Строки

Сапожников Денис

Contents

1 Префикс-функция

1.1 Определения

Определение (Префикс строки)**.** Префиксом строки называется начало строки; первые несколько подряд идущих символов.

Определение (Суффикс строки)**.** Суффиксом строки называется конец строки; последние несколько подряд идущих символов.

Определение (Бордер строки)**.** Бордером строки называется любой префикс строки длины k, который равен суффиксу строки той же длины.

Определение. Если префикс, суффикс или бордер не полностью совпадают со строкой, то их принято называть собственными.

Например, пусть есть строка "ababa". Она содержит:

- Префиксы: ε[1], a, ab, aba, abab, ababa
- Суффиксы: ε, a, ba, aba, baba, ababa
- Бордеры: ε, a, aba, ababa

Префикс, суффикс и бордер ababa будут не собственными, остальные – собственными.

Далее будем подразумевать, что рассматриваются собственные суффиксы/префиксы/бордеры, если не оговорено иное.

Определение. Пусть дана строка $s = s_0 s_1 s_2 \dots s_{n-1}$ длины n. Для каждой позиции необходимо посчитать π_i = длине максимального собственного бордера подстроки $s_{0..i}$. Массив π принято называть префикс-функцией [2].

Например, для строки ababaa это будет следующий массив: $\pi = [0, 0, 1, 2, 3, 1]$ [3].

1.2 Алгоритм в лоб

Алгоритм "в лоб" будет действительно "в лоб". Пройдемся по всем префиксам строки и простым перебором со сравнением подстрок будем находить максимальный бордер:

```
vector<int> pi_function(string s) {
  int n = s.size();
  vector<int> pi(n);
  for (int i = 1; i < n; ++i)
    for (int j = 0; j <= i; ++j)
      if (s.substr(0, j) == s.substr(i - j, j))
        pi[i] = j;
  return pi;
}
```

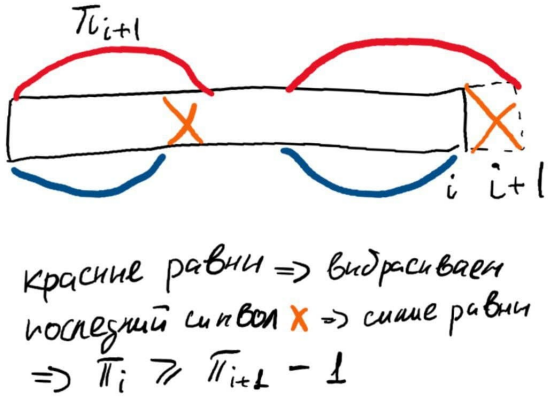
[1] В теории строк пустую строку принято обозначать за ε

[2] Почему-то только в России его обозначают за π, это началось с emaxx

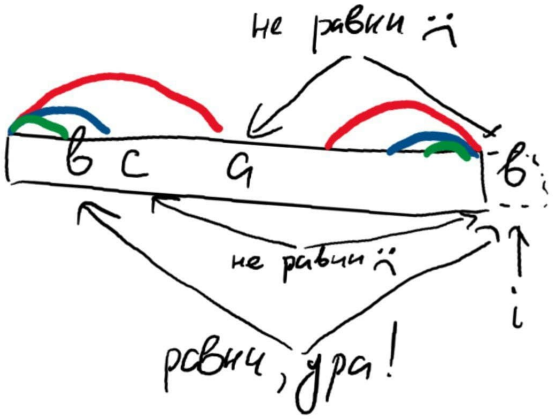
[3] Обратите внимание, мы всегда ищем собственный бордер, поэтому $\pi_0 = 0$

1.3 Идеи по ускорению

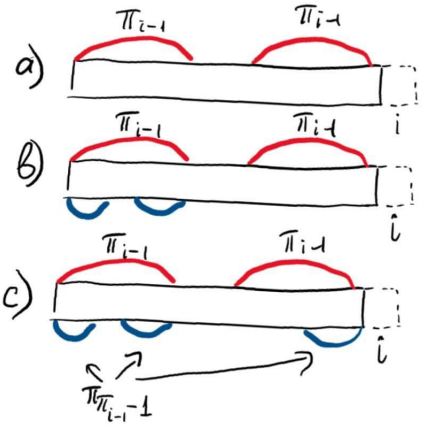
Идея 1: Заметим, что префикс функция при переходе от i к $i+1$ не может увеличиться больше, чем на 1. Действительно, предположим противное, тогда посмотрим на строку $s[0 \dots \pi_{i+1}]$. По определению префикс-функции, она равна $s[i+1-\pi_{i+1} \dots i+1]$. Но раз эти строки равны, то равны и их префиксы: $s[0 \dots \pi_{i+1}-1] = s[i+1-\pi_i \dots i]$. То есть мы нашли бордер для строки $s[0 \dots i]$ длины $\pi_{i+1}-1$. Противоречие.



Идея 2: Исходя из прошлой идеи, нам необходимо уметь просматривать все бордеры строки на префиксе длины $i-1$, чтобы пытаться их «продлить». А именно, если после бордера длины j идет буква s_i (то есть $s_j = s_i$), то мы «продлили бордер» до позиции i и нашли π_i[1].



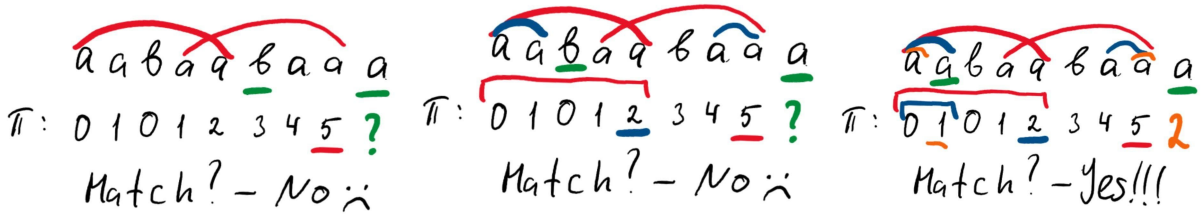
На примере выше зеленые строки равны и после них идет одна и та же буква, значит π_i будет равно длине зеленой строки + 1. Остается вопрос: а как перебирать все бордеры? Нам поможет следующая картинка:



[1] Доказательство аналогично доказательству из идеи 1

На картинке (a) мы видим, что две красные строки равны (как префикс-функция). Раз равны красные строки, то найдем максимальный бордер для левой красной строки (это будет синий бордер, (b)). Но раз красные строки равны, то такой же максимальный бордер будет и у правой красной строки (c). Осталось понять, что длину бордера для левой синей строки мы уже знаем, так как мы уже вычислили значения префикс-функции для той позиции.

Для лучшего понимания, посмотрите, как вычисляются значения префикс-функции на примере:



В массиве префикс-функции каждый раз подчёркивается длина следующего бордера.

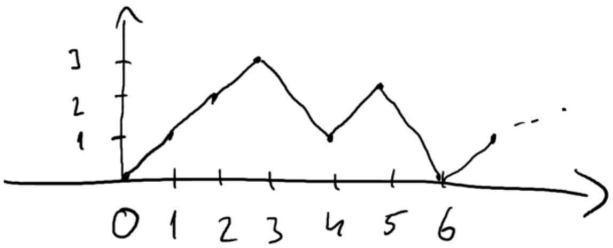
1.4 Алгоритм

Теперь мы готовы к написанию простого и элегантного кода: если мы рассматриваем текущий бордер длины j, то его (строки $s_{0..j}$) максимальный бордер лежит в позиции π_{j-1} (ещё раз обратитесь к картинке выше).

```
vector<int> pi_function(string s) {
  int n = s.size();
  vector<int> pi(n);
  for (int i = 1; i < n; ++i) {
    int j = pi[i - 1];
    while (j > 0 && s[j] != s[i])
      j = pi[j - 1];
    pi[i] = j + (s[i] == s[j]);
  }
}
```

1.5 Асимптотика алгоритма

Единственное не очевидное место – это while. Докажем, что его суммарное время работы – $O(n)$. Мне очень нравится графическое доказательство: нарисуем график какой-нибудь префикс-функции:



О любом таком графике можно сказать, что увеличение значения от i к $i+1$ не может происходить больше чем на 1. А суммарное время работы while – это сумма всех "падений". Но нельзя упасть ниже нуля. Это значит, что мы не сможем упасть суммарно на большее количество, чем можем подняться за всё время, то есть не больше чем на n. Значит суммарное время работы while не превосходит n. Значит префикс-функция вычисляется за $O(n)$.

1.6 Алгоритм Кнута-Морриса-Пратта

Задача. Дана строка s и текст t. Требуется найти количество вхождений строки s в текст t за $O(|s| + |t|)$.

Решение. Запишем строку $s\#t$ и вычислим на ней префикс-функцию, где $\#$ — символ-разделитель, который не встречается в s и t. Если на какой-то позиции i: $\pi_i = |s|$, то это означает, что текущий суффикс строки t равен строке s, а это то, что мы и хотели найти. Три человека трудилось в поте лица над этим алгоритмом и теперь у него есть название: **алгоритм Кнута-Морриса-Пратта**.

1.7 Задачи

Задача 1. Пусть у вас есть функция isSubstring, которая проверяет, является ли одна строка подстрокой другой строки. Вам нужно определить, является ли строка s_1 циклическим сдвигом строки s_2, используя только один вызов функции isSubstring.

Задача 2. Дана строка s. Для каждого префикса s необходимо найти количество его вхождений за $O(|s|)$.

Решение.
Идея 1: Рассмотрим все бордеры строки $s[0..i]$ для конкретного i. Раз мы рассматриваем все бордеры, то мы переберём все строки, которые равны префиксу s, и при этом оканчивающиеся в позиции i. Перебрав все i, мы найдем просто все подстроки, которые совпадают с некоторым префиксом s.

Пока мы получили алгоритм за квадрат, хочется быстрее.
Идея 2: Воспользуемся стандартной техникой с оффлайновым проходом. А именно, поставим изначально в каждую позицию в массиве dp по 1. Далее будем идти по убыванию i с сохранением полуинварианта: в $dp[i]$ лежит количество строк, которые совпадают с префиксом длины i.

Раз у нас есть $dp[i]$ строк, которые равны префиксу, то у каждой из них есть максимальный бордер размера π_{i-1}, который по определению тоже совпадает с префиксом. Тогда добавим в позцию π_{i-1} текущее количество префиксов длины i. То есть, $dp[\pi_{i-1}] += dp[i]$. Меньшие бордеры будут рассмотрены в позиции π_{i-1} аналогично.

Задача 3. Дана строка s. Необходимо найти наименьший период строки[1].

Решение. В первую очередь, нужно нарисовать несколько (штук 10) примеров и поискать закономерность, связанную с префикс-функцией. Закономерность будет вида $\pi_{|s|} + T = |s|$. Это позволяет найти период строки за $O(n)$. Докажем чуть более сильную теорему об этом.

Лемма. *Если у строки длины n есть бордер длины k, то у нее также имеется период длины $n - k$.*

Proof. Пусть дана строка α.

Напишем формально определение бордера длины k строки α:

$\forall i = 1 \dots k : \alpha[i] = \alpha[i + (n - k)]$ Сделаем замену $x = n - k$:

$\forall i = 1 \dots n - x : \alpha[u] = \alpha[i + x]$ Получили определение периода длины x. Но $x = n - k$, значит у строки α есть период длины $n - k$.

Есть и более красивое доказательство, которое мы разобрали на паре; для него нужна красивая картинка, но не в этот раз. □

Задача 4. Придумайте линейный по сложности алгоритм для вычисления количества различных циклических сдвигов строки s.

[1] Периодом строки s называется такое наименьшее число T, что $\forall i : s_i = s_{i+T}$

Задача 5. Даны строки p и s. s состоит только из маленьких букв английского алфавита, в то время как p может содержать ещё и символы '*'. Требуется за время $O(|p|+|s|)$ определить, можно ли подставить вместо символов '$*$' какие-то строки (возможно, различные), так чтобы строка p стала подстрокой s.

Задача 6. Секретная задача для тех, кто читает конспект.

Есть ещё одна интересная теорема, которая очень часто применяется в теории строк, но не в этой лекции.

Теорема (Fine and Wilf's theorem)**.** *Если у строки w есть периоды p и q, где $|w| \geq p + q - gcd(p, q)$, то $gcd(p, q)$ также является периодом этой строки.*

Зачем она нужна? Ну с помощью неё можно умудриться считать префикс-функцию в онлайне с добавлением и удалением символов за $O(\log n)$. Увы, но я даже не справляюсь найти теорию по этому алгоритму на любом языке, так что пруфов не будет.

2 Z-функция

2.1 Определение

Пусть дана строка $s = s_0 s_1 s_2 \dots s_{n-1}$. Для каждой позиции необходимо вычислить z_i – наибольший общий префикс строки и её i-го суффикса.

Приведём несколько примеров:

- $s =$ aaaaa, $z = [0, 4, 3, 2, 1]$
- $s =$ aaabaab, $z = [0, 2, 1, 0, 2, 1, 0]$

2.2 Алгоритм в лоб

Мы просто для каждой позиции i перебираем ответ для неё z_i, начиная с нуля, и до тех пор, пока мы не обнаружим несовпадение или не дойдём до конца строки.

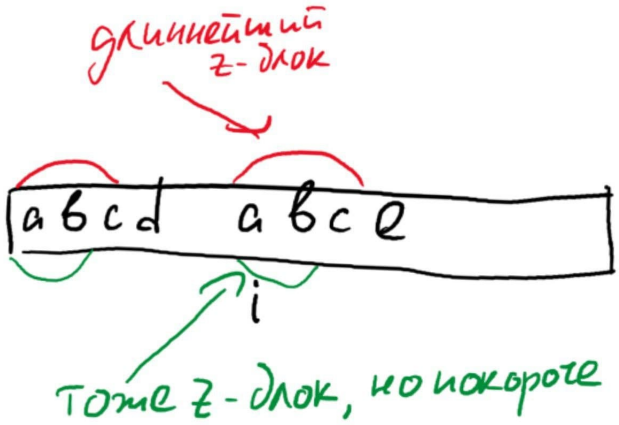
```
vector<int> z_function_trivial (string s) {
  int n = s.size();
  vector<int> z (n);
  for (int i = 1; i < n; ++i)
  while (i + z[i] < n && s[z[i]] == s[i+z[i]])
    ++z[i];
  return z;
}
```

Разумеется, эта реализация слишком неэффективна, перейдём теперь к построению эффективного алгоритма.

2.3 Ускоряем

Чтобы получить эффективный алгоритм, будем вычислять значения $z[i]$ по очереди: от $i = 1$ до $n - 1$, и при этом постараемся при вычислении очередного значения $z[i]$ максимально использовать уже вычисленные значения.

Назовём подстроку, совпадающую с префиксом строки s, z-блоком. Например, значение искомой Z-функции $z[i]$ – это длиннейший z-блок, начинающийся в позиции i (и заканчиваться он будет в позиции $i + z[i] - 1$).



Для этого будем поддерживать координаты $[l; r]$ самого правого z-блока, т.е. из всех обнаруженных отрезков будем хранить тот, который оканчивается правее всего. В некотором

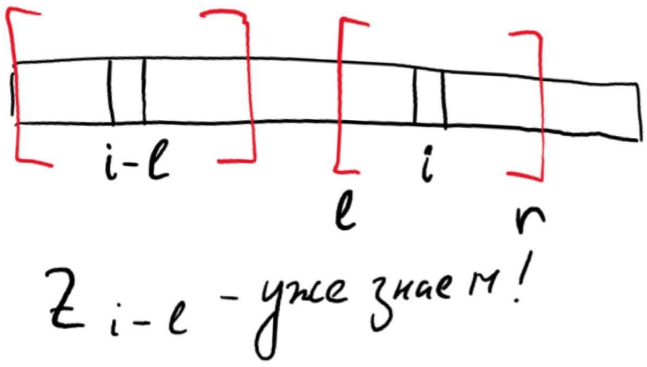
смысле, индекс r – это такая граница, до которой наша строка уже была просканирована алгоритмом, а всё остальное – пока ещё не известно [1].

Тогда если текущий индекс, для которого мы хотим посчитать очередное значение Z-функции, – это i, мы имеем один из двух вариантов:

- $i > r$ – т.е. текущая позиция лежит за пределами того, что мы уже успели обработать. Тогда будем искать $z[i]$ тривиальным алгоритмом, т.е. просто пробуя значения $z[i] = 0$, $z[i] = 1$, и т.д. Заметим, что в итоге, если $z[i]$ окажется > 0, то мы будем обязаны обновить координаты самого правого z-блока $[l; r]$ — т.к. $i + z[i] - 1$ гарантированно окажется больше r.

- $i \le r$ – т.е. текущая позиция лежит внутри отрезка совпадения $[l; r]$. Тогда мы можем использовать уже подсчитанные предыдущие значения Z-функции, чтобы проинициализировать значение $z[i]$ не нулём, а каким-то возможно бОльшим числом.

  Для этого заметим, что подстроки $s[l \dots r]$ и $s[0 \dots r - l]$ совпадают. Это означает, что в качестве начального приближения для $z[i]$ можно взять соответствующее ему значение из отрезка $s[0 \dots r - l]$, а именно, значение $z[i - l]$.

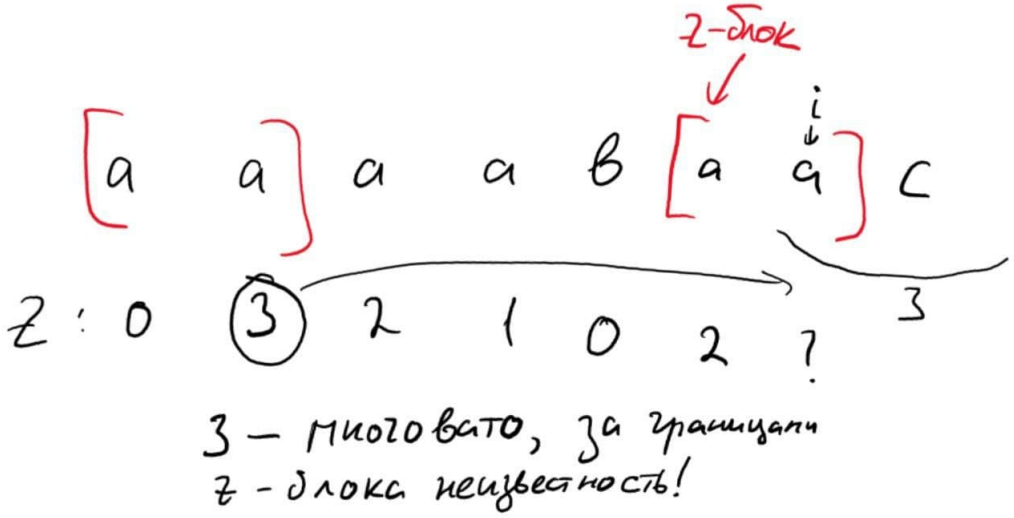


  Однако значение $z[i - l]$ могло оказаться слишком большим: таким, что при применении его к позиции i оно "вылезет" за пределы границы r. Этого допустить нельзя, т.к. про символы правее r мы ничего не знаем, и они могут отличаться от требуемых.

  Приведём пример такой ситуации, на примере строки: aaaabaac.

  Когда мы дойдём до предпоследней позиции $i = 6$, текущим самым правым отрезком будет $[5; 6]$. Позиции 6 с учётом этого отрезка будет соответствовать позиция $6 - 5 = 1$, ответ в которой равен $z[1] = 3$. Очевидно, что таким значением инициализировать $z[6]$ нельзя, оно совершенно некорректно. Максимум, каким значением мы могли инициализировать – это 1, поскольку это наибольшее значение, которое не вылазит за пределы отрезка $[l; r]$.

[1] Это важное замечание, оно понадобится нам ниже в случае $i \le r$



Таким образом, в качестве начального приближения для $z[i]$ безопасно брать только такое выражение:

$$\widetilde{z}[i] = \min(r - i + 1, z[i - l])$$

Проинициализировав $z[i]$ таким значением $\widetilde{z}[i]$, мы снова дальше действуем тривиальным алгоритмом – потому что после границы r, вообще говоря, могло обнаружиться продолжение z-блока, предугадать которое одними лишь предыдущими значениями Z-функции мы не могли.

Таким образом, весь алгоритм представляет из себя два случая, которые фактически различаются только начальным значением $z[i]$: в первом случае оно полагается равным нулю, а во втором – определяется по предыдущим значениям по указанной формуле. После этого обе ветки алгоритма сводятся к выполнению тривиального алгоритма, стартующего сразу с указанного начального значения.

Алгоритм получился весьма простым. Несмотря на то, что при каждом i в нём так или иначе выполняется тривиальный алгоритм – мы достигли существенного прогресса, получив алгоритм, работающий за линейное время. Почему это так, рассмотрим ниже, после того, как приведём реализацию алгоритма.

2.4 Реализация

```
vector<int> z_function(string s) {
  int n = s.size();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r)
      z[i] = min(r - i + 1, z[i - l]);
    while (i+z[i] < n && s[z[i]] == s[i+z[i]])
      ++z[i];
    if (i + z[i] - 1 > r)
      l = i, r = i + z[i] - 1;
  }
  return z;
}
```

Прокомментируем эту реализацию.

Всё решение оформлено в виде функции, которая по строке возвращает массив длины n – вычисленную Z-функцию. Массив $z[]$ изначально заполняется нулями. Текущий самый правый z-блок полагается равным $[0;0]$, т.е. заведомо маленький отрезок, в который не попадёт ни одно i.

Внутри цикла по $i = 1 \dots n-1$ мы сначала по описанному выше алгоритму определяем начальное значение $z[i]$ – оно либо останется нулём, либо вычислится на основе приведённой формулы. После этого выполняется тривиальный алгоритм, который пытается увеличить значение $z[i]$ настолько, насколько это возможно.

В конце выполняется обновление текущего самого правого z-блока $[l;r]$, если, конечно, это обновление требуется — т.е. если $i + z[i] - 1 > r$.

2.5 Асимптотика алгоритма

Докажем, что приведённый выше алгоритм работает за линейное относительно длины строки время, т.е. за $O(n)$.

Нас интересует вложенный цикл while, т.к. всё остальное – лишь константные операции, выполняемые $O(n)$ раз. Покажем, что каждая итерация этого цикла while приведёт к увеличению правой границы r на единицу. Для этого рассмотрим обе ветки алгоритма:

- $i > r$

  В этом случае либо цикл while не сделает ни одной итерации (если $s[0] \neq s[i]$), либо же сделает несколько итераций, продвигаясь каждый раз на один символ вправо, начиная с позиции i, а после этого – правая граница r обязательно обновится.

  Поскольку $i > r$, то мы получаем, что действительно каждая итерация этого цикла увеличивает новое значение r на единицу.

- $i \leq r$

  В этом случае мы по приведённой формуле инициализируем значение $z[i]$ некоторым числом $\widetilde{z}_i$. Сравним это начальное значение $\widetilde{z}_i$ с величиной $r - i + 1$, получаем три варианта:

  1. $\widetilde{z}_i < r - i + 1$

     Докажем, что в этом случае ни одной итерации цикл while не сделает.

     Это легко доказать, например, от противного: если бы цикл while сделал хотя бы одну итерацию, это бы означало, что определённое нами значение $\widetilde{z}_i$ было неточным, меньше настоящей длины z-блока. Но т.к. строки $s[l \dots r]$ и $s[0 \dots r-l]$ совпадают, то это означает, что в позиции $z[i-l]$ стоит неправильное значение: меньше, чем должно быть.

     Таким образом, в этом варианте из корректности значения $z[i-l]$ и из того, что оно меньше $r - i + 1$, следует, что это значение совпадает с искомым значением $z[i]$.

  2. $\widetilde{z}_i = r - i + 1$

     В этом случае цикл while может совершить несколько итераций, однако каждая из них будет приводить к увеличению нового значения r на единицу: потому что первым же сравниваемым символом будет $s[r+1]$, который вылазит за пределы отрезка $[l;r]$.

  3. $\widetilde{z}_i > r - i + 1$

     Этот вариант принципиально невозможен, в силу определения $\widetilde{z}_i = \min(r-i+1, \dots)$.

Таким образом, мы доказали, что каждая итерация вложенного цикла приводит к продвижению указателя r вправо. Т.к. r не могло оказаться больше $n-1$, это означает, что всего этот цикл сделает не более $n-1$ итерации.

Поскольку вся остальная часть алгоритма, очевидно, работает за $O(n)$, то мы доказали, что и весь алгоритм вычисления Z-функции выполняется за линейное время.

3 Алгоритм Манакера

Этот алгоритм настолько бесполезен, на сколько это только возможно, ~~но меня держат в заложниках и заставляют его рассказывать~~ но будем считать его простым упражнением на строки.

Алгоритм решает следующую задачу: посчитать количество всех палиндромов в строке. Таких может быть $O(n^2)$, что, конечно же, много и в лоб из-за этого задачу решить не получится. Вместо этого для каждой позиции мы насчитаем длину максимального четного и нечетного палиндрома с центром в ней. Разберем случай с нечетными палиндромами, а четные к ним легко сведутся. Длинц нечетных палиндромов с центрами в позиции i обозначим за d_i.

Мы будем действовать один-в-один алгоритму z-функции. А именно, будем поддерживать границы самого правого известного нечетного палиндрома l и r, а текущая позиция, в которой хотим посчитать длину палиндрома — i. Далее ровно такие же случаи:

- $i > r$ – т.е. текущая позиция лежит за пределами того, что мы уже успели обработать. Тогда будем искать $d[i]$ тривиальным алгоритмом, т.е. просто пробуя значения $d[i] = 0$, $d[i] = 1$, и т.д. Заметим, что в итоге, если $d[i]$ окажется > 0, то мы будем обязаны обновить координаты самого правого палиндрома $[l; r]$ — т.к. $i + d[i] - 1$ гарантированно окажется больше r.

- $i \le r$ – т.е. текущая позиция лежит внутри известного палиндрома $[l; r]$. Тогда мы можем использовать уже подсчитанные предыдущие значения d, чтобы проинициализировать значение $d[i]$ не нулём, а каким-то возможно бОльшим числом.

  В качестве начального приближения $\widetilde{d}[i]$ можно брать $d[l + (r - i)]$ (из-за того, что палиндром симметричен). Но аналогично z-функции, мы можем случайно выйти за границы текущего самого правого палиндрома. Поэтому определим приближение по следующей формуле:

  $$\widetilde{d}[i] = \min(d[l + (r - i)], r - i + 1)$$

  Проинициализировав $d[i]$ таким значением $\widetilde{d}[i]$, мы снова дальше действуем тривиальным алгоритмом – потому что после границы r, вообще говоря, могло обнаружиться продолжение палиндрома, предугадать которое одними лишь предыдущими значениями Z-функции мы не могли.

Аналогично z-функции, этот алгоритм работает за линейное время.

Для четного случая, дабы не страдать с индексами применим трюк: вставим сивмол # между каждой парой символов строки. Тогда четные палиндромы в исходной строке — это нечетные палиндромы с центрами в #.

```cpp
vector<int> manacher_odd(string s) {
  int n = (int) s.size();
  vector<int> d(n, 1);
  int l = 0, r = 0;
  for (int i = 1; i < n; i++) {
    if (i < r)
      d[i] = min(r - i + 1, d[l + r - i]);
    while (i - d[i] >= 0 && i + d[i] < n && s[i - d[i]] == s[i + d[i]])
      d[i]++;
    if (i + d[i] - 1 > r)
      l = i - d[i] + 1, r = i + d[i] - 1;
  }
  return d;
}

long long manacher(string s) {
  string t = "#";
  for (int i = 0; i < s.size(); ++i) {
    t += s[i] + "#";
  }
  auto d = manacher_odd(t);
  long long sum = 0;
  for (int x : d)
    sum += x;
  return sum;
}
```

4 Хеши

Большинство задач на строки сильно бы упростились, если бы можно было проверять на равенство две строки за $O(1)$, а не за их длину. Хеши как раз предлагают этот метод.

4.1 Полиномиальный хеш

4.1.1 Определение

Определение. Полиномиальным хешом строки $s = s_0 s_1 s_2 \dots s_{n-1}$ называется сумма

$$hash(s) = (s_0 + s_1 \cdot Q + s_2 \cdot Q^2 + \dots + s_{n-1} Q^{n-1}) \% P$$

для некоторых чисел P и Q, называемых **модулем хеша** и **основанием хеша** соответственно.

Зачем нам это нужно? Теперь мы будем говорить, что если $hash(s) = hash(t)$, то строки s и t равны[1].

Возникает резонный вопрос: Строк много, а чисел мало, как быть? А именно, если число P порядка 10^9 (а именно такое число обычно и будет использоваться), то мы отображаем набор из экспоненты строк (строк длины 100 из 26 символов 26^{100}) в числа из интервала $[0; P-1]$. Это значит, что разным строкам будут сопоставлены одни и те же числа.

Не надо паники, это нормально. Ситуация, когда две строки имеют одинаковый хеш называется **коллизия**. Идея в том, что коллизии бывают не так часто, если правильно всё сделать.

Утверждение (Бездоказательно)**.** *Отображение строк в их полиномиальные хеши преобразует строку в случайное число*[2]*. А про случайные числа есть интересное наблюдение, именуемое парадоксом дней рождений.*

4.1.2 Парадокс дней рождений

Если в классе 23 ученика или более, то более вероятно то, что у какой-то пары одноклассников дни рождения придутся на один день, чем то, что у каждого будет свой неповторимый день рождения (то есть верятность совпадения дней рождений в классе из 23 и более людей превышает 50%).

Эту вероятность вычислить совсем не сложно.

- Если в классе один ученик, то вероятность того, что все ДР в разные дни равна 1.
- Если в классе есть два ученика, то вероятность того, что ДР 2го ученика попадет на ДР первого равна $\frac{1}{365}$. То есть вероятность того, что у всех ДР в разные дни равна $1 - \frac{1}{365}$
- Если в классе есть три ученика, то вероятность того, что ДР третьего ученика не попадет на ДР первых двух равна $1 - \frac{2}{365}$ (если их ДР в разные дни). Таким образом, вероятность того, что у трех учеников ДР в разные дни равна $\left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right)$
- И т.д. получаем формулу вида

$$\left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right)\left(1 - \frac{3}{365}\right) \times \dots \times \left(1 - \frac{23}{365}\right) < 0.5$$

[1] с большой вероятностью

[2] при наличии некоторых дополнительных условий, о которых в разделе о выборе P и Q

Если бы в году было n дней, то потребовалось бы примерно $\sqrt{n}$ чисел. Эмпирически это означает, что если вы делаете k сравнений строк, то P должно быть порядка k^2.

4.1.3 О выборе P и Q

1. Чтобы отображение было случайным, необходимо, чтобы P было простым числом

2. Чтобы не было «глупых» коллизий, необходимо следующие условия:

   (a) $s_i < Q$
   Если $Q = 2$, а $s = 012, t = 221$, то $hash(s) = hash(t) = 10$.

   (b) $s_i > 0$
   Без этого хеши для строк 000 и 00 будут равны. То есть если вы делаете $c \to c -$ 'a', то вы как раз отