Краткий конспект

Лекция 1. Поиск мотивов

версия 0.1(незавершенная)

Д. Ищенко* Б. Коварский* И. Алтухов* Д. Алексеев*

11 февраля, 2016

*МФТИ

1 Зачем искать мотивы?

2 Несколько слов о сложности алгоритмов

При разработке алгоритма важно представлять и оценивать кол-во времени, необходимое для его исполнения при конкретных входных данных, а также объем компьютерных ресурсов, задействованных при исполнении алгоритма. Когда мы говорим про «конкретные данные», мы подразумеваем определенную величину, например, размер памяти, выделяемой под входные данные в битах. Будем называть эту величину n. Для простоты будем считать, что время необходимое для исполнения алгоритма *пропорционально* кол-ву элементарных операций (которые в свою очередь определены архитектурой процессора). Будем считать элементарными операциями: сложение, вычитание, умножение, деление, вычисление корня, а также сравнение двух величин. Тогда оценка времени сводится к определению $f(n)$, функции количества элементарных операций от размера входных данных. Нас не будет интересовать точное значение $f(n)$ (это и не всегда возможно определить), а только лишь его оценка (чаще всего оценка сверху). Определяется она с помощью термина «O большое» для асимптотического поведения функций. $f(n) = O(g(n))$ означает, что кол-во операций $f(n)$ при увеличении n будет возрастать не быстрее, чем $g(n)$, умноженная на некоторую константу.

$$\exists (C > 0), n_0 : \forall (n > n_0)\ f(n) \leq Cg(n)$$

Например, сложность алгоритма который вычисляет простую сумму k входных чисел a_k оценивается, как $O(k)$. Грубо говоря, мы выполняем $(k-1)$ операций сложения. Таким образом кол-во операций $f(n)$ будет возрастать пропорционально кол-ву входных чисел k. Объем входных данных в битах можем оценить, как некоторую константу (например кол-во бит, зарезервированных под одно входящее число) умноженную на их кол-во $n = c \cdot k$, тогда $f(n) = f(c \cdot k) = k - 1 = O(k)$. Рассмотрим другой пример, алгоритм находящий ... алгоритм $O(n^2)$ ...

Аналогичные рассуждения в оценке применимы и к вычислению необходимой памяти (чаще оперативной) выделяемой при исполнении алгоритма. Оценивается кол-во выделяемых бит $m(n)$ от размера входных данных и оценивается с помощью $O(p(n))$. Возвращаясь к алгоритму вычисления суммы k чисел a_k. Допустим, алгоритм построен следующим образом: (i) прочесть все k чисел и записать в массив, (ii) просуммировать все a_k и результат записать в s, (iii) выдать результат s. Задействованная оперативная память – величина $m(n) = m(k \cdot c) = k \cdot c + c = c(k+1) = O(k)$ (нам необходимо записать в массив все k чисел, каждое из которых занимает c бит, а также выделить память для переменной s размера c бит). Если же изменить алгоритм следующим образом: (i) объявляется переменная $s = 0$ для хранения суммы, (ii) поочередно читается одно число из a_k и добавляется к s, $s = s + a_k$, после чего a_k удаляется из памяти (iii) выводится s. То в такой реализации нам необходимо хранить всего два значения s и текущее a_k, а значит всего $2c$ бит. Другими словами, кол-во необходимой памяти *не зависит* от размера входных данных (кол-ва входных чисел), такой вариант оценивается, как $m(n) = 2c = O(1)$. При этом в обоих реализациях алгоритма оценка времени одинакова $f_1(n) = f_2(n) = O(k)$.

Идеальным случаем ... написать про $O(n)$ ...

3 Простой подход к поиску мотива

Вернемся к задаче о поиске мотива. Есть строка (геном) S и паттерн (мотив) M:

```
 S :  TATGCATGCATGA
 M :  ATGCTGA
```

Необходимо определить все позиции вхождения M в S. Рассмотрим самый простой алгоритм, заключающийся в полном переборе всех позиций в S, подстановки в них M и проверки попарных совпадений символов (нуклеотидов).

```
 S :  TATGCATGCATGA
M1 :  ATGCATGA
      *
M2     ATGCATGA
       +++++++*
M3      ATGCATGA
        *
M4       ATGCATGA
         *
M5        ATGCATGA
          *
M6         ATGCATGA
           ++++++++
```

Обозначаем символом «*» проверку на совпадение, которая вернула значения $FALSE$ (символы отличаются), а символом «+» проверку, вернувшую $TRUE$ (символы совпадают). Очевидно, что встречая несовпадение символов в M и S, нет необходимости сравнивать оставшиеся символы, и мы сдвигаем мотив M на «+1» позицию относительно строки S и опять начинаем проверку с первого символа. В случае, если все символы совпали, считаем, что определили вхождение мотива M в строку S. Нетрудно показать, что при длине n строки S и длине m строки M, в худшем случае нам необходимо провести порядка $n \cdot m$ сравнений, оценка временной сложности алгоритма $f(n) = O(n \cdot m)$.

