x)$ как $A(x) \cdot A(x\text{-}1)$. Так как многочлен может быть факторизован во временном многочлене по его максимальному показателю, $\Delta A(x)$ может быть факторизован во временном многочлене от общей длины отрезка, что дает желаемый псевдополиномиальный алгоритм для проблемы Turnpike.

Псевдополиномиальные алгоритмы полезны на практике, потому что практические примеры обычно не включают огромные расстояния. Несмотря на то, что для проблемы Turnpike существует псевдополиномиальный алгоритм, такой алгоритм для кажущейся аналогичной проблемы Beltway остается неоткрытым.

Бактерии и грибы не имеют монополии на продуцирование циклических пептидов; животные и растения тоже производят их (хотя и через совершенно другой механизм). Первый циклический пептид, обнаруженный у животных (называемый θ-defensin), был обнаружен в 1999 году у макак. θ-defensin предотвращает проникновение вирусов в клетки и проявляет сильную антивирусную активность. Однако, вопрос о том, как приматы делают θ-defensin, остается загадкой.

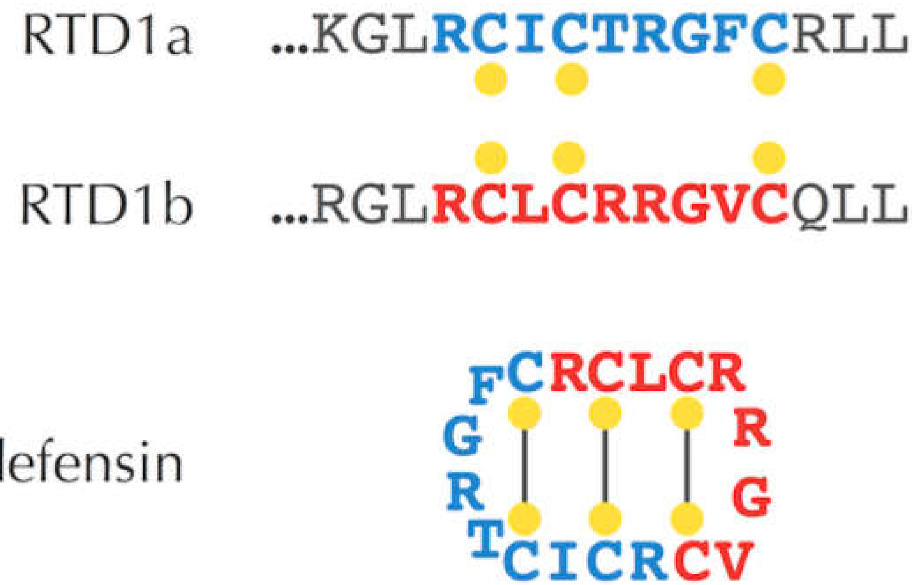



Рис. 6.2. 18-аминокислотный пептид θ-defensin образуется путем разрезания двух 9-аминокислотных длинных пептидов RCICTRGFC и RCLCRRGVC из белков *RTD1a* и *RTD1b*, их конкатенирования и затем зацикливания полученного пептида (наряду с введением трех дисульфидных мостиков, которые образуют связи через пептид).

Интересно, что макаки и бабуины производят θ-defensin, тогда как люди и шимпанзе этого не делают. Это расхождение заставляет задаться вопросом, была ли мутация у предка человека-шимпанзе, что привело к потере этого очень полезного пептида. Интересно, что гены, очень похожие на *RTD1a* и *RTD1b*, существуют у людей, но кодон в одном из этих генов, мутировал в стоп-кодон, таким образом сокращая кодированный белок. Так как этот стоп-кодон расположен перед 9-аминокислотным пептидом, ответственным за θ-defensin, люди не продуцируют этот пептид и, следовательно, не могут образовывать θ-defensin.

В эксперименте Venkataraman et al., 2009 [13] продемонстрировал, что люди могут получить θ-defensin. Некоторые лекарства могут заставить рибосому игнорировать стоп-кодоны и продолжать транслировать РНК даже после столкновения со стоп-кодоном. Исследователи продемонстрировали, что после лечения таким препаратом клетки человека начали продуцировать человеческий вариант θ-defensin. Неожиданным завершением этого эксперимента является то, что хоть люди и шимпанзе потеряли θ-defensin миллионы лет назад, все еще существуют таинственные ферменты, необходимыми для удаления и вставки его составных пептидов.

Некоторые биологи считают, что, поскольку ферменты, образующие θ-defensin, все еще работают у людей, они должны быть необходимы для чего-то другого. Если бы эти ферменты не обеспечивали некоторого избирательного преимущества, то со временем мутации бы приводили к тому, что их гены становились псевдогенами или нефункциональными остатками ранее работавших генов. Наиболее естественным объяснением того, почему эти ферменты все еще функционируют, является то, что люди производят еще неоткрытые циклические пептиды и что ферменты, необходимые для θ-defensin, также используются для «удаления и вставки» других (еще неоткрытых) циклических пептидов. Гипотеза о том, что человек может иметь неоткрытые циклические пептиды, не так невероятна, как можно было бы подумать, потому что у биологов еще нет надежных алгоритмов для обнаружения циклопептидов из миллиардов спектров, созданных в сотнях лабораторий, анализирующих протеом человека.

По умолчанию исследователи предполагают, что все спектры, когда-либо приобретенные в исследованиях протеомов человека, исходили из линейных пептидов.

## Лекция 7. Поиск мотивов.

Ежедневные распорядки животных, растений и даже бактерий контролируются внутренними часами, называемыми циркадными часами. Любой, кто испытал страдания смены часовых поясов, знает, что эти часы никогда не прекращают идти. Известно, что крысы и исследователи-добровольцы, находясь в бункере, поддерживают примерно 24-часовой цикл активности и отдыха в полной темноте. И, как и любые часы, циркадные часы могут нарушать свою работу, результатом чего является генетическое заболевание, известное как синдром отсроченного наступления фазы сна или дисания (DSPS).

Циркадные часы должны иметь некоторый базис на молекулярном уровне, который порождает много вопросов. Как отдельные клетки животных и растений, не говоря уже о бактериях, знают, когда они должны уменьшать или увеличивать производство определенных белков. Есть ли «ген синхронизации». Можно ли объяснить, почему сердечные приступы чаще возникают утром, в то время как приступы астмы чаще встречаются ночью. И можно ли идентифицировать гены, которые отвечают за «нарушение» циркадных часов, вызывающее DSPS.

В начале 1970-х годов Рон Конопка и Сеймур Бензер идентифицировали мутантных мух с аномальными циркадными паттернами и проследили мутации мух до одного гена. Биологам понадобилось еще два десятилетия, чтобы обнаружить аналогичный тактовый ген у млекопитающих, который был всего лишь первой частью головоломки. На данный момент обнаружено еще много циркадных генов; эти гены, называющиеся вневременной, часовой и циклический, организуют поведение сотен других генов и демонстрируют высокую степень эволюционного сохранения среди видов.

Сначала сосредоточимся на растениях, поскольку сохранение циркадных часов в растениях является вопросом жизни и смерти. Биологи считают, что более тысячи генов растений являются циркадными, включая гены, связанные с фотосинтезом, фоторецептором и цветением. Эти гены должны каким-то образом узнавать, в какое время эти процессы происходят, чтобы изменить транскрипцию или экспрессию генов в течение дня.

Оказывается, каждая клетка растения отслеживает наступление дня и ночи независимо от других клеток, и только три гена растений, называемые LCY, CCA1 и TOC1, являются главными хронометристами. Те регуляторные гены и регуляторные белки, которые они кодируют, часто контролируются внешними факторами (например, доступностью питательных веществ или солнечным светом), чтобы организмы могли корректировать свою экспрессию генов.

Например, регуляторные белки, контролирующие циркадные часы в растениях, координируют циркадную активность следующим образом. TOC1 способствует экспрессии LCY и CCA1, тогда как LCY и CCA1 подавляют экспрессию TOC1, приводя к контуру с отрицательной обратной связью. Утром солнечный свет активирует транскрипцию LCY и CCA1, вызывая подавление транскрипции TOC1. С уменьшением света также происходит производство LCY и CCA1, которые, в свою очередь, больше не подавляют TOC1. Транскрипция TOC1 достигает пика ночью и начинает способствовать транскрипции LCY и CCA1, которые, в свою очередь, подавляют транскрипцию TOC1, и цикл начинается снова.

LCY, CCA1 и TOC1 способны контролировать транскрипцию других генов, потому что регуляторные белки, которые они кодируют, являются транскрипционными факторами или основными регуляторными белками, которые включают и выключают другие гены. Транскрипционный фактор регулирует ген путем связывания с определенным коротким интервалом ДНК, который называется регуляторным мотивом или сайтом связывания транскрипционного фактора в области гена длиной 600-1000 нуклеотидов, предшествующей началу гена. Например, CCA1 связывается с AAAAAATCT в восходящей области многих генов, регулируемых CCA1.

Было бы легко найти регуляторные мотивы, если бы они были полностью консервативны, но реальность более сложна, поскольку регуляторные мотивы могут изменяться в некоторых позициях, например, CCA1 может вместо

AAAAAATCT связываться с AAGAACTCT. Нужно разработать алгоритмы поиска мотивов, решающие проблему обнаружения «скрытого сообщения», разделяемого набором строк.

В 2000 году Стив Кей использовал массивы ДНК для определения того, какие гены в растении *Arabidopsis thaliana* активируются в разное время суток. Затем он выделил восходящие области почти 500 генов, которые проявляли циркадное поведение и искал в них наиболее часто встречающиеся паттерны. Если объединить эти восходящие регионы в одну строку, можно обнаружить, что AAAATATCT является неожиданно частым словом, появляющимся 46 раз.

Кей назвал AAAATATCT вечерним элементом (*evening element*) и выполнил простой эксперимент, чтобы доказать, что это действительно регуляторный мотив, ответственный за экспрессию циркадных генов в *Arabidopsis thaliana*. После того, как он мутировал вечерний элемент в восходящей области одного гена, ген потерял свое циркадное поведение.

Вечерний элемент в растениях очень консервативен, поэтому его легко найти, однако, мотивы, имеющие много мутаций, более неуловимы. Например, если заразить муху бактерией, муха включит иммунные гены для борьбы с инфекцией. Таким образом, некоторые гены с повышенными уровнями экспрессии после заражения, вероятно, становятся генами иммунитета. Действительно, некоторые из этих генов имеют 12-мер, аналогичный TCGGGGATTTCC в восходящих областях, сайт связывания транскрипционного фактора NF-κB, который активирует различные гены иммунитета у мух. Однако, сайты связывания NF-κB не являются такими же консервативными, как вечерний элемент. Ниже приведен образец из десяти сайтов связывания NF-κB из генома *Drosophila melanogaster*; наиболее частые нуклеотиды в каждом столбце показаны цветными буквами верхнего регистра.

```
TCGGGGgTTTtt
cCGGtGAcTTaC
aCGGGGATTTtC
TtGGGGAcTTtt
TCGGGGATTTCC
TtGGGGAcTTCC
TCGGGGATTcat
TCGGGGATTcCt
TaGGGGAacTaC
TCGGGtATaaCC
```

Цель – превратить биологическую задачу поиска регуляторных мотивов в вычислительную проблему. Ниже 15-мерное скрытое сообщение введено в случайном порядке в каждую из десяти случайно сформированных цепочек ДНК. Этот пример имитирует сайт связывания транскрипционного фактора, скрывающийся в восходящих областях десяти генов.

```
atgaccgggatactgataaaaaaaagggggggggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
acccctattttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaataaaaaaaaaggggggga
tgagtatccctgggatgacttaaaaaaaaggggggggtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
gctgagaattggatgaaaaaaaagggggggtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
tcccttttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaataaaaaaaagggggggcttatag
gtcaatcatgttcttgtgaatggatttaaaaaaaaggggggggaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
cggttttggcccttgttagaggcccccgtaaaaaaaagggggggcaattatgagagagctaatctatcgcgtgcgtgttcat
aacttgagttaaaaaaaagggggggctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcataaaaaaaaagggggggaccgaaagggaag
ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttaaaaaaaaggggggga
```

Это простая задача – применение алгоритма решения задачи *FrequentWords* для конкатенации этих строк немедленно выявит наиболее частый 15-мер, показанный ниже, как вставленный паттерн. Поскольку эти короткие строки были сгенерированы случайным образом, маловероятно, что они содержат другие частые 15-меры.

```
atgaccgggatactgatAAAAAAAAGGGGGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
acccctattttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaataAAAAAAAAGGGGGGGa
tgagtatccctgggatgacttAAAAAAAAGGGGGGGtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
gctgagaattggatgAAAAAAAAGGGGGGGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
tcccttttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAAAAAAAAGGGGGGGcttatag
gtcaatcatgttcttgtgaatggatttAAAAAAAAGGGGGGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
cggttttggcccttgttagaggcccccgtAAAAAAAAGGGGGGGcaattatgagagagctaatctatcgcgtgcgtgttcat
aacttgagttAAAAAAAAGGGGGGGctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatAAAAAAAAGGGGGGGaccgaaagggaag
ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttAAAAAAAAGGGGGGGa
```

Теперь представьте себе, что вместо вставки точно такой же структуры во все последовательности, мутируем шаблон перед тем, как вставлять его в каждую последовательность, случайно изменяя нуклеотиды в четырех случайно выбранных положениях внутри каждого имплантированного 15-мера, как показано ниже.

```
atgaccgggatactgatAgAAgAAAGGttGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
acccctattttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaatacAAtAAAAcGGcGGGa
tgagtatccctgggatgacttAAAAtAAtGGaGtGGtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
gctgagaattggatgcAAAAAAAGGGattGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
tcccttttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAtAAtAAAGGaaGGGcttatag
gtcaatcatgttcttgtgaatggatttAAcAAtAAGGGctGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
cggttttggcccttgttagaggcccccgtAtAAAcAAGGaGGGCcaattatgagagagctaatctatcgcgtgcgtgttcat
aacttgagttAAAAAAtAGGGaGccctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatAcTAAAAAGGaGcGGaccgaaagggaag
ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttActAAAAAGGaGcGGa
```

Алгоритм *FrequentWords* здесь не поможет, так как AAAAAAAAGGGGGGG даже не появляется в приведенных выше последовательностях. Возможно, можно было бы применить решение задачи *FrequentWords* с промахами.

Однако рассмотренный алгоритм для задачи *FrequentWords* с промахами, направлен на поиск скрытых сообщений с небольшим количеством промахов и небольшим размером *k*-меров (например, один или два промаха для *DnaA*-боксов длиной 9). Этот алгоритм, вероятно, будет слишком медленным при поиске вставленного выше мотива, который длиннее и имеет больше мутаций.

Кроме того, задачи *FrequentWords* недостаточно, поскольку она неправильно моделирует биологическую проблему обнаружения мотивов. *DnaA*-бокс – это шаблон, который появляется сравнительно часто в течение относительно короткого промежутка генома. Регуляторный мотив представляет собой образец, который появляется хотя бы один раз (возможно, с мутацией) в каждой из многих разных областей, разбросанных по всему геному.

Имея набор строк *Dna* и целое число *d*, *k*-мер является (*k*, *d*)-мотивом, если он появляется в каждой строке *Dna* с не более чем *d* промахами. Например, вставленный 15-мер в приведенных выше строках представляет собой (15, 4)-мотив.

**Задача.** Найти все (*k*, *d*)-мотивы в наборе строк.

Вход: набор строк *Dna* и целые числа *k* и *d*.

Выход: все (*k*, *d*)-мотивы в *Dna*.

Полный перебор – это общая методика решения задач, которая исследует всех возможных кандидатов на решение и проверяет, решает ли каждый кандидат проблему. Такие алгоритмы требуют мало усилий для разработки и гарантируют получение правильного решения, но они могут занимать огромное количество времени, и количество кандидатов может быть слишком большим для проверки.

Метод полного перебора для решения проблемы вставленных мотивов основан на наблюдении, что любой (*k*, *d*)-мотив должен иметь не более *d* несоответствий, кроме некоторого *k*-мера, появляющегося в одной из строк *Dna*. Поэтому можно сгенерировать все такие *k*-меры, а затем проверить, какие из них являются (*k*, *d*)-мотивами.

```
MotifEnumeration(Dna, k, d)
    Patterns ← an empty set
    for each k-mer Pattern in Dna
        for each k-mer Pattern' differing from Pattern by at most d
         mismatches
            if Pattern' appears in each string from Dna with at most d
         mismatches
                add Pattern' to Patterns
    remove duplicates from Patterns
    return Patterns
```

**Задача 7.1.** Реализовать *MotifEnumeration*.

Вход: набор строк *Dna* и целые числа *k* и *d*.

Выход: все (*k*, *d*)-мотивы в *Dna*.

Пример входа:

3 1

ATTTGGC

TGCCTTA

CGGTATC

GAAAATT

Пример выхода:

ATA ATT GTT TTT

К сожалению, для больших значений *k* и *d* *MotifEnumeration* работает довольно медленно, и поэтому попытаемся использовать другой подход. Может быть, можно обнаружить вставленный образец, идентифицируя два наиболее схожих *k*-мера между каждой парой строк в *Dna*. Однако, рассмотрим имплантированные 15-меры AgAAgAAAGGttGGG и cAAtAAAAcGGGGcG, каждый из которых отличается от AAAAAAAAGGGGGGG четырьмя промахами. Несмотря на то, что данные 15-меры похожи на правильный мотив AAAAAAAAGGGGGGG, они не так похожи в сравнении друг с другом, имея восемь несоответствий:

```
AgAAgAAAGGttGGG
|| ||   | || |
cAAtAAAAcGGGGcG
```

Поскольку эти два вставленных шаблона настолько различны, нужно заботиться о том, можно ли найти их, выполнив поиск двух похожих *k*-меров среди пар строк в *Dna*.

Хотя проблема вставленного мотива представляет собой полезную абстракцию биологической проблемы обнаружения мотивов, она имеет некоторые ограничения. Например, когда Стив Кей использовал массив ДНК, чтобы сделать вывод о циркадных генах в растениях, он не ожидал, что все гены в результирующем наборе будут иметь вечерний элемент (или его варианты) в своих регионах. Аналогичным образом, биологи не ожидают, что все гены с повышенным уровнем экспрессии инфицированных мух должны регулироваться NF-κB.

Массив ДНК представляет собой набор молекул ДНК, прикрепленных к твердой поверхности. Каждой области на массиве назначается уникальная последовательность ДНК, называемая пробой, которая измеряет уровень экспрессии определенного гена, известного как цель. В большинстве массивов пробы синтезируются, а затем прикрепляются к стеклянному или кремниевому чипу (см. Рисунок 7.1). Флуоресцентно помеченные мишени затем связываются с соответствующей пробой (например, когда их последовательности являются комплементарными), генерируя флуоресцентный сигнал. Сила этого сигнала зависит от количества целевого образца, который связывается с пробой в данном месте. Таким образом, чем выше уровень экспрессии гена, тем выше интенсивность его флуоресцентного сигнала на матрице. Поскольку массив может содержать миллионы проб, биологи могут измерять экспрессию многих генов в эксперименте с одним массивом. Эксперимент с ДНК-матрицей, который идентифицировал вечерний элемент в *Arabidopsis thaliana*, измерял экспрессию 8000 генов.

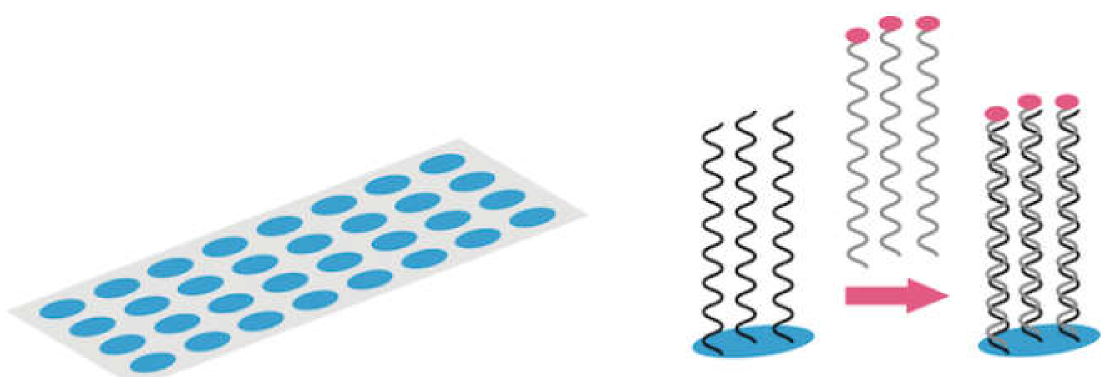


Рис. 7.1. Флуоресцентно помеченная ДНК связывается с комплементарной пробой на ДНК-матрице.

Эксперименты с ДНК-матрицами по своей природе являются зашумленными, и некоторые гены, найденные в этих экспериментах, не имеют ничего общего с циркадными часами в растениях или генами иммунитета у мух. Для таких шумных наборов данных любой алгоритм для проблемы поиска вставленного мотива потерпит неудачу, поскольку, когда ни одна последовательность не содержит сайт связывания транскрипционного фактора, ($k$, $d$)-мотив не существует.

Более подходящая формулировка задачи поиска мотивов будет оценивать отдельные примеры мотивов в зависимости от того, насколько они похожи на «идеальный» мотив (т.е. сайт связывания транскрипционного фактора, который связывается лучшим образом с транскрипционным фактором). Однако, поскольку идеальный мотив неизвестен, будем выбирать $k$-меры из каждой строки и оценивать эти $k$-меры в зависимости от того, насколько они похожи друг на друга.

Чтобы определить оценку, рассмотрим $t$ последовательностей ДНК длины $n$, и выберем $k$-меры из каждой последовательности, чтобы сформировать коллекцию *Motifs*, которую представим в виде матрицы размером $t \times k$. На рисунке 7.2 показана матрица мотивов для сайтов связывания NF-κB, самый популярный нуклеотид в каждом столбце матрицы мотивов указан заглавными буквами.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Motifs** | T | C | G | G | G | G | g | T | T | T | t | t |
| | c | C | G | G | t | G | A | c | T | T | a | C |
| | a | C | G | G | G | G | A | T | T | T | t | C |
| | T | t | G | G | G | G | A | c | T | T | t | t |
| | a | a | G | G | G | G | A | c | T | T | C | C |
| | T | t | G | G | G | G | A | c | T | T | C | C |
| | T | C | G | G | G | G | A | T | T | c | a | t |
| | T | C | G | G | G | G | A | T | T | c | C | t |
| | T | a | G | G | G | G | A | a | c | T | a | C |
| | T | C | G | G | G | t | A | T | a | a | C | C |

Рис. 7.2. Матрица мотивов для сайтов связывания NF-κB.

Если в столбце имеется несколько самых популярных нуклеотидов, то произвольно выбирается один из них. Обратите внимание, что положения 2 и 3 являются наиболее консервативными (нуклеотид G полностью сохраняется в этих положениях), тогда как положение 10 является наименее консервативным.

Изменяя выбор $k$-меров в каждой последовательности, можно построить большое количество различных матриц мотивов из данного образца последовательностей ДНК. Цель – таким образом выбрать $k$-меры, чтобы в результате получилась самая «консервативная» матрица мотивов, что означает матрицу с преобладанием заглавных букв (и, следовательно, наименьшим количеством строчных букв). Оставив в стороне вопрос о том, как выбирать такие $k$-меры, сосредоточимся сначала на определении для полученных матриц мотивов *Score*(*Motifs*) как количества непопулярных (строчных) букв в матрице *Motifs*. Цель – найти коллекцию *k*-меров, которая минимизирует эту оценку.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Motifs | T | C | G | G | G | G | g | T | T | T | t | t |
| | c | C | G | G | t | G | A | c | T | T | a | C |
| | a | C | G | G | G | G | A | T | T | T | t | C |
| | T | t | G | G | G | G | A | c | T | T | t | t |
| | a | a | G | G | G | G | A | c | T | T | C | C |
| | T | t | G | G | G | G | A | c | T | T | C | C |
| | T | C | G | G | G | G | A | T | T | c | a | t |
| | T | C | G | G | G | G | A | T | T | c | C | t |
| | T | a | G | G | G | G | A | a | c | T | a | C |
| | T | C | G | G | G | t | A | T | a | a | C | C |

Score 3 + 4 + 0 + 0 + 1 + 1 + 1 + 5 + 2 + 3 + 6 + 4 = 30

Рис. 7.3. Матрица мотивов для сайтов связывания NF-κB) и оценка *Score*.

Можно построить $4 \times k$ матрицу *Count*(*Motifs*), подсчитывая количество вхождений каждого нуклеотида в каждом столбце матрицы *Motifs*; ($i, j$)-й элемент *Count*(*Motifs*) хранит количество раз, которое нуклеотид $i$ появляется в столбце $j$. Далее разделим все элементы матрицы *Count* на $t$, количество строк в *Motifs*. Это приводит к матрице профиля $P$ = *Profile*(*Motifs*), для которой $P_{i,j}$ - частота $i$-го нуклеотида в $j$-м столбце матрицы *Motifs*. Обратите внимание, что элементы любого столбца матрицы профиля в сумме дают 1. На рисунке 7.4 показаны матрицы *Motifs*, *Count* и *Profile* для сайтов связывания NF-κB.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Motifs** | | T | C | G | G | G | G | g | T | T | T | t | t |
| | | c | C | G | G | t | G | A | c | T | T | a | C |
| | | a | C | G | G | G | G | A | T | T | T | t | C |
| | | T | t | G | G | G | G | A | c | T | T | t | t |
| | | a | a | G | G | G | G | A | c | T | T | C | C |
| | | T | t | G | G | G | G | A | c | T | T | C | C |
| | | T | C | G | G | G | G | A | T | T | c | a | t |
| | | T | C | G | G | G | G | A | T | T | c | C | t |
| | | T | a | G | G | G | G | A | a | c | T | a | C |
| | | T | C | G | G | G | t | A | T | a | a | C | C |

**Score** 3 + 4 + 0 + 0 + 1 + 1 + 1 + 5 + 2 + 3 + 6 + 4 = 30

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Count** | A: | 2 | 2 | 0 | 0 | 0 | 0 | 9 | 1 | 1 | 1 | 3 | 0 |
| | C: | 1 | 6 | 0 | 0 | 0 | 0 | 0 | 4 | 1 | 2 | 4 | 6 |
| | G: | 0 | 0 | 10 | 10 | 9 | 9 | 1 | 0 | 0 | 0 | 0 | 0 |
| | T: | 7 | 2 | 0 | 0 | 1 | 1 | 0 | 5 | 8 | 7 | 3 | 4 |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Profile** | A: | .2 | .2 | 0 | 0 | 0 | 0 | .9 | .1 | .1 | .1 | .3 | 0 |
| | C: | .1 | .6 | 0 | 0 | 0 | 0 | 0 | .4 | .1 | .2 | .4 | .6 |
| | G: | 0 | 0 | 1 | 1 | .9 | .9 | .1 | 0 | 0 | 0 | 0 | 0 |
| | T: | .7 | .2 | 0 | 0 | .1 | .1 | 0 | .5 | .8 | .7 | .3 | .4 |

Рис. 7.4. Матрицы *Motifs, Count и Profile* для сайтов связывания NF-κB) и оценка *Score*.

Наконец, формируем консенсусную строку, обозначаемую *Consensus*(*Motifs*), из самых популярных букв в каждом столбце матрицы мотива. Если правильно выбирать *Motifs* из коллекции восходящих регионов, то *Consensus*(*Motifs*) обеспечивает идеальный регуляторный мотив для этих регионов. Например, консенсусной строкой для сайтов связывания NF-κB на рисунке 7.5 является TCGGGGATTTCC.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Motifs** | T | C | G | G | G | G | g | T | T | T | t | t |
| | c | C | G | G | t | G | A | c | T | T | a | C |
| | a | C | G | G | G | G | A | T | T | T | t | C |
| | T | t | G | G | G | G | A | c | T | T | t | t |
| | a | a | G | G | G | G | A | c | T | T | C | C |
| | T | t | G | G | G | G | A | c | T | T | C | C |
| | T | C | G | G | G | G | A | T | T | c | a | t |
| | T | C | G | G | G | G | A | T | T | c | C | t |
| | T | a | G | G | G | G | A | a | c | T | a | C |
| | T | C | G | G | G | t | A | T | a | a | C | C |
| **Consensus** | T | C | G | G | G | G | A | T | T | T | C | C |

Рис. 7.5. Консенсусная строка для матрицы *Motifs*.

Рассмотрим второй столбец (содержащий 6 C, 2 A и 2 T) и последний столбец (содержащий 6 C и 4 T) в матрице мотивов. Оба этих столбца вносят 4 в *Score*(*Motifs*).

Для многих биологических мотивов в некоторых положениях имеются два нуклеотида с примерно одинаковой способностью связываться с транскрипционным фактором. Например, шестнадцатинуклеотидный сайт связывания транскрипционного фактора CSRE у дрожжей *S.cerevisiae* состоит из пяти сильно консервативных позиций (1, 8, 9, 12 и 13) в дополнении к одиннадцати слабо консервативным позициям, каждая из которых имеет два нуклеотида с подобными частотами.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C | G/C | G/T | T/A | C/T | G/C | C/G | A | T | G/T | C/G | A | T | C/T | C/T | G/T |

Следуя этому примеру, более подходящее представление консенсусной строки TCGGGGATTTCC для сайтов связывания NF-κB должно включать альтернативы наиболее популярным нуклеотидам в каждом столбце. Например, нуклеотиды с частотами, равными или превышающими 0.4. В этом смысле последний столбец (6 C, 4 T) в матрице мотивов NF-κB (воспроизведенный ниже) «более консервативен», чем второй столбец (6 C, 2 A, 2 T) и должен получать более низкую оценку.

Каждый столбец *Profile*(*Motifs*) соответствует распределению вероятности или набору неотрицательных чисел, которые в сумме дают 1. Например, второй столбец в матрице профиля для сайтов связывания NF-κB соответствует вероятностям 0.2, 0.6, 0.0, и 0.2 для A, C, G и T соответственно.

Энтропия является мерой неопределенности распределения вероятностей $(p_1, \dots, p_N)$ и определяется следующим образом:

$$H(p_1, \dots, p_N) = -\sum_{i=1}^{N} p_i \cdot \log_2 p_i$$

Например, энтропия распределения вероятностей (0.2, 0.6, 0.0, 0.2), соответствующая второму столбцу матрицы профиля NF-κB, равна:

$$-(0.2\log_2 0.2 + 0.6\log_2 0.6 + 0.0\log_2 0.0 + 0.2\log_2 0.2) \approx 1.371$$

тогда как энтропия более консервативного последнего столбца (0.0, 0.6, 0.0, 0.4) равна:

$$-(0.0\log_2 0.0 + 0.6\log_2 0.6 + 0.0\log_2 0.0 + 0.4\log_2 0.4) \approx 0.971$$

и энтропия очень консервативного 5-го столбца (0.0, 0.0, 0.9, 0.1) равна:

$$-(0.0\log_2 0.0 + 0.0\log_2 0.0 + 0.9\log_2 0.9 + 0.1\log_2 0.1) \approx 0.467$$

Технически, $\log_2 0.0$ не определен, но при вычислении энтропии предполагается, что $\log_2 0.0$ равен 0.

Энтропия полностью консервативного третьего столбца равна 0, что является минимально возможной энтропией. С другой стороны, столбец с одинаково вероятными нуклеотидами (все вероятности равны 1/4) имеет максимально возможную энтропию $4 \cdot \frac{1}{4} \cdot \log_2 \frac{1}{4} = 2$. В общем, чем более консервативен столбец, тем меньше его энтропия. Таким образом, энтропия предлагает улучшенный метод оценки мотивов: энтропия матрицы мотивов определяется как сумма энтропий ее столбцов.

Другим применением энтропии является логотип мотива, диаграмма для визуализации мотивов, которая состоит из столбцов букв в каждой позиции. Логотип мотивов для матрицы мотивов NF-κB показан на рисунке 7.6. Относительные размеры букв указывают их частоту в столбце. Общая высота букв в столбце основана на информационном содержании столбца, который определяется как $2 - H(p_1, \ldots, p_N)$. Чем ниже энтропия, тем выше информационное содержание, что означает, что высокие столбцы в логотипе мотивов являются высококонсервативными.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Motifs** | T | C | G | G | G | G | g | T | T | T | t | t |
| | c | C | G | G | t | G | A | c | T | T | a | C |
| | a | C | G | G | G | G | A | T | T | T | t | C |
| | T | t | G | G | G | G | A | c | T | T | t | t |
| | a | a | G | G | G | G | A | c | T | T | C | C |
| | T | t | G | G | G | G | A | c | T | T | C | C |
| | T | C | G | G | G | G | A | T | T | c | a | t |
| | T | C | G | G | G | G | A | T | T | c | C | t |
| | T | a | G | G | G | G | A | a | c | T | a | C |
| | T | C | G | G | G | t | A | T | a | a | C | C |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Profile** | A: | .2 | .2 | 0 | 0 | 0 | 0 | .9 | .1 | .1 | .1 | .3 | 0 |
| | C: | .1 | .6 | 0 | 0 | 0 | 0 | 0 | .4 | .1 | .2 | .4 | .6 |
| | G: | 0 | 0 | 1 | 1 | .9 | .9 | .1 | 0 | 0 | 0 | 0 | 0 |
| | T: | .7 | .2 | 0 | 0 | .1 | .1 | 0 | .5 | .8 | .7 | .3 | .4 |


Рис. 7.6. Логотип мотивов для матрицы *Motifs*.

## Лекция 8. Поиск медианной строки. Жадный поиск мотивов.

Теперь, когда введена оценка коллекции $k$-меров, можно сформулировать проблему поиска мотивов.

**Задача.** Найти для заданного набора строк набор *k*-меров, по одному для каждой строки, которые минимизируют оценку полученного мотива.

Вход: набор строк *Dna* и целое число *k*.

Выход: коллекция *Motifs* *k*-меров, по одному для каждой строки в *Dna*, минимизирующая *Score*(*Motifs*) среди всех возможных вариантов *k*-меров.

Алгоритм полного перебора для проблемы поиска мотивов (называемый *BruteForceMotifSearch*) рассматривает каждый возможный выбор *k*-меров *Motifs* из *Dna* (один *k*-мер для каждой строки из *n* нуклеотидов) и возвращает коллекции *Motifs*, имеющие минимальную оценку. Поскольку в каждой из *t* последовательностей имеется $n - k + 1$ вариантов *k*-меров, существуют $(n - k + 1)^t$ различных способов формирования *Motifs*. Для каждого выбора *Motifs* алгоритм вычисляет *Score*(*Motifs*), который требует $k \cdot t$ шагов. Таким образом, при условии, что *k* меньше *n*, общее время работы алгоритма $O(n^t \cdot k \cdot t)$. Нужно разработать более быстрый алгоритм.

Поскольку *BruteForceMotifSearch* неэффективен, вместо того, чтобы исследовать все мотивы в *Dna* и потом выводить консенсусную строку из *Motifs*:

*Motifs* → *Consensus*(*Motifs*),

сначала исследуем все потенциальные *k*-мерные консенсусные строки, а затем найдем наилучшие возможные *Motifs* для каждой консенсусной строки:

*Consensus*(*Motifs*) → *Motifs*.

Чтобы переформулировать проблему поиска мотивов, нужно разработать альтернативный способ вычисления *Score*(*Motifs*). В *Score*(*Motifs*) количество строчных букв в матрице мотивов, вычислялось столбец за столбцом. Например, оценка матрицы мотивов NF-κB предварительно вычислена как 3 + 4 + 0 + 0 + 1 + 1 + 1 + 5 + 2 + 3 + 6 + 4 = 30. На рисунке 8.1 показано, что *Score*(*Motifs*) можно также вычислить по строкам (3 + 4 + 2 + 4 + 3 + 2 + 3 + 2 + 4 + 3 = 30). Стоит обратить внимание, что каждый элемент последней суммы представляет собой количество несоответствий между консенсусной строкой TCGGGGATTTCC и мотивом в соответствующей строке матрицы мотивов, т.е. расстояние Хэмминга между этими строками. Для первой строки представленной на рисунке 8.1 матрицы мотивов *d*(TCGGGGATTTCC, TCGGGGgTTTtt) = 3.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Motifs** | T | C | G | G | G | G | g | T | T | T | t | t | 3 |
| | c | C | G | G | t | G | A | c | T | T | a | C | 4 |
| | a | C | G | G | G | G | A | T | T | T | t | C | 2 |
| | T | t | G | G | G | G | A | c | T | T | t | t | 4 |
| | a | a | G | G | G | G | A | c | T | T | C | C | 3 |
| | T | t | G | G | G | G | A | c | T | T | C | C | 2 |
| | T | C | G | G | G | G | A | T | T | c | a | t | 3 |
| | T | C | G | G | G | G | A | T | T | c | C | t | 2 |
| | T | a | G | G | G | G | A | a | c | T | a | C | 4 |
| | T | C | G | G | G | t | A | T | a | a | C | C | 3 |

**Score** 3 + 4 + 0 + 0 + 1 + 1 + 1 + 5 + 2 + 3 + 6 + 4 = **30**

**Consensus** T C G G G G A T T T C C

Рис. 8.1. Мотивы и оценка *Score* в дополнение к консенсусной строке для сайтов связывания NF-кB. Вместо того, чтобы добавлять неконсенсусные элементы столбец за столбцом (т. е. нуклеотиды в нижнем регистре), можно добавить их по строкам, как показано справа от матрицы мотивов. Каждое значение в конце строки соответствует расстоянию Хэмминга между этой строкой и консенсусной строкой.

Для заданной коллекции *k*-меров *Motifs* = {$Motif_1$, ..., $Motif_t$} и *k*-мера *Pattern*, определим *d*(*Pattern*, *Motifs*) как сумму расстояний Хэмминга между *Pattern* и каждым $Motif_i$:

$$d(Pattern, Motifs) = \sum_{i=1}^{t} HammingDistance(Pattern, Motif_i).$$

Поскольку *Score*(*Motifs*) соответствует подсчету строчных букв по столбцам и *d*(*Consensus*(*Motifs*), *Motifs*) соответствует подсчету этих элементов по строкам, получим, что

$$Score(Motifs) = d(Consensus(Motifs), Motifs).$$

Это уравнение дает идею. Вместо поиска коллекции *k*-меров *Motifs*, минимизирующих *Score*(*Motifs*), можно искать потенциальную консенсусную строку *Pattern*, минимизирующую *d*(*Pattern*, *Motifs*) среди всех возможных *k*-меров *Pattern* и всех возможных вариантов *k*-меров *Motifs* в *Dna*. Эта проблема эквивалентна задаче поиска мотивов.

**Задача.** Найти для заданного набора строк набор *k*-меров, по одному для каждой строки, которые минимизируют расстояние между всеми возможными шаблонами и всеми возможными наборами *k*-меров.

Вход: набор строк *Dna* и целое число *k*.

Выход: *k*-мер *Pattern* и набор *k*-меров, по одному из каждой строки в *Dna*, минимизирующие *d*(*Pattern*, *Motifs*) среди всех возможных вариантов *Pattern* и *Motifs*.

На первый взгляд может показаться, задача усложнена. Вместо того, чтобы искать все мотивы, теперь нужно искать по всем *Motifs*, а также всем *k*-мерам *Pattern*. Ключевое наблюдение для решения эквивалентной проблемы поиска мотивов состоит в том, что, зная *Pattern*, не нужно исследовать все возможные коллекции *Motifs*, чтобы минимизировать *d*(*Pattern*, *Motifs*).

Определим *Motifs*(*Pattern*, *Dna*) как набор *k*-меров, который минимизирует *d*(*Pattern*, *Motifs*) для данного *Pattern* и всех возможных наборов *k*-меров *Motifs* в *Dna*. Например, для строк *Dna*, показанных ниже, пять цветных 3-меров представляют собой *Motifs*(AAA, *Dna*).

```
        ttaccttAAc
        gAtAtctgtc
Dna     Acggcgttcg
        ccctAAAgag
        cgtcAgAggt
```

Причина, по которой не нужно рассматривать все возможные коллекции *Motifs* в *Dna* = $Dna_1$, ..., $Dna_t$ – это то, что можно генерировать *k*-меры в *Motifs*(*Pattern*, *Dna*) по одному за раз; т.е. можно выбрать *k*-мер в $Dna_i$ независимо от выбора *k*-меров во всех остальных строках *Dna*. Для данного *k*-мера *Pattern* и длинной строки *Text*, *d*(*Pattern*, *Text*) обозначает минимальное расстояние Хэмминга между *Pattern* и любым *k*-мером в *Text*,

$$d(Pattern, Text) = \min_{all\ k-mers\ Pattern'\ in\ Text} HammingDistance(Pattern, Pattern')$$

Например, *d*(GATTCTCA, GCAAAGACGCTGACCAA) = 3.

Тот *k*-мер в *Text*, для которого расстояние Хэмминга с *Pattern* минимально, обозначается *Motif*(*Pattern*, *Text*). Для приведенного выше примера, *Motif*(GATTCTCA, GCAAAGACGCTGACCAA) = GACGCTGA.

Обозначение *Motif*(*Pattern*, *Text*) неоднозначно, поскольку в *Text* может быть несколько *k*-меров, которые достигают минимального расстояния Хэмминга с *Pattern*. Например, *Motif*(AAG, GCAATCCTCAGC) может быть либо AAT, либо CAG. Однако, эта двусмысленность не влияет на дальнейший анализ.

Для данного *k*-мера *Pattern* и набора строк *Dna* = {$Dna_1$, ..., $Dna_t$}, *d*(*Pattern*, *Dna*) определяется как сумма расстояний между *Pattern* и всеми строками в *Dna*:

$$d(Pattern, Dna) = \sum_{i=1}^{t} d(Pattern, Dna_i).$$

Например, для строк *Dna* ниже, *d*(AAA, *Dna*) = 1 + 1 + 2 + 0 + 1 = 5.

```
ttaccttAAc 1
gAtAtctgtc 1
Acggcgttcg 2
ccctAAAgag 0
cgtcAgAggt 1
```

Цель – найти *k*-мер *Pattern*, который минимизирует *d*(*Pattern*, *Dna*) по всем *k*-мерам *Pattern*, эту же задачу пытается решить эквивалентная проблема поиска мотивов. Назовем такой *k*-мер медианной строкой для *Dna*.

**Задача.** Найти медианную строку.

Вход: набор строк *Dna* и целое число *k*.

Выход: *k*-мер *Pattern*, который минимизирует *d*(*Pattern*, *Dna*) среди всех *k*-меров *Pattern*.

Поиск медианной строки требует решения проблемы двойной минимизации. Нужно найти *k*-мер *Pattern*, который минимизирует *d*(*Pattern*, *Dna*), где эта функция сама вычисляется как взятие минимума по всем выборам *k*-меров из каждой строки в *Dna*.

Далее представлен псевдокод для решения полным перебором задачи поиска медианной строки.

```
MEDIANSTRING(Dna, k)
    distance ← ∞
    for each k-mer Pattern from AA…AA to TT…TT
        if distance > d(Pattern, Dna)
            distance ← d(Pattern, Dna)
            Median ← Pattern
    return Median
```

**Задача 8.1.** Реализовать *MedianString*.

Вход: набор строк *Dna* и целое число *k*.

Выход: *k*-мер *Pattern*, который минимизирует *d*(*Pattern*, *Dna*) среди всех *k*-меров *Pattern*.

Пример входа:

3

AAATTGACGCAT

GACGACCACGTT

CGTCAGCGCCTG

GCTGAGCACCGG

AGTTCGGGACAG

Пример выхода:

GAC

Чтобы понять смысл переформулирования проблемы поиска мотивов в качестве эквивалентной задачи поиска медианной строки, рассмотрим время выполнения *MedianString* и *BruteForceMotifSearch*. Первый алгоритм вычисляет *d*(*Pattern*, *Dna*) для каждого из $4^k$ *k*-меров *Pattern*. Каждое вычисление *d*(*Pattern*, *Dna*) требует одного прохода по каждой строке в *Dna*, что требует приблизительно $k \cdot n \cdot t$ операций для *t* строк длины *n* в *Dna*. Поэтому *MedianString* имеет время работы O $(4^k \cdot n \cdot k \cdot t)$, которое на практике лучше времени работы O $(n^t \cdot k \cdot t)$ *BruteForceMotifSearch*, поскольку длина мотива (*k*) обычно не превышает 20 нуклеотидов, тогда как *t* измеряется в тысячах.

Можно видеть, что переосмысление формулирования проблемы может привести к значительному улучшению времени выполнения, необходимого для её решения. В данном случае это замечание, что *Score*(*Motifs*) можно вычислить по строкам вместо столбцов быстрым алгоритмом *MedianString*.

До сих пор предполагалось, что значение *k* известно заранее, что на практике не так. В результате это вынуждает использовать алгоритмы поиска мотивов для разных значений *k*, а затем попытаться вывести правильную длину мотива. Поскольку регуляторные мотивы бывают достаточно длинные – *MedianString* может быть слишком медленным, чтобы их найти.

Многие алгоритмы – это итеративные процедуры, которые должны выбирать среди множества альтернатив на каждой итерации. Некоторые из этих альтернатив могут привести к правильным решениям, тогда как другие не могут. Жадные алгоритмы выбирают «самую привлекательную» альтернативу на каждой итерации. Например, жадный алгоритм в шахматах может попытаться захватить самую ценную фигуру противника на каждом шагу. Тем не менее, эта стратегия, заглядывающая только на один шаг вперед, может привести к катастрофическим результатам. В целом, наиболее жадные алгоритмы обычно не могут найти точного решения проблемы; вместо этого они часто бывают быстрыми эвристиками, которые используются для скорости, чтобы найти приблизительное решение. Для многих биологических проблем жадные алгоритмы окажутся весьма полезными.

Рассмотрим жадный подход к поиску мотивов. Пусть *Motifs* – совокупность *k*-меров, взятых из *t* строк *Dna*. Профильную матрицу с *k* столбцами можно рассматривать как коллекцию *k* кубиков, которые выбрасываются, чтобы случайным образом генерировать *k*-мер. Например, если первый столбец матрицы профиля (0,2, 0,1, 0,0, 0,7), то генерируем A в качестве первого нуклеотида с вероятностью 0.2, C с вероятностью 0.1, G с вероятностью 0.0 и T с вероятностью 0.7.

На рисунке 8.2 показана матрица профилей для сайтов связывания NF-κB, где одиночная цветная запись в *i*-м столбце соответствует *i*-му нуклеотиду в ACGGGGATTACC. Вероятность *Pr*(ACGGGGATTACC | *Profile*), которую *Profile* генерирует для ACGGGGATTACC, вычисляется путем умножения выделенных записей в матрице профиля.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Profile** | A: | .2 | .2 | 0 | 0 | 0 | 0 | .9 | .1 | .1 | .1 | .3 | 0 |
| | C: | .1 | .6 | 0 | 0 | 0 | 0 | 0 | .4 | .1 | .2 | .4 | .6 |
| | G: | 0 | 0 | 1 | 1 | .9 | .9 | .1 | 0 | 0 | 0 | 0 | 0 |
| | T: | .7 | .2 | 0 | 0 | .1 | .1 | 0 | .5 | .8 | .7 | .3 | .4 |
| **String** | | A | C | G | G | G | G | A | T | T | A | C | C |

**Probability** .2 ·.6 · 1 · 1 ·.9 ·.9 ·.9 ·.5 ·.8 ·.1 ·.4 ·.6 = **0.000839808**

Рис. 8.1. Матрица *Profile* для сайтов связывания NF-κB.

*k*-мер имеет тенденцию иметь более высокую вероятность, когда он больше похож на консенсусную строку профиля. Например, для того же профиля и его консенсусной строки TCGGGGATTTCC:

Pr (**TCGGGGATTTCC** | *Profile*) = **0.7** · **0.6** · **1.0** · **1.0** · **0.9** · **0.9** · **0.9** · **0.5** · **0.8** · **0.7** · **0.4** · **0.6** =0.0205753

которая больше, чем значение *Pr*(ACGGGGATTACC | *Profile*) = 0.000839808, вычисленное ранее.

Для заданной матрицы *Profile* можно оценить вероятность каждого *k*-мера в строке *Text* и найти наиболее вероятный *k*-мер в *Text*, т.е. *k*-мер, который, скорее всего, был сгенерирован *Profile* среди все *k*-меров в *Text*. Например, ACGGGGATTACC является наиболее вероятным 12-мером в силу *Profile* в GGTACGGGGATTACCT. В самом деле, каждый другой 12-мер в этой строке имеет вероятность 0. В общем случае, если в *Text* имеется несколько наиболее вероятных *k*-меров, выбирается первый такой *k*-мер, встречающийся в *Text*.

**Задача 8.2.** Найти наиболее вероятный *k*-мер в силу *Profile* в строке.

Вход: строка *Text*, целое число *k* и матрица *Profile* размером $4\times k$.

Выход: наиболее вероятный *k*-мер в силу *Profile* в строке *Text*.

Пример входа:

ACCTGTTTATTGCCTAAGTTCCGAACAAACCCAATATAGCCCGAG GGCCT

5

0.2 0.2 0.3 0.2 0.3

0.4 0.3 0.1 0.5 0.1

0.3 0.3 0.5 0.2 0.4

0.1 0.2 0.1 0.1 0.2

Пример выхода:

CCGAG

Предлагаемый жадный алгоритм поиска мотива, *GreedyMotifSearch*, пробует каждый из *k*-меров в $Dna_1$ в качестве первого мотива. Для данного выбора *k*-мера $Motif_1$ в $Dna_1$ он создает матрицу профиля *Profile* для этого одиночного *k*-мера и устанавливает $Motif_2$ равным самому вероятному *k*-меру в $Dna_2$. Затем он выполняет итерацию, обновляя *Profile* как матрицу профиля, образованную из $Motif_1$ и $Motif_2$, и устанавливает $Motif_3$ равным самому вероятному *k*-меру в $Dna_3$. В общем случае, после обнаружения (*i-1*) *k*-меров *Motifs* в первых (*i-1*) строках *Dna*, *GreedyMotifSearch* конструирует *Profile*(*Motifs*) и выбирает наиболее вероятный *k*-мер из $Dna_i$ на основе этой профильной матрицы. После получения *k*-мера из каждой строки для получения коллекции *Motifs*, *GreedyMotifSearch* проверяет, превосходит ли *Motifs* текущую коллекцию мотивов с лучшей оценкой, а затем перемещает $Motif_1$ на один символ в $Dna_1$, начиная снова весь процесс генерации *Motifs*.

```
GREEDYMOTIFSEARCH(Dna, k, t)
    BestMotifs ← motif matrix formed by first k-mers in each string
                from Dna
    for each k-mer Motif in the first string from Dna
        Motif1 ← Motif
        for i = 2 to t
            form Profile from motifs Motif1, …, Motifi - 1
            Motifi ← Profile-most probable k-mer in the i-th string
                    in Dna
        Motifs ← (Motif1, …, Motift)
        if Score(Motifs) < Score(BestMotifs)
            BestMotifs ← Motifs
    return BestMotifs
```

**Задача 8.3.** Реализовать *GreedyMotifSearch*.

Вход: целые числа *k* и *t*, коллекция строк *Dna*.

Выход: коллекция строк *BestMotifs*, полученных в результате применения *GreedyMotifSearch*(*Dna, k, t*). Если на каком-либо шаге найдется более одного наиболее вероятного *k*-мера в данной строке, использовать тот, который встречается первым.

Пример входа:

3 5

GGCGTTCAGGCA

AAGAATCAGTCA

CAAGGAGTTCGC

CACGTCAATCAC

CAATAATATTCG

Пример выхода:

CAG

CAG

CAA

CAA

CAA

*GreedyMotifSearch* может на первый взгляд показаться достаточно хорошим алгоритмом, однако, на самом деле это не так. Можно проверить, найдет ли *GreedyMotifSearch* (4,1)-мотив ACGT, имплантированный в строки *Dna*, показанные ниже.

```
ttACCTtaac
gATGTctgtc
acgGCGTtag
ccctaACGAg
cgtcagAGGT
```

Предположим, что алгоритм уже правильно выбрал имплантированный 4-мер ACCT из первой последовательности и построил соответствующий *Profile*:

```
A: 1 0 0 0
C: 0 1 1 0
G: 0 0 0 0
T: 0 0 0 1
```

Теперь алгоритм готов к поиску наиболее вероятного 4-мера в силу *Profile* во второй последовательности. Проблема состоит в том, что в матрице *Profile* так много нулей, что вероятность каждого 4-мера, кроме ACCT, равна нулю. Таким образом, если ACCT не присутствует в каждой строке в *Dna*, мало шансов, что *GreedyMotifSearch* найдет имплантированный мотив. Нули в матрице профиля – это проблема, которую нужно решить.

В 1650 году, после того, как шотландцы объявили Карла II королем во время Гражданской войны в Англии, Оливер Кромвель сделал знаменитый призыв к Шотландской церкви. Призывая их увидеть ошибку королевского союза, он говорил: «Я умоляю вас, во имя Христа, помнить, что вы можете ошибаться».

Шотландцы отклонили апелляцию, и Кромвель вторгся в Шотландию в ответ. Его цитата позже вдохновила статистический принцип, называемый правилом Кромвеля, в котором говорится, что не нужно использовать вероятности 0 или 1, если речь не идет о логических утверждениях, которые могут быть истинными или ложными. Другими словами, нужно допускать небольшую вероятность для крайне маловероятных событий.

В любом наблюдаемом наборе данных существует вероятность, особенно для событий с низкой вероятностью или небольшими наборами данных, что событие с ненулевой вероятностью не происходит. Поэтому его наблюдаемая частота равна нулю; однако установление эмпирической вероятности события равным нулю представляет собой неточное упрощение, которое может вызвать проблемы. Искусственно регулируя вероятность редких событий, эти проблемы можно смягчить.

Правило Кромвеля имеет отношение к вычислению вероятности строки на основе матрицы профиля. Например, рассмотрим следующий *Profile*.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Profile** | A: | .2 | .2 | 0 | 0 | 0 | 0 | .9 | .1 | .1 | .1 | .3 | 0 |
| | C: | .1 | .6 | 0 | 0 | 0 | 0 | 0 | .4 | .1 | .2 | .4 | .6 |
| | G: | 0 | 0 | 1 | 1 | .9 | .9 | .1 | 0 | 0 | 0 | 0 | 0 |
| | T: | .7 | .2 | 0 | 0 | .1 | .1 | 0 | .5 | .8 | .7 | .3 | .4 |
| **String** | | T | C | G | T | G | G | A | T | T | T | C | C |

**Probability** $.7 \cdot .6 \cdot 1 \cdot 0 \cdot .9 \cdot .9 \cdot .9 \cdot .5 \cdot .8 \cdot .7 \cdot .4 \cdot .6 = 0$

Четвертый символ TCGTGGATTTCC заставляет *Pr*(TCGTGGATTTCC | *Profile*) равняться нулю. В результате всей строке присваивается нулевая вероятность, хотя TCGTGGATTTCC отличается от консенсусной строки только в одной позиции. В этом отношении TCGTGGATTTCC имеет такую же низкую вероятность, что и AAATCTTGGAA, которая сильно отличается от консенсусной строки.

Чтобы улучшить эту несправедливую оценку, биоинформатики часто заменяют нули небольшими числами, называемыми псевдослучайными.

Простейший подход к внедрению псевдослучайностей называется правилом преемственности Лапласа. В случае мотивов, псевдослучайности часто состоят в добавлении 1 (или некоторого другого небольшого числа) к каждому элементу *Count*(*Motifs*). Например, есть следующие мотивы, оценки и профили:

```
Motifs       T   A   A   C
             G   T   C   T
             A   C   T   A
             A   G   G   T


 Count   A:  2   1   1   1
         C:  0   1   1   1
         G:  1   1   1   0
         T:  1   1   1   2


Profile  A: 2/4 1/4 1/4 1/4
         C:  0  1/4 1/4 1/4
         G: 1/4 1/4 1/4  0
         T: 1/4 1/4 1/4 2/4
```

Правило преемственности Лапласа добавляет 1 к каждому элементу *Count*(*Motifs*), обновляя две указанные выше матрицы:

```
 Count   A: 2+1 1+1 1+1 1+1
         C: 0+1 1+1 1+1 1+1
         G: 1+1 1+1 1+1 0+1
         T: 1+1 1+1 1+1 2+1


Profile  A: 3/8 2/8 2/8 2/8
         C: 1/8 2/8 2/8 2/8
         G: 2/8 2/8 2/8 1/8
         T: 2/8 2/8 2/8 3/8
```

Единственное изменение, которое нужно внести в *GreedyMotifSearch*, чтобы исключить нули из матриц профиля, которые он создает – это заменить шестую строку псевдокода в *GreedyMotifSearch*, которая выделена ниже зеленым цветом.

```
GreedyMotifSearch(Dna, k, t)
    form a set of k-mers BestMotifs by selecting 1st k-mers in each string from Dna
    for each k-mer Motif in the first string from Dna
        Motif_1 ← Motif
        for i = 2 to t
            form Profile from motifs Motif_1, …, Motif_i - 1
            Motif_i ← Profile-most probable k-mer in the i-th string in Dna
        Motifs ← (Motif_1, …, Motif_t)
        if Score(Motifs) < Score(BestMotifs)
            BestMotifs ← Motifs
    output BestMotifs
```

После применения правила преемственности Лапласа, *GreedyMotifSearch* определяется следующим образом:

```
GreedyMotifSearch(Dna, k, t)
    form a set of k-mers BestMotifs by selecting 1st k-mers in each string from Dna
    for each k-mer Motif in the first string from Dna
        Motif_1 ← Motif
        for i = 2 to t
            apply Laplace's Rule of Succession to form Profile from motifs
                Motif_1, …, Motif_i - 1
            Motif_i ← Profile-most probable k-mer in the i-th string in Dna
        Motifs ← (Motif_1, …, Motif_t)
        if Score(Motifs) < Score(BestMotifs)
            BestMotifs ← Motifs
    output BestMotifs
```

Теперь применим правило преемственности Лапласа для поиска (4,1)-мотива ACGT, имплантированного в следующие строки *Dna*:

```
ttACCTtaac
gATGTctgtc
acgGCGTtag
ccctaACGAg
cgtcagAGGT
```

Предположим, что алгоритм уже выбрал имплантированный 4-мер ACCT из первой последовательности. Можно построить соответствующие матрицы оценок и профиля с помощью правила преемственности Лапласа:

```
Count    A: 1+1 0+1 0+1 0+1
         C: 0+1 1+1 1+1 0+1
         G: 0+1 0+1 0+1 0+1
         T: 0+1 0+1 0+1 1+1

Profile  A: 2/5 1/5 1/5 1/5
         C: 1/5 2/5 2/5 1/5
         G: 1/5 1/5 1/5 1/5
         T: 1/5 1/5 1/5 2/5
```

Используем эту матрицу профиля для вычисления вероятностей всех 4-меров во второй строке из *Dna*:

| gATG | ATGT | TGTc | GTct | Tctg | ctgt | tgtc |
|---|---|---|---|---|---|---|
| $1/5^4$ | **$4/5^4$** | $1/5^4$ | **$4/5^4$** | $2/5^4$ | $2/5^4$ | $1/5^4$ |

В второй последовательности есть два наиболее вероятных 4-мера в силу *Profile* (ATGT и GTct); предположим, что выбран имплантированный 4-мер ATGT.

Теперь есть следующие матрицы мотивов, оценок и профиля:

```
Motifs     A  C  C  T
           A  T  G  T

Count    A: 2+1 0+1 0+1 0+1
         C: 0+1 1+1 1+1 0+1
         G: 0+1 0+1 1+1 0+1
         T: 0+1 1+1 0+1 2+1

Profile  A: 3/6 1/6 1/6 1/6
         C: 1/6 2/6 2/6 1/6
         G: 1/6 1/6 2/6 1/6
         T: 1/6 2/6 1/6 3/6
```

Используем эту матрицу профиля для вычисления вероятностей всех 4-меров в третьей строке из *Dna*, acgGCGTtag:

| acgG | cgGC | gGCG | GCGT | CGTt | GTta | Ttag |
|---|---|---|---|---|---|---|
| **$12/6^4$** | $2/6^4$ | $2/6^4$ | **$12/6^4$** | $3/6^4$ | $2/6^4$ | $2/6^4$ |

Во второй последовательности (acgG и GCGT) есть два наиболее вероятных 4-мера в силу *Profile*. На этот раз предположим, что acgG выбран вместо имплантированного 4-мера GCGT.

Теперь есть следующие матрицы мотивов, оценок и профиля:

```
Motifs       A  C  C  T
             A  T  G  T
             a  c  g  G

Count     A: 3+1 0+1 0+1 0+1
          C: 0+1 2+1 1+1 0+1
          G: 0+1 0+1 2+1 1+1
          T: 0+1 1+1 0+1 2+1

Profile   A: 4/7 1/7 1/7 1/7
          C: 1/7 3/7 2/7 1/7
          G: 1/7 1/7 3/7 2/7
          T: 1/7 2/7 1/7 3/7
```

Используем этот профиль для вычисления вероятностей всех 4-меров в 4-й строке из *Dna*, ccctaACGAg:

| ccct | ccta | ctaA | taAC | aACG | ACGA | CGAg |
|---|---|---|---|---|---|---|
| $18/7^4$ | $3/7^4$ | $2/7^4$ | $1/7^4$ | $16/7^4$ | **$36/7^4$** | $2/7^4$ |

Несмотря на то, что был пропущен имплантированный 4-мер в третьей последовательности, теперь найдет имплантированный 4-мер в четвертой строке в *Dna* в качестве наиболее вероятного 4-мера ACGA в силу *Profile*.

Это дает следующие матрицы мотивов, оценок и профиля:

```
Motifs       A  C  C  T
             A  T  G  T
             a  c  g  G
             A  C  G  A

Count     A: 4+1 0+1 0+1 1+1
          C: 0+1 3+1 1+1 0+1
          G: 0+1 0+1 3+1 1+1
          T: 0+1 1+1 0+1 2+1

Profile   A: 5/8 1/8 1/8 2/8
          C: 1/8 4/8 2/8 1/8
          G: 1/8 1/8 4/8 2/8
          T: 1/8 2/8 1/8 3/8
```

Теперь используем этот профиль для вычисления вероятностей всех 4-меров в пятой строке в *Dna*, cgtcagAGGT:

| cgtc | gtca | tcag | cagA | agAG | gAGG | AGGT |
|---|---|---|---|---|---|---|
| $1/8^4$ | $8/8^4$ | $8/8^4$ | $8/8^4$ | $10/8^4$ | $8/8^4$ | **$60/8^4$** |

Наиболее вероятный 4-мер в силу *Profile* в 5-й строке в *Dna* – AGGT, имплантированный 4-мер. В результате *GreedyMotifSearch* создал следующую матрицу мотивов, которая подразумевает правильную консенсусную строку ACGT:

| | | | | |
|---|---|---|---|---|
| **Motifs** | A | C | C | T |
| | A | T | G | T |
| | a | c | g | G |
| | A | C | G | A |
| | A | G | G | T |

**Задача 8.4.** Реализовать *GreedyMotifSearch* с псевдослучайными значениями.

Вход: целые числа *k* и *t*, коллекция строк *Dna*.

Выход: коллекция строк *BestMotifs*, полученных в результате применения *GreedyMotifSearch*(*Dna, k, t*) с псевдослучайными значениями. Если на каком-либо шаге найдется более одного наиболее вероятного *k*-мера в данной строке, использовать тот, который встречается первым.

Пример входа:

3 5

GGCGTTCAGGCA

AAGAATCAGTCA

CAAGGAGTTCGC

CACGTCAATCAC

CAATAATATTCG

Пример выхода:

TTC

ATC

TTC

ATC

TTC

## Лекция 9. Случайный поиск мотивов.

Перейдем к рандомизированным алгоритмам. Французский математик и натуралист 18-го века Жорж-Луи Бюффон впервые доказал, что рандомизированные алгоритмы полезны, путем случайного бросания игл на параллельные полосы дерева, с использованием результатов этого эксперимента была аппроксимирована константа $\pi$.

Граф де Бюффон был естествоиспытателем 18-го века, чьи работы по естественной истории были популярны в то время. Однако, его первая работа была в области математики; в 1733 году он написал эссе о средневековой французской игре под названием «franc carreau». В этой игре один игрок подкидывает монету в воздух, а монета приземляется на шахматной доске. Игрок выигрывает, если монета полностью попадает в один из квадратов на доске и проигрывает в противном случае (см. Рисунок 9.1). Буффон задал естественный вопрос: какова вероятность победы игрока.

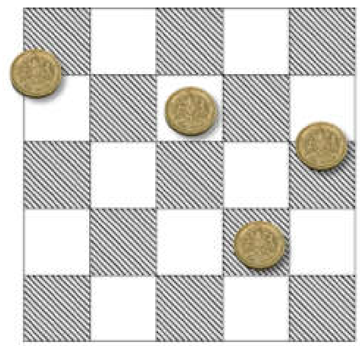


Рис. 9.1. Игра «franc carreau» с четырьмя монетами. Две из монет упали на один из квадратов шахматной доски и считаются победителями, тогда как две другие упали на границе и считаются проигравшими.

Предположим, что шахматная доска состоит всего из одного квадрата с длиной стороны 1, монета имеет радиус $r<1/2$, а центр монеты всегда приземляется внутри квадрата. Тогда игрок может выиграть только в том случае, если центр круга попадает в воображаемый центральный квадрат с длиной стороны $1-2r$ (рисунок 9.2). Предполагая, что монета приземляется равновероятно в любом месте большого квадрата, вероятность того, что монета полностью падает на меньший квадрат, определяется отношением площадей двух квадратов $(1-2r)^2$.

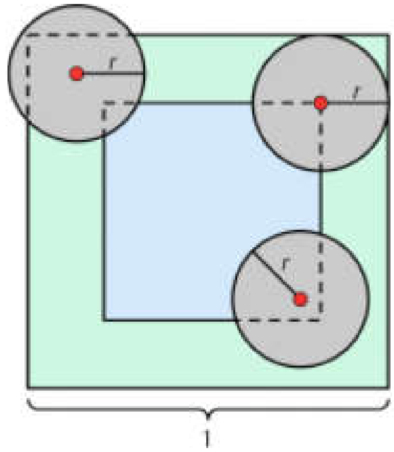



Рис. 9.2. Три монеты, показанные на одном квадрате шахматной доски (зеленый внешний квадрат); одна монета – проигравший, другая – победитель, а третья соответствует граничному случаю. Можно видеть, что если монета имеет радиус *r*, то вероятность выигрыша соответствует вероятности того, что центр монеты (показанный как красная точка) попадает в синий квадрат, длина стороны которого равна $1\text{-}2r$. Эта вероятность представляет отношение площадей квадратов, которое составляет $(1-2r)^2$.

Спустя четыре десятилетия Бюффон опубликовал статью, описывающую аналогичную игру, в которой игрок бросает иглу на пол, покрытый длинными деревянными панелями одинаковой ширины. В этой игре, которая стала известна как игла Бюффона, игрок выигрывает, если игла полностью попадает на одну из панелей. Вычисление вероятности выигрыша теперь осложняется тем, что игла описывается ориентацией в дополнение к ее положению. Тем не менее, первая игра дает представление о том, как решить эту проблему: как только зафиксировано положение центра иглы, набор различных возможных ориентаций создаст круг (рисунок 9.3).

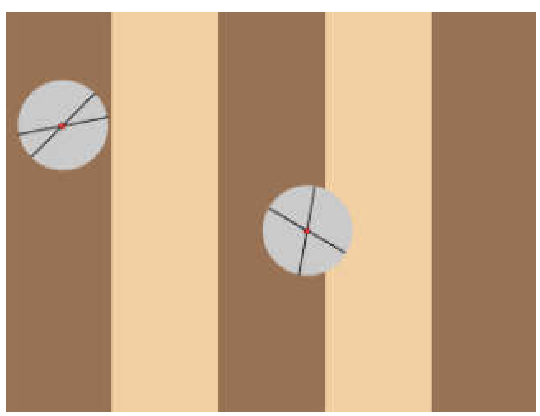


Рис. 9.3. Как только зафиксирована точка для центра иглы (показана как красная точка), коллекция возможных ориентаций очерчивает круг. В круге слева игла всегда будет находиться внутри темно-коричневой панели, независимо от ее ориентации. В круге справа одна из двух игл находится внутри темно-коричневой панели, тогда как другая пересекает границу с соседней панелью.

Вероятность победы игрока зависит от длины иглы относительно расстояния между деревянными панелями. Предположим, что обе эти длины равны 2, и найдем вероятность проигрыша вместо выигрыша. С этой целью сначала можно задать более простой вопрос: если центр иглы приземлялся бы в одном и том же месте каждый раз, какова вероятность того, что игла пересечет панель.

Чтобы ответить на этот вопрос, нарисуем панель, в которую игла попадает на координатную плоскость, при этом ось Y делит панель на две меньшие панели шириной 1 (см. Рисунок 9.4). Если центр иглы попадает в положение $(x, y)$ при $x>0$, то его ориентация может быть описана углом $\theta$, где $\theta$ от $-\pi/2$ до $\pi/2$ радиан. Если $\theta = 0$, то игла пересечет линию $y = 1$; если $\theta = \pi/2$, то игла не будет пересекать линию $y = 1$. Но что более важно, так как центральное положение иглы фиксировано, должен быть некоторый критический угол $\alpha(x)$ такой, что игла всегда касается этой линии, если $-\alpha(x) \leq \theta \leq \alpha(x)$. Если игла бросается случайным образом, то любое значение $\theta$ равновероятно, поэтому получается, что вероятность проигрыша при этом положении иглы равна $2\cdot\alpha(x)/\pi$.

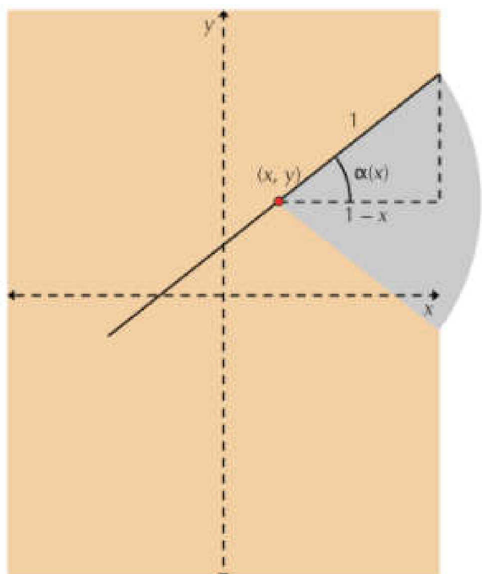



Рис. 9.4. Как только зафиксирована точка $(x, y)$ для центра иглы, существует критический угол $\alpha(x)$ такой, что все углы между $-\alpha(x)$ и $\alpha(x)$ вызовут пересечение иглой границы между панелями. В данном случае длина иглы равна ширине панели.

По тем же соображениям, игла может занимать любую позицию $x$ с равной вероятностью. Чтобы найти вероятность проигрыша, $Pr(loss)$, нужно вычислить «среднее» значение $2\cdot\alpha(x)/\pi$, так как $x$ непрерывно изменяется от -1 до 1. Это среднее может быть представлено с использованием интеграла:

$$\Pr(loss) = \frac{\int_{-1}^{1} 2\cdot\frac{\alpha(x)}{\pi}dx}{1-(-1)} = \int_{-1}^{1}\frac{\alpha(x)}{\pi}dx = 2\int_{0}^{1}\frac{\alpha(x)}{\pi}dx$$

Применяя основы тригонометрии, можно сказать, что cos $\alpha(x)$ равен $1-x$, так что $\alpha(x)$ = arccos $(1-x)$. После внесения этой замены в приведенное выше уравнение вероятность $Pr(loss)$ должна быть равна $2/\pi$. Можно видеть, что эта вероятность будет такой же, как в ситуации, когда игла будет сброшена на любое количество деревянных панелей.

В 1812 году Лаплас указал, что иглу Бюффона можно использовать для приближения $\pi$, и появился первый в мире алгоритм Монте-Карло. В частности, можно эмпирически аппроксимировать вероятность $P_e$ проигрыша, просто подбрасывая иглу в воздух тысячи раз. Как только эта эмпирическая вероятность будет вычислена, можно будет заключить, что $P_e$ примерно равно $2/\pi$, и, следовательно, $\pi \approx 2/P_e$.

Рандомизированные алгоритмы могут быть неинтуитивными, поскольку им не хватает контроля, как в традиционных алгоритмах. Некоторые рандомизированные алгоритмы обеспечивают точные решения, несмотря на то, что они полагаются на случайные принятия решений. Однако большинство рандомизированных алгоритмов, включая алгоритмы поиска мотивов, являются алгоритмами Монте-Карло. Эти алгоритмы не гарантируют точных решений, но они быстро находят приближенные решения. Из-за скорости их можно запускать много раз, что позволяет выбрать наилучшее приближение из тысяч запусков.

Ранее *Profile*(*Motifs*) определялась как матрица профиля, созданная из коллекции $k$-меров *Motifs* в *Dna*. Теперь для заданного набора строк *Dna* и произвольной матрицы *Profile* размером $4\times k$, *Motifs*(*Profile, Dna*) определяется как совокупность $k$-меров, сформированных наиболее вероятными $k$-мерами в силу *Profile* в каждой последовательности из *Dna*. Например, рассмотрим следующие *Profile* и *Dna*:

```
          A:  4/5  0    0  1/5          ttaccttaac
Profile   C:   0  3/5  1/5  0      Dna  gatgtctgtc
          G:  1/5 1/5  4/5  0           acggcgttag
          T:   0  1/5   0  4/5          ccctaacgag
                                        cgtcagaggt
```

Взятие наиболее вероятного 4-мера в силу *Profile* из каждой строки *Dna* производит следующие 4-меры (показано красным):

```
                      ttACCTtaac
                      gATGTctgtc
Motifs(Profile,Dna)   acgGCGTtag
                      ccctaACGAg
                      cgtcagAGGT
```

В общем случае, можно начать с набора случайно выбранных $k$-меров *Motifs* в *Dna*, построить *Profile*(*Motifs*) и использовать этот профиль для создания новой коллекции $k$-меров:

*Motifs*(*Profile*(*Motifs*), *Dna*)

Есть ожидания, что *Motifs*(*Profile*(*Motifs*), *Dna*) имеет лучший результат, чем оригинальная коллекция $k$-меров *Motifs*. Затем можно сформировать профильную матрицу этих $k$-меров,

*Profile*(*Motifs*(*Profile*(*Motifs*), *Dna*)),

и использовать её для формирования наиболее вероятных $k$-меров,

*Motifs*(*Profile*(*Motifs*(*Profile*(*Motifs*), *Dna*)), *Dna*).

Можно продолжить итерировать

…*Profile*(*Motifs*(*Profile*(*Motifs*(*Profile*(*Motifs*), *Dna*)), *Dna*))…

до тех пор, пока оценка сконструированных мотивов продолжает улучшаться, что и делает *RandomizedMotifSearch*. Чтобы реализовать этот алгоритм, нужно случайно выбрать начальную коллекцию $k$-меров, которые формируют матрицу мотивов *Motifs*. Для этого понадобится генератор случайных чисел (обозначается *Random*($N$)), который с равной вероятностью может возвращать любое целое число от 1 до $N$.

```
RandomizedMotifSearch(Dna, k, t)
    randomly select k-mers Motifs = (Motif1, …, Motift) in each string
        from Dna
    BestMotifs ← Motifs
    while forever
        Profile ← Profile(Motifs)
        Motifs ← Motifs(Profile, Dna)
        if Score(Motifs) < Score(BestMotifs)
            BestMotifs ← Motifs
        else
            return BestMotifs
```

Поскольку один запуск *RandomizedMotifSearch* может генерировать довольно плохой набор мотивов, биоинформатики обычно используют этот алгоритм тысячи раз. При каждом запуске они начинают с нового случайно выбранного набора $k$-меров, выбирая в итоге лучший набор $k$-меров, найденный во всех этих прогонах.

**Задача 9.1.** Реализовать *RandomizedMotifSearch*.

Вход: целые числа $k$ и $t$, коллекция строк *Dna*.

Выход: коллекция строк *BestMotifs*, полученных в результате применения *RandomizedMotifSearch* (*Dna, k, t*) 1000 раз с псевдослучайными значениями.

Пример входа:

8 5

CGCCCCTCTCGGGGGTGTTCAGTAAACGGCCA

GGGCGAGGTATGTGTAAGTGCCAAGGTGCCAG

TAGTACCGAGACCGAAAGAAGTATACAGGCGT

TAGATCAAGTTTCAGGTGCACGTCGGTGAACC

AATCCACCAGCTCCACGTGCAATGTTGGCCTA

Пример выхода:

TCTCGGGG

CCAAGGTG

TACAGGCG

TTCAGGTG

TCCACGTG

На первый взгляд кажется, что *RandomizedMotifSearch* не может найти ничего полезного. Чтобы исследовать *RandomizedMotifSearch*, запустим его на пяти коротких строках с имплантированным (4,1)-мотивом ACGT (показан синим цветом) и представим, что он выбирает следующие 4-меры *Motifs* (выделенные жирным шрифтом) на первой итерации. Как и ожидалось, он пропускает имплантированный мотив почти в каждой строке.

| | |
|---|---|
| | ttACCT**taac** |
| | gAT**GTct**gtc |
| *Dna* | **ccgG**CGTtag |
| | c**acta**ACGAg |
| | cgtcag**AGGT** |

Теперь строим профильную матрицу *Profile*(*Motifs*) выбранных 4-меров:

| | | | | |
|---|---|---|---|---|
| | **t** | **a** | **a** | **c** |
| | **G** | **T** | **c** | **t** |
| *Motifs* | **c** | **c** | **g** | **G** |
| | **a** | **c** | **t** | **a** |
| | **A** | **G** | **G** | **T** |

| | | | | | |
|---|---|---|---|---|---|
| | A: | 0.4 | 0.2 | 0.2 | 0.2 |
| *Profile*(*Motifs*) | C: | 0.2 | 0.4 | 0.2 | 0.2 |
| | G: | 0.2 | 0.2 | 0.4 | 0.2 |
| | T: | 0.2 | 0.2 | 0.2 | 0.4 |

и вычисляет вероятности каждого 4-мера в *Dna* на основе этой профильной матрицы. Например, вероятность первого 4-мера в первой строке *Dna* равна Pr(ttAC | *Profile*) = 0,2 · 0,2 · 0,2 · 0,2 = 0,0016.

| | | | | | | |
|---|---|---|---|---|---|---|
| ttAC (.0016) | tACC (.0016) | ACCT (.0128) | CCTt (.0064) | CTta (.0016) | Ttaa (.0016) | taac (.0016) |
| gATG (.0016) | ATGT (.0128) | TGTc (.0016) | GTct (.0032) | Tctg (.0032) | ctgt (.0032) | tgtc (.0016) |
| ccgG (.0064) | cgGC (.0036) | gGCG (.0016) | GCGT (.0128) | CGTt (.0032) | GTta (.0016) | Ttag (.0016) |
| cact (.0032) | acta (.0064)) | ctaA (.0016) | taAC (.0016) | aACG (.0032) | ACGA (.0128) | CGAg (.0016) |
| cgtc (.0016) | gtca (.0016) | tcag (.0016) | cagA (.0032) | agAG (.0032) | gAGG (.0032) | AGGT (.0128) |

Максимальные вероятности в каждой строке показаны зеленым цветом ниже.

| | | | | | | |
|---|---|---|---|---|---|---|
| ttAC (.0016) | tACC (.0016) | ACCT (**.0128**) | CCTt (.0064) | CTta (.0016) | Ttaa (.0016) | taac (.0016) |
| gATG (.0016) | ATGT (**.0128**) | TGTc (.0016) | GTct (.0032) | Tctg (.0032) | ctgt (.0032) | tgtc (.0016) |
| ccgG (.0064) | cgGC (.0036) | gGCG (.0016) | GCGT (**.0128**) | CGTt (.0032) | GTta (.0016) | Ttag (.0016) |
| cact (.0032) | acta (.0064)) | ctaA (.0016) | taAC (.0016) | aACG (.0032) | ACGA (**.0128**) | CGAg (.0016) |
| cgtc (.0016) | gtca (.0016) | tcag (.0016) | cagA (.0032) | agAG (.0032) | gAGG (.0032) | AGGT (**.0128**) |

Выберем наиболее вероятный 4-мер в каждой строке выше как новую коллекцию *Motifs* (показано ниже). Обратите внимание, что эта коллекция захватила все пять имплантированных мотивов в *Dna*.

```
ttACCTtaac
gATGTctgtc
ccgGCGTtag
cactaACGAg
cgtcagAGGT
```

Рассмотренный набор имплантированных мотивов привел к следующей матрице профиля.

```
A: 0.8 0.0 0.0 0.2
C: 0.0 0.6 0.2 0.0
G: 0.2 0.2 0.8 0.0
T: 0.0 0.2 0.0 0.8
```

Если строки в *Dna* были действительно случайными, то ожидается, что все нуклеотиды в выбранных *k*-мерах будут одинаково вероятными, в результате чего ожидается *Profile*, в котором каждая запись составляет приблизительно 0.25:

```
A: 0.25 0.25 0.25 0.25
```

```
C: 0.25 0.25 0.25 0.25
G: 0.25 0.25 0.25 0.25
T: 0.25 0.25 0.25 0.25
```

Такой однородный профиль бесполезен для поиска мотивов, потому что ни одна строка не является более вероятной, чем любая другая в соответствии с этим профилем, и он не дает никаких подсказок о том, как выглядит имплантированный мотив.

Если бы повезло, и имплантированные *k*-меры *Motifs* были бы выбраны с самого начала, в результате появилась бы первая из двух профильных матриц выше. На практике, скорее всего, получится профиль где-то между этими двумя крайностями, например,

A: **0.4** 0.2 0.2 0.2

C: 0.2 **0.4** 0.2 0.2

G: 0.2 0.2 **0.4** 0.2

T: 0.2 0.2 0.2 **0.4**

Эта матрица профиля уже начала указывать на имплантированный мотив ACGT, т.е. ACGT является наиболее вероятным 4-мером, который может быть сгенерирован этим профилем. *RandomizedMotifSearch* разработан так, что последующие шаги имеют хорошие шансы привести к этому имплантированному мотиву (хоть это и не точно).

Можно рассмотреть следующий аргумент. Если бы *Dna* были случайными строками, то *RandomizedMotifSearch* начинался бы с почти единообразного профиля, и работать было бы не с чем. Однако, ключевое замечание состоит в том, что строки в *Dna* не являются случайными, потому что они включают имплантированный мотив. Эти множественные вхождения одного и того же мотива могут создавать смещение в профильной матрице, направляя ее от однородного профиля к имплантированному мотиву. Например, рассмотрим снова исходные случайно выбранные *k*-меры (выделенные жирным шрифтом ниже):

ttACCT**taac**

gAT**GTct**gtc

**ccgG**CGTtag

c**acta**ACGAg

cgtcag**AGGT**

4-мер AGGT в последней строке случайно захватил имплантированный мотив. Фактически профиль, образованный из оставшихся 4-меров (taac, GTct, ccgG, acta), является однородным.

Хотя вероятность того, что случайно выбранные *k*-меры захватят все имплантированные мотивы, пренебрежимо мала, вероятность того, что они захватят хотя бы один имплантированный мотив, значительна. Даже в случае сложных мотивов, для которых эта вероятность мала, можно многократно запускать *RandomizedMotifSearch*, так что он почти наверняка «поймает» хотя бы один имплантированный мотив, создав таким образом статистический уклон, указывающий на правильный мотив.

К сожалению, захват одного имплантированного мотива часто является недостаточным для приведения *RandomizedMotifSearch* к оптимальному решению. Поэтому, поскольку количество исходных позиций *k*-меров огромно, стратегия случайного выбора мотивов часто бывает не столь успешной, как в простом примере. Вероятность того, что эти случайно выбранные *k*-меры смогут привести к оптимальному решению, относительно невелика.

## Лекция 10. Семплирование по Гиббсу.

Стоит напомнить, что *RandomizedMotifSearch* может изменять все *t* строк *Motifs* за одну итерацию. Эта стратегия может оказаться безрассудной, поскольку некоторые правильные мотивы (зафиксированные в *Motifs*) могут быть отброшены на следующей итерации. *GibbsSampler* – более осторожный итерационный алгоритм, который отбрасывает один *k*-мер из текущего набора мотивов на каждой итерации и решает либо сохранить его, либо заменить на новый. Таким образом, этот алгоритм движется с большей осторожностью в пространстве всех мотивов, как показано на рисунке 10.1.


Рис. 10.1. *RandomizedMotifSearch* (слева) может изменять все *k*-меры на одном шаге, *GibbsSampler* (справа) может изменять один *k*-мер на одном шаге.

Подобно *RandomizedMotifSearch*, *GibbsSampler* начинает со случайно выбранных *k*-меров в каждой из последовательностей ДНК, но на каждой итерации он делает случайный, а не детерминированный выбор. Он использует случайно выбранные *k*-меры ($Motif_1$, ..., $Motif_t$), чтобы создать другой (с надеждой на лучший результат) набор *k*-меров. В отличие от *RandomizedMotifSearch* (который детерминировано определяет новые мотивы как *Motifs*(*Profile*(*Motifs*), *Dna*)), *GibbsSampler* случайным образом выбирает целое число *i* между 1 и *t*, а затем случайным образом изменяет только один *k*-мер $Motif_i$.

Чтобы описать, как *GibbsSampler* обновляет *Motifs*, понадобится несколько более продвинутый генератор случайных чисел. Для заданного распределения вероятностей ($p_1$, ..., $p_n$), этот генератор случайных чисел, обозначенный *Random*($p_1$, ..., $p_n$), моделирует неравномерное распределение для *n* элементов и возвращает целое число *i* с вероятностью $p_i$. Например, стандартное равномерное распределение для 6 элементов представляется генератором случайных чисел *Random*(1/6, 1/6, 1/6, 1/6, 1/6, 1/6), тогда как неравномерное распределение может представляться генератором случайных чисел *Random*(0.1, 0.2, 0.3, 0.05, 0.1, 0.25). *GibbsSampler* далее обобщает генератор случайных чисел, используя функцию *Random*($p_1$, ..., $p_n$), определенную для любого набора неотрицательных чисел, т.е. не обязательно удовлетворяющую условию, что сумма $p_i$ равна 1. Если $p_i$ суммируется как некоторое $C > 0$, то *Random*($p_1$, ..., $p_n$) определяется как *Random*($p_1$/C, ..., $p_n$/C), где ($p_1$/C, ..., $p_n$/C) – распределение вероятности. Например, для (0.1, 0.2, 0.3) с 0.1 + 0.2 + 0.3 = 0.6,

$$Random(0.1, 0.2, 0.3) = Random\left(\frac{0.1}{0.6}, \frac{0.2}{0.6}, \frac{0.3}{0.6}\right) = Random\left(\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\right)$$

Ранее было определено понятие наиболее вероятного *k*-мера в силу *Profile* в строке. Теперь определим генерируемый *Profile* *k*-мер в строке *Text*. Для каждого *k*-мера *Pattern* в *Text* вычисляется вероятность *Pr*(*Pattern* | *Profile*), в результате получится $n = |Text|-k+1$ вероятностей ($p_1$, ..., $p_n$). Эти вероятности не обязательно суммируются как 1, но можно по-прежнему сформировать на их основе генератор случайных чисел *Random*($p_1$, ..., $p_n$). *GibbsSampler* использует этот генератор случайных чисел для выбора произвольно сформированного *k*-мера в силу *Profile* на каждом шаге: если выбрано число *i*, произвольно сформированный *k*-мер в силу *Profile* определяется как *i*-й *k*-мер в *Text*. Псевдокод ниже повторяет эту процедуру *N* раз, однако, на практике *GibbsSampler* зависит от различных правил остановки.

```
GIBBSSAMPLER(Dna, k, t, N)
    randomly select k-mers Motifs = (Motif1, …, Motift) in each string
        from Dna
    BestMotifs ← Motifs
    for j ← 1 to N
        i ← Random(t)
        Profile ← profile matrix constructed from all strings in Motifs
                     except for Motifi
        Motifi ← Profile-randomly generated k-mer in the i-th sequence
        if Score(Motifs) < Score(BestMotifs)
            BestMotifs ← Motifs
    return BestMotifs
```

**Задача 10.1.** Реализовать *GibbsSampler*.

Вход: целые числа *k*, *t* и *N*, коллекция строк *Dna*.

Выход: коллекция строк *BestMotifs*, полученных в результате применения *GibbsSampler*(*Dna, k, t, N*) 20 раз с псевдослучайными значениями.

Пример входа:

8 5 100

CGCCCCTCTCGGGGGTGTTCAGTAAACGGCCA

GGGCGAGGTATGTGTAAGTGCCAAGGTGCCAG

TAGTACCGAGACCGAAAGAAGTATACAGGCGT

TAGATCAAGTTTCAGGTGCACGTCGGTGAACC

AATCCACCAGCTCCACGTGCAATGTTGGCCTA

Пример выхода:

TCTCGGGG

CCAAGGTG

TACAGGCG

TTCAGGTG

TCCACGTG

Проиллюстрируем, как *GibbsSampler* работает на тех же строках *Dna*, которые были рассмотрены ранее, с имплантированным (4,1)-мотивом ACGT. Пусть на начальном этапе алгоритм выбрал следующие 4-меры (выделены полужирным шрифтом):

```
        ttACCTtaac
        gATGTctgtc
Dna     ccgGCGTtag
        cactaACGAg
        cgtcagAGGT
```

и на первой итерации он случайно выбрал третью строку для удаления:

```
        ttACCTtaac
        gATGTctgtc
Dna     ..........
        cactaACGAg
        cgtcagAGGT
```

Это приводит к следующим матрицам мотивов, оценок и профиля.

```
Motifs        t    a    a    c
              G    T    c    t
              a    c    t    a
              A    G    G    T

          A:  2    1    1    1
Count     C:  0    1    1    1
          G:  1    1    1    0
          T:  1    1    1    2

          A: 2/4  1/4  1/4  1/4
Profile   C:  0   1/4  1/4  1/4
          G: 1/4  1/4  1/4   0
          T: 1/4  1/4  1/4  2/4
```

Стоит заметить, что матрица профиля немного более консервативна, чем однородный профиль, что заставляет задаться вопросом, есть ли в данном случае шанс направиться к имплантированному мотиву.

Теперь используем эту матрицу профиля для вычисления вероятностей всех 4-меров в удаленной строке ccgGCGTtag:

| ccgG | cgGC | gGCG | GCGT | CGTt | GTta | Ttag |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1/128 | 0 | 1/256 | 0 |

Все, кроме двух из этих вероятностей, равны нулю. Эта ситуация похожа на ту, которая была с *GreedyMotifSearch*, и поэтому нужно увеличить нулевые вероятности на небольшие псевдослучайности.

Применение правила преемственности Лапласа дает следующие обновленные матрицы оценок и профилей:

```
          A:   3     2     2     2
  Count   C:   1     2     2     2
          G:   2     2     2     1
          T:   2     2     2     3

          A:  3/8   2/8   2/8   2/8
Profile   C:  1/8   2/8   2/8   2/8
          G:  2/8   2/8   2/8   1/8
          T:  2/8   2/8   2/8   3/8
```

После добавления псевдослучайностей, вероятности каждого 4-мера в удаленной строке ccgGCGTtag пересчитываются следующим образом:

| ccgG | cgGC | gGCG | GCGT | CGTt | GTta | Ttag |
|---|---|---|---|---|---|---|
| $4/8^4$ | $8/8^4$ | $8/8^4$ | $24/8^4$ | $12/8^4$ | $16/8^4$ | $8/8^4$ |

Поскольку эти вероятности суммируются как C = 80/84, распределение для семи элементов представлено генератором случайных чисел

$$Random\left(\frac{4/8^4}{80/8^4}, \frac{8/8^4}{80/8^4}, \frac{8/8^4}{80/8^4}, \frac{24/8^4}{80/8^4}, \frac{12/8^4}{80/8^4}, \frac{16/8^4}{80/8^4}, \frac{8/8^4}{80/8^4}\right)$$
$$= Random\left(\frac{4}{80}, \frac{8}{80}, \frac{8}{80}, \frac{24}{80}, \frac{12}{80}, \frac{16}{80}, \frac{8}{80}\right).$$

Предположим, что после применения этого генератора получается произвольно сформированный 4-мер GCGT (четвертый 4-мер в удаленной последовательности). Удаленная строка ccgGCGTtag теперь добавляется обратно в коллекцию мотивов, а GCGT заменяет ранее выбранный ccgG в третьей последовательности:

```
ttACCTtaac
gATGTctgtc
ccgGCGTtag
cactaACGAg
cgtcagAGGT
```

Снова генерируем равномерное распределение для пяти элементов и случайным образом выбираем первую строку из *Dna* для удаления:

```
..........
gATGTctgtc
ccgGCGTtag
cactaACGAg
cgtcagAGGT
```

После построения матриц мотивов и профиля получится следующее:

|  |  |  |  |  |
|---|---|---|---|---|
| *Motifs* | G | T | c | t |
|  | G | C | G | T |
|  | a | c | t | a |
|  | A | G | G | T |

|  |  |  |  |  |  |
|---|---|---|---|---|---|
|  | A: | 2/4 | 0 | 0 | 1/4 |
| *Profile* | C: | 0 | 2/4 | 1/4 | 0 |
|  | G: | 2/4 | 1/4 | 2/4 | 0 |
|  | T: | 0 | 1/4 | 1/4 | 3/4 |

Матрица профиля выглядит более «предвзятой» по отношению к имплантированному мотиву, чем предыдущая матрица профиля.

Обновляем матрицы мотивов и профиля с псевдослучайными значениями:

|  |  |  |  |  |  |
|---|---|---|---|---|---|
|  | A: | 3 | 1 | 1 | 2 |
| *Count* | C: | 1 | 3 | 2 | 1 |
|  | G: | 3 | 2 | 3 | 1 |
|  | T: | 1 | 2 | 2 | 4 |

|  |  |  |  |  |  |
|---|---|---|---|---|---|
|  | A: | 3/8 | 1/8 | 1/8 | 2/8 |
| *Profile* | C: | 1/8 | 3/8 | 2/8 | 1/8 |
|  | G: | 3/8 | 2/8 | 3/8 | 1/8 |
|  | T: | 1/8 | 2/8 | 2/8 | 4/8 |

Вычисляем вероятности всех 4-меров в удаленной строке ttACCTtaac:

| ttAC | tACC | ACCT | CCTt | CTta | Ttaa | taac |
|---|---|---|---|---|---|---|
| $2/8^4$ | $2/8^4$ | $72/8^4$ | $24/8^4$ | $8/8^4$ | $4/8^4$ | $1/8^4$ |

После генерирования распределения для семи элементов, приходим к произвольно сформированному *k*-меру ACCT, который добавляется к коллекции *Motifs* (выделено жирным шрифтом):

tt**ACCT**taac gAT**GTct**gtc ccg**GCGT**tag c**acta**ACGAg cgtcag**AGGT**

После генерирования распределения для пяти элементов, произвольно выбираем четвертую строку для удаления, добавляем псевдослучайные величины и создаем результирующие матрицы оценок и профиля:

```
                    ttACCTtaac
                    gATGTctgtc
Dna                 ccgGCGTtag
                    ..........
                    cgtcagAGGT

Motifs          A    C    C    T
                G    T    c    t
                G    C    G    T
                A    G    G    T

           A:   3    1    1    1
Count      C:   1    3    3    1
           G:   3    2    3    1
           T:   1    2    1    5

           A:  3/8  1/8  1/8  1/8
Profile    C:  1/8  3/8  3/8  1/8
           G:  3/8  2/8  3/8  1/8
           T:  1/8  2/8  1/8  5/8
```

Вычисляем вероятности всех 4-меров в удаленной строке cactaACGAg:

| cact | acta | ctaA | taAC | aACG | ACGA | CGAg |
|---|---|---|---|---|---|---|
| $15/8^4$ | $9/8^4$ | $2/8^4$ | $1/8^4$ | $9/8^4$ | $27/8^4$ | $2/8^4$ |

Нужно сгенерировать распределение для семи элементов, чтобы создать случайный 4-мер в силу *Profile*. Предполагая наиболее вероятный сценарий, в котором выбирается ACGA, обновляем выбранные 4-меры следующим образом:

```
ttACCTtaac
gATGTctgtc
ccgGCGTtag
cactaACGAg
cgtcagAGGT
```

Можно видеть, что алгоритм начинает сходиться. Последующая итерация будет захватывать все имплантированные мотивы после того, как будет выбрана вторая строка в *Dna* (когда неправильный 4-мер GTct, скорее всего, изменится на имплантированный (4,1)-мотив ATGT).

Хотя *GibbsSampler* хорошо работает во многих случаях, он может сходиться к субоптимальному решению, особенно для сложных проблем поиска с неуловимыми мотивами. Локальный оптимум – это решение, оптимальное в рамках небольшого соседнего набора решений, что контрастирует с глобальным оптимумом или оптимальным решением по всем возможным решениям. Поскольку *GibbsSampler* исследует только небольшой набор решений, он может «застревать» в локальном оптимуме. По этой причине, аналогично *RandomizedMotifSearch*, его следует запускать много раз с надеждой, что один из этих запусков будет создавать мотивы с наилучшей оценкой. Однако, сближение с локальным оптимумом является лишь одним из многих вопросов, которые нужно рассмотреть в поиске мотивов.

Поиск мотивов становится затруднительным, если распределение нуклеотидов в образце искажено. В этом случае поиск $k$-меров с минимальной оценкой или энтропией может привести к биологически нерелевантному мотиву, составленному из наиболее частых нуклеотидов в образце. Например, если A имеет частоту 85%, а T, G и C имеют частоты 5%, то $k$-мер AA ... AA может представлять собой мотив с минимальным счетом, таким образом скрывая биологически релевантные мотивы. Например, релевантный мотив GCCG со счетом 5 в приведенном ниже примере теряется из-за 4-мера aaaa со счетом 1.

```
taaaaGtCGa
acGCtGaaaa
aaaaGCCtat
acCCGaataa
agaaaaGgCG
```

Чтобы найти биологически релевантные мотивы в образцах с смещенными нуклеотидными частотами, можно использовать обобщение энтропии под названием «относительная энтропия».

Для заданного набора строк *Dna*, относительная энтропия матрицы профиля размером $4 \times k$ $P = (p_{r,j})$ определяется как

$$\sum_{j=1}^{k} \sum_{r \in A,C,G,T} p_{r,j} \cdot \log_2\left(\frac{p_{r,j}}{b_r}\right) = \sum_{j=1}^{k} \sum_{r \in A,C,G,T} p_{r,j} \cdot \log_2(p_{r,j}) - \sum_{j=1}^{k} \sum_{r \in A,C,G,T} p_{r,j} \cdot \log_2(b_r)$$

где $b_r$ – частота нуклеотида $r$ в *Dna*. Заметим, что сумма в выражении для энтропии имеет отрицательный знак $\left(-\sum_{j=1}^{k} \sum_{r \in A,C,G,T} p_{r,j} \cdot \log_2(p_{r,j})\right)$, тогда как сумма в левой части уравнения относительной энтропии не имеет этого отрицательного знака. Поэтому, несмотря на минимизацию энтропии матрицы мотивов, теперь нужно максимизировать относительную энтропию.

В приведенной выше формуле относительной энтропии слагаемое $-\sum_{j=1}^{k} \sum_{r \in A,C,G,T} p_{r,j} \cdot \log_2(b_r)$ называется кросс-энтропией матрицы профиля *P*; относительная энтропия матрицы профиля – это разница между кросс-энтропией и энтропией профиля. Например, относительная энтропия для мотива GCCG в примере ниже равна 9.85 – 3.53 = 6.32. В этом примере $b_A$=0.5, $b_C$=0.18, $b_G$=0.2 и $b_T$=0.12.

```
          G      t      C      G
          G      C      t      G
          G      C      C      t
          c      C      C      G
          G      g      C      G

     A:  0.0    0.0    0.0    0.0
     C:  0.2    0.6    0.8    0.0
     G:  0.8    0.2    0.0    0.8
     T:  0.0    0.2    0.2    0.2

      entropy:  0.72 + 1.37 + 0.72 + 0.72 =  3.53
cross-entropy:   2.35 + 2.56 + 2.47 + 2.47 = 9.85
```

Для более консервативного, но нерелевантного мотива aaaa относительная энтропия равна 4.18 – 0.72 = 3.46, как показано ниже. Таким образом, GCCG проигрывает aaaa в отношении энтропии, но выигрывает в случае относительной энтропии.

Еще одна сложность в обнаружении мотивов состоит в том, что многие мотивы лучше всего представлены в разных алфавитах, чем в алфавите из 4 нуклеотидов. Пусть W обозначает либо A, либо T, S обозначает либо G, либо C, K обозначает либо G, либо T, а Y обозначает либо C, либо T. Теперь рассмотрим мотив CSKWYWWATKWATYYK, который представляет мотив CSRE у дрожжей (показан ниже). Этот сильный мотив в гибридном алфавите соответствует 211 различным мотивам в стандартном 4-буквенном алфавите нуклеотидов. Однако, каждый из этих 211 мотивов слишком слаб, чтобы их можно было найти рассмотренными алгоритмами.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C | G/C | G/T | T/A | C/T | G/C | C/G | A | T | G/T | C/G | A | T | C/T | C/T | G/T |

Туберкулез (TB) представляет собой инфекционное заболевание, которое вызывается бактерией *Mycobacterium tuberculosis* (MTB). Несмотря на то, что распространение туберкулеза значительно сократилось из-за антибиотиков, сейчас появляются штаммы, которые противостоят всем доступным методам лечения. MTB является успешным как патоген, потому что он может сохраняться у людей в течение десятилетий, не вызывая болезни; на самом деле треть населения мира имеет скрытые инфекции MTB, при которых MTB находится в состоянии покоя в организме хозяина и может активироваться или не активизироваться позднее. Широкая распространенность скрытых инфекций затрудняет борьбу с эпидемиями туберкулеза. Поэтому биологам интересно узнать, что заставляет болезнь скрываться и как MTB активируется внутри хозяина.

Остается неясным, почему МТВ может оставаться латентным так долго и как он выживает во время латентности. Сопротивление латентного ТВ антибиотикам подразумевает, что МТВ может иметь способность прекращать экспрессию большинства генов и оставаться бездействующей. Гибернация в бактериях называется споруляцией, потому что многие бактерии образуют защитные и метаболически неактивные споры, которые могут выживать в тяжелых условиях, позволяя бактериям сохраняться в окружающей среде до тех пор, пока условия не улучшатся.

Гипоксия или нехватка кислорода часто связаны со скрытыми формами туберкулеза. Биологи обнаружили, что МТВ становится бездействующим в среде с низким содержанием кислорода, предположительно с надеждой на то, что легкие хозяина будут восстановлены настолько, чтобы потенциально распространить болезнь в будущем. Поскольку МТВ демонстрирует замечательную способность выживать в течение многих лет без кислорода, важно идентифицировать гены МТВ, ответственные за развитие скрытого состояния в гипоксических условиях. Биологи заинтересованы в поиске транскрипционного фактора, который «ощущает» нехватку кислорода и запускает генетическую программу, которая влияет на экспрессию многих генов, позволяя МТВ адаптироваться к гипоксии.

В 2003 году биологи обнаружили регулятор выживаемости покоя (*DosR*), транскрипционный фактор, который регулирует многие гены, чья активность резко изменяется в гипоксических условиях. Тем не менее, остается неясным, как *DosR* регулирует эти гены, и сайт связывания транскрипционного фактора остается неизвестным. В попытке решить эту головоломку биологи провели эксперимент с ДНК-матрицей и обнаружили 25 генов, уровень экспрессии которых значительно изменяется в гипоксических условиях. Учитывая вышерасположенные области этих генов, каждый из которых имеет длину в 250 нуклеотидов, можно попытаться обнаружить «скрытое сообщение», которое *DosR* использует для контроля экспрессии этих генов.

Для упрощения задачи были выбраны только 10 из 25 генов (представлены ниже). Можно попытаться определить мотивы в этом наборе данных, используя арсенал разработанных ранее инструментов поиска мотивов.

```
GCGCCCCGCCCGGACAGCCATGCGCTAACCCTGGCTTCGATGGCGCCGGCTCAGTTAGGGCCGGAAGTCCCCAATGTG
GCAGACCTTTCGCCCCTGGCGGACGAATGACCCCAGTGGCCGGGACTTCAGGCCCTATCGGAGGGCTCCGGCGCGGTG
GTCGGATTTGTCTGTGGAGGTTACACCCCAATCGCAAGGATGCATTATGACCAGCGAGCTGAGCCTGGTCGCCACTGG
AAAGGGGAGCAACATC
CCGATCGGCATCACTATCGGTCCTGCGGCCGCCCATAGCGCTATATCCGGCTGGTGAAATCAATTGACAACCTTCGAC
TTTGAGGTGGCCTACGGCGAGGACAAGCCAGGCAAGCCAGCTGCCTCAACGCGCGCCAGTACGGGTCCATCGACCCGC
GGCCCACGGGTCAAACGACCCTAGTGTTCGCTACGACGTGGTCGTACCTTCGGCAGCAGATCAGCAATAGCACCCCGA
CTCGAGGAGGATCCCG
ACCGTCGATGTGCCCGGTCGCGCCGCGTCCACCTCGGTCATCGACCCCACGATGAGGACGCCATCGGCCGCGACCAAG
CCCCGTGAAACTCTGACGGCGTGCTGGCCGGGCTGCGGCACCTGATCACCTTAGGGCACTTGGGCCACCACAACGGGC
CGCCGGTCTCGACAGTGGCCACCACCACACAGGTGACTTCCGGCGGGACGTAAGTCCCTAACGCGTCGTTCCGCACGC
GGTTAGCTTTGCTGCC
```

```
GGGTCAGGTATATTTATCGCACACTTGGGCACATGACACACAAGCGCCAGAATCCCGGACCGAACCGAGCACCGTGGG
TGGGCAGCCTCCATACAGCGATGACCTGATCGATCATCGGCCAGGGCGCCGGGCTTCCAACCGTGGCCGTCTCAGTAC
CCAGCCTCATTGACCCTTCGACGCATCCACTGCGCGTAAGTCGGCTCAACCCTTTCAAACCGCTGGATTACCGACCGC
AGAAAGGGGGCAGGAC
GTAGGTCAAACCGGGTGTACATACCCGCTCAATCGCCCAGCACTTCGGGCAGATCACCGGGTTTCCCCGGTATCACCA
ATACTGCCACCAAACACAGCAGGCGGGAAGGGGCGAAAGTCCCTTATCCGACAATAAAACTTCGCTTGTTCGACGCCC
GGTTCACCCGATATGCACGGCGCCCAGCCATTCGTGACCGACGTCCCCAGCCCCAAGGCCGAACGACCCTAGGAGCCA
CGAGCAATTCACAGCG
CCGCTGGCGACGCTGTTCGCCGGCAGCGTGCGTGACGACTTCGAGCTGCCCGACTACACCTGGTGACCACCGCCGACG
GGCACCTCTCCGCCAGGTAGGCACGGTTTGTCGCCGGCAATGTGACCTTTGGGCGCGGTCTTGAGGACCTTCGGCCCC
ACCCACGAGGCCGCCGCCGGCCGATCGTATGACGTGCAATGTACGCCATAGGGTGCGTGTTACGGCGATTACCTGAAG
GCGGCGGTGGTCCGGA
GGCCAACTGCACCGCGCTCTTGATGACATCGGTGGTCACCATGGTGTCCGGCATGATCAACCTCCGCTGTTCGATATC
ACCCCGATCTTTCTGAACGGCGGTTGGCAGACAACAGGGTCAATGGTCCCCAAGTGGATCACCGACGGGCGCGGACAA
ATGGCCCGCGCTTCGGGGACTTCTGTCCCTAGCCCTGGCCACGATGGGCTGGTCGGATCAAAGGCATCCGTTTCCATC
GATTAGGAGGCATCAA
GTACATGTCCAGAGCGAGCCTCAGCTTCTGCGCAGCGACGGAAACTGCCACACTCAAAGCCTACTGGGCGCACGTGTG
GCAACGAGTCGATCCACACGAAATGCCGCCGTTGGGCCGCGGACTAGCCGAATTTTCCGGGTGGTGACACAGCCCACA
TTTGGCATGGGACTTTCGGCCCTGTCCGCGTCCGTGTCGGCCAGACAAGCTTTGGGCATTGGCCACAATCGGGCCACA
ATCGAAAGCCGAGCAG
GGCAGCTGTCGGCAACTGTAAGCCATTTCTGGGACTTTGCTGTGAAAAGCTGGGCGATGGTTGTGGACCTGGACGAGC
CACCCGTGCGATAGGTGAGATTCATTCTCGCCCTGACGGGTTGCGTCTGTCATCGGTCGATAAGGACTAACGGCCCTC
AGGTGGGGACCAACGCCCCTGGGAGATAGCGGTCCCCGCCAGTAACGTACCGCTGAACCGACGGGATGTATCCGCCCC
AGCGAAGGAGACGGCG
TCAGCACCATGACCGCCTGGCCACCAATCGCCCGTAACAAGCGGGACGTCCGCGACGACGCGTGCGCTAGCGCCGTGG
CGGTGACAACGACCAGATATGGTCCGAGCACGCGGGCGAACCTCGTGTTCTGGCCTCGGCCAGTTGTGTAGAGCTCAT
CGCTGTCATCGAGCGATATCCGACCACTGATCCAAGTCGGGGGCTCTGGGGACCGAAGTCCCCGGGCTCGGAGCTATC
GGACCTCACGATCACC
```

Запуская *MedianString* и *RandomizedMotifSearch* для *k* от 8 до 12, возвращаем консенсусные строки, показанные на рисунке 10.1.

| ***k*** | **consensus** | **score** | ***k*** | **consensus** | **score** |
|---|---|---|---|---|---|
| 8 | CATCGGCC | 11 | 8 | CCGACGGG | 13 |
| 9 | GGCGGGGAC | 16 | 9 | CCATCGGCC | 16 |
| 10 | GGTGGCCACC | 19 | 10 | CCATCGGCCC | 21 |
| 11 | GGACTTCCGGC | 20 | 11 | ACCTTCGGCCC | 25 |
| 12 | GGACTTCCGGCC | 23 | 12 | GGACCAACGGCC | 28 |

Рис. 10.1. Консенсусные строки, возвращаемые *MedianString* (слева) и *RandomizedMotifSearch* (справа).

Несмотря на то, что консенсусные строки, возвращаемые *RandomizedMotifSearch*, обычно отклоняются от медианных строк, консенсусная строка длиной 12 (GGACCAACGGCC, с результатом 28) очень похожа на медианную строку (GGACTTCCGGCC с результатом 23).

Хотя мотивы, возвращенные *RandomizedMotifSearch*, немного менее консервативны, чем мотивы, возвращенные *MedianString*, прежний алгоритм имеет преимущество в том, что он может найти более длинные мотивы (поскольку *MedianString* становится слишком медленным для более длинных мотивов). Мотив длиной 20, возвращаемый *RandomizedMotifSearch* – CGGGACCTACGTCCCTAGCC (с результатом 57). Как показано ниже, консенсусные строки длиной 12, найденные *RandomizedMotifSearch* и *MedianString*, «внедрены» с небольшими вариациями в более длинный мотив длиной 20:

```
  GGACC**A**ACG**G**CC
CGGGACCTACGTCCCTAGCC
  GGAC**T**T**C**CG**G**CC
```

Наконец, в 2000 запусков с $N = 200$ *GibbsSampler* вернул ту же самую консенсусную строку длиной 20 для набора данных *DosR* как *RandomizedMotifSearch*, но создал другую коллекцию мотивов с меньшей оценкой 55.

Можно видеть, что различные алгоритмы поиска мотивов дают несколько различные результаты, и остается неясным, как найти все сайты связывания *DosR* в МТВ.

## Список литературы


[1] http://bioinformaticsinstitute.ru/teachers/vyahhi

[2] N.C. Jones, P.A. Pevzner. Introduction to Bioinformatics Algorithms. The MIT Press, Cambridge, MA 2004.

[3] P.A. Pevzner., P. Compeau. Bioinformatics Algorithms: An Active Learning Approach, Active Learning Publishers, 2014.

[4] J.D. Watson, F.H.C. Crick. Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid. Nature 171, 737—738 (1953).

[5] http://bioinformaticsalgorithms.com/excerpt/Pseudocode.pdf

[6] L.J. Guibas, A.M. Odlyzko. String overlaps, pattern matching, and nontransitive games. Journal of Combinatorial Theory, Series A. 30: 183—208 (1981).

[7] M.Meselson, F.W. Stahl. The Replication of DNA in Escherichia coli. Proc Natl Acad Sci USA. 44: 671—682 (1958).

[8] http://tubic.tju.edu.cn/Ori-Finder/

[9] X. Wang, C. Lesterlin, R. Reyes-Lamothe, G. Ball, D.J. Sherratt. Replication and segregation of an Escherichia coli chromosome with two replication origins. Proc Natl Acad Sci USA. 108: 243—250 (2011).

[10] X.Xia. DNA Replication and Strand Asymmetry in Prokaryotic and Mitochondrial Genomes. Current Genomics. 13: 16—27 (2012).

[11] M. Lundgren, A. Andersson, L. Chen, P. Nilsson, Rolf Bernander. Three replication origins in Sulfolobus species: Synchronous initiation of chromosome replication and asynchronous termination. Proc Natl Acad Sci USA. 101: 7046—7051 (2004).

[12] J. Rosenblatt, P.D. Seymour. The Structure of Homometric Sets. SIAM. J. on Algebraic and Discrete Methods, 3(3), 343–350 (1982).

[13] N. Venkataraman, A.L. Cole, P. Ruchala, A.J. Waring, R.I. Lehrer, O. Stuchlik, J. Pohl, A.M. Cole. Reawakening Retrocyclins: Ancestral Human Defensins Active Against HIV-1. PLoS Biol 7(4): e1000095 (2009).

Игорь Ильясович Юсипов
Алёна Игоревна Калякулина
Михаил Васильевич Иванченко

**ВВЕДЕНИЕ В АЛГОРИТМЫ БИОИНФОРМАТИКИ**

**Учебное пособие (лекционный материал)**

Федеральное государственное автономное образовательное учреждение
высшего образования “Нижегородский государственный
университет им. Н.И. Лобачевского”.
603950, Нижний Новгород, пр. Гагарина, 23.