4 Усовершенствование простого подхода

Попробуем усовершенствовать подход, используя некоторые наблюдения. Например, мы знаем, что паттерн M начинается с символа «A». Будем запоминать при сравнении M и S, позиции, в которых в S встречается «A» и при очередном сдвиге M относительно S, будем производить его не на «+1» позицию, а сразу на позицию, в которой в S стоит символ «A». Такой подход изображен ниже.

```
 S :  TATGCATGCATGA
M1 :  ATGCATGA
      *
M2     ATGCATGA
       +++++++*
M3         ATGCATGA
           ++++++++
```

Очевидно, что мы уменьшили кол-во операций сравнения. Поступим еще «умнее», при втором сравнении (M2) мотива со строкой S, «запомним», что в S после символа «A» (в шестой позиции) стоят нуклеотиды «T» и «G», как раз те, с которых начинается мотив M. Поэтому произведя сдвиг M до шестой позиции, не будем повторно их сравнивать, а начнем сравнение сразу с четвертого символа. Подход изображен ниже:

```
S  :  TATGCATGCATGA
M1 :  ATGCATGA
      *
M2     ATGCATGA
       +++++++*
M3         ATGCATGA
              +++++
```

Мы добились дополнительного уменьшения кол-ва операций сравнения, но это был частный пример и рассуждали мы в очень «свободной» форме. Пока непонятно, как формализовать термины «запомнить», «заметить» и т.д. Возникает необходимость описать подход в виде алгоритма.

5 Z-алгоритм

Обобщение вышеописанных наблюдений формализуется в виде Z-алгоритма. Перед непосредственным разбором алгоритма, рассмотрим понятие «предобработки» («предпроцессинга») строки. Этим термином назовем проведение каких-либо операций со строкой еще до выполнения самого алгоритма поиска мотива. Причем, можно проводить предобработку как самой строки S (мы коснемся этого позже при разборе суффиксных деревьев), так и строки содержащей мотив M. Наш подход будет смесью этих двух вариантов (о чем будет сказано ниже) и позволит решить задачу поиска мотива за линейное время $O(n+m)$, где n – длина строки (генома) S, m – длина мотива M.

Введем несколько обозначений и определений:

(i) Обозначим $S[k..m]$ подстроку из S, начинающуюся с k-го и заканчивающуюся m-м символом. $S[k]$ – просто k-й символ строки.

(ii) **k-й префикс** строки S: подстрока длины k, начинающаяся с первого символа $S[1..k]$.

(iii) Для строки S и позиции $i \geq 2$, $Z_i(S)$ – длина максимальной подстроки S, начинающейся с позиции i и совпадающей с префиксом S той же длины.

Рассмотрим строку S и укажем различные значения $Z_i(S)$:

```
S  :  ATGCATGCATGA
       |  |   | |
       |  |   | Z11 = 0
       Z2 = 0 |
          |   |
          |   Z9 = 3 [ATG]
          Z5 = 7 [ATGCATG]
```

Для лучшего понимания величины $Z_i(S)$ введем понятие Z-ящика (Z-box), каждый «ящик» начинается в некоторой позиции $i \geq 2$, в которой $Z_i > 0$, длина ящика соответствует значению Z_i.

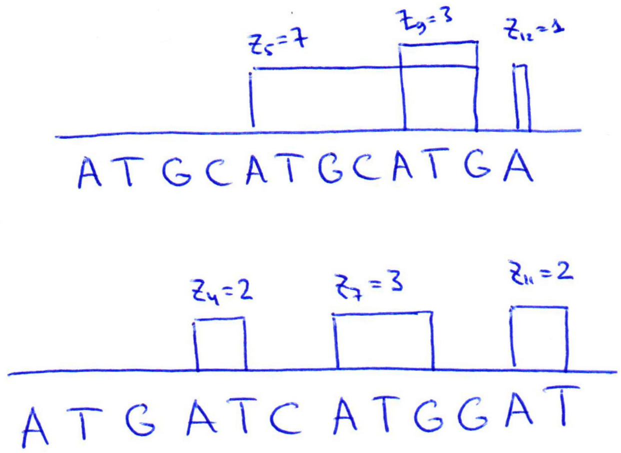

Рис. 1: Визуальное изображение значений Z_i в виде Z-ящиков. Длина ящика соответствует значению Z_i, высота ящика смысла не имеет, и изменяется *только для удобства визуализации*.

Введем еще две дополнительные величины, их осмысление потребует некоторой *внимательности и усердия*:

(iv) Для любого $i \geq 2$, r_i – координата самого правого символа во всех Z-ящиках, которые начинаются в позициях $\leq i$

(v) Для любого $i \geq 2$, l_i – это позиция, с которой начинается Z-ящик, которому соответствует r_i. В случае, если Z-ящиков заканчивающихся на r_i несколько, выбирается наименьшая позиция.

Разберем несколько примеров определения r_i и l_i на рисунке:

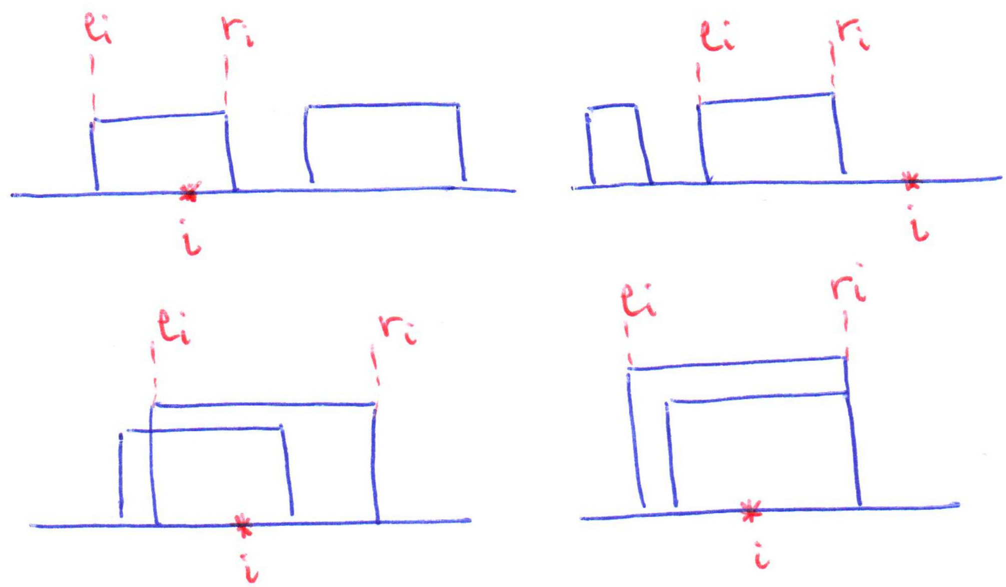

Рис. 2: Примеры определения r_i и l_i для заданного i.

Покажем, что вычисление всех значений Z_i (что эквивалентно построению всех Z-ящиков) возможно за линейное время, т.е. $f(n) = O(n)$, где n – длина строки S. Сначала разберем частный случай, а затем перейдем к общему алгоритму.

Для того, чтобы вычислить Z_2 достаточно сравнивать $S[k]$ с $S[k-1]$, начиная с $k = 2$, пока не дойдем до первого несовпадения. Автоматическим мы определяем r_2 и l_2, $r_2 = Z_2 + 1$, $l_2 = 2$. Если не совпадают даже $S[2] \neq S[1]$, то $Z_2 = r_2 = l_2 = 0$.

Теперь предположим, что мы находимся на $(k + 1 = 101)$-й позиции $(k = 100)$, все значения $Z_2..Z_{100}$ уже известны и $r_k = r_{100} = 110$, $l_k = l_{100} = 80$ (значит есть «ящик» $Z_{80} = 110 - 80 + 1 = 31$) (Рис. 3). Стоит задача вычислить Z_{101}.

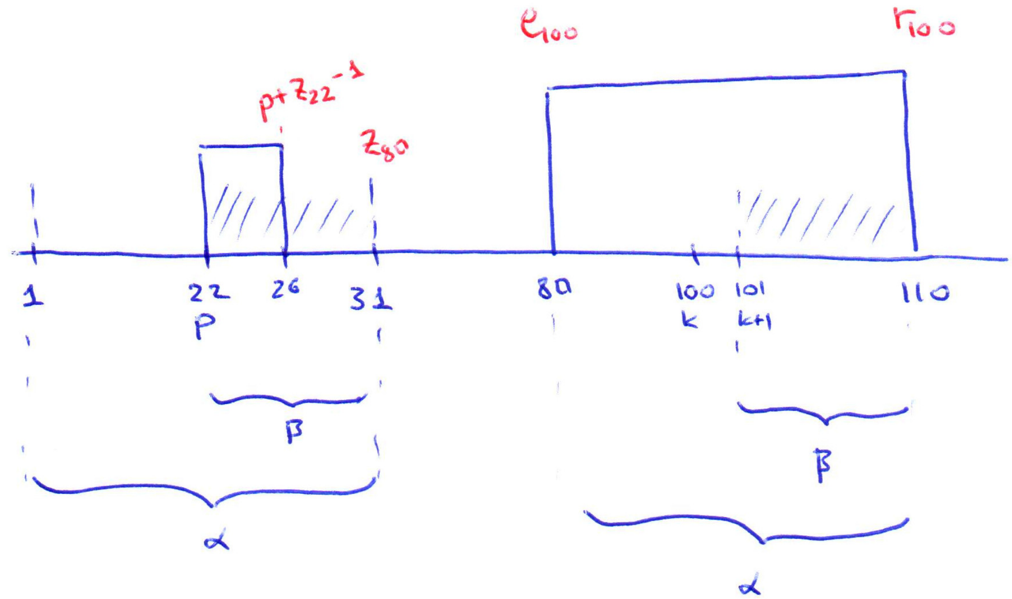

Рис. 3: Определение Z_{101} на основании предыдущих значений $\{Z_i\}$, r_{100}, l_{100}.

Назовем последовательность $S[80..110] = \alpha$, тогда такая же последовательность α находится в начале строки $S[1..31]$ (по определению Z-ящика). Очевидно, что последовательность, начинающаяся с $k = 101$, $S[101..110]$, совпадает с $S[22..31]$ (одинаковые подпоследовательности α), назовем их β. Для вычисления Z_{101} достаточно посмотреть на значение Z_{22} (ящика, начинающегося с 22 позиции). Пусть $Z_{22} = 5$, это значит, что начиная с символа $S[22]$ только пять символов совпадают с префиксом ($S[1..5]$), т.е. мы автоматически вычисляем $Z_{101} = Z_{22} = 5$ без дополнительных сравнений символов (т.к. Z-ящик, начинающийся с 22-ой позиции, короче строки β, а строка начинающаяся со 101 позиции равна β.

Мы рассмотрели важный частный случай, перейдем к формальному описанию алгоритма вычисления Z_{k+1} элемента, при вычисленных $\{Z_2, Z_3, ..., Z_k\}$, r_k, l_k:

(i) Если $k + 1 > r_k$ (ни один Z-ящик, не покрывает $(k + 1)$-й символ), Z_{k+1} вычисляется последовательным сравнением символов $S[k + 1]$ с $S[1]$, $S[k + 2]$ с $S[2]$ и т.д. до первого несовпадения. Z_{k+1} будет равно кол-ву совпавших элементов, если $Z_{k+1} > 0$, то $r_{k+1} = k + Z_{k+1}$, $l_{k+1} = k + 1$.

(ii) Если $k+1 \le r_k$ (($k+1$)-й символ содержится в Z-ящике), подстрока $S[l_k..r_k]$ (назовем её α) совпадает с префиксом S. Тогда S_{k+1} совпадает с p-м символом, где $p = k+1-l_k+1$. Строка $S[(k+1)..r_k]$ (назовем ее β) совпадает со строкой $S[p..Z_{l_k}]$. Это означает, что строка начинающаяся с позиции $(k+1)$ совпадает с префиксом хотя бы на Z_p символов или длину строки β (обозначим её $|\beta|$). Рассмотрим два случая:

   (a) $Z_p < |\beta|$: тогда $Z_{k+1} = Z_p$, $r_{k+1} = r_k$, $l_{k+1} = l_k$.

   (b) $Z_p \ge |\beta|$: тогда $S[(k+1)..r_k]$ является префиксом S, и $Z_{k+1} \ge |\beta| = r_k - k$. Но Z_k может быть больше $|\beta|$, поэтому начинаем сравнение $S[r_k+1]$ с символами $S[|\beta|+1]$, $S[r_k+2]$ с символвами $S[|\beta|+2]$ и т.д. до первого несовпадения. Пусть несовпадение возникло в позиции $q \ge r_k+1$, тогда $Z_{k+1} = q-(k+1)$, $r_{k+1} = q-1$ и $l_{k+1} = k+1$.

Такой алгоритм найдет все значения Z_k за линейное время $O(n)$, т.к. как каждый символ мы сравниваем только один раз с другим символом строки). А теперь вернемся к нашей задаче поиска мотива M в последовательности S. Объединим строки M и S в одну, вставив между ними символ, который не встречается ни в M ни в S (если мы работаем с нуклеотидными последовательностями, то можно выбрать любой символ, отличный от A, T, G, C), например, «\$»:

```
M$S
```

А теперь применим алгоритм вычисления Z-значений для такой последовательности. Очевидно, что позиции вхождения M в S будут теми позициями, в которых Z_k равно длине мотива, т.е. $Z_k = |M|$. Вот и все.

6 Куда двигаться дальше?

7 Ссылки

[1] Gusfield D. Algorithms on strings, trees and sequences: computer science and computational biology. – Cambridge university press, 1997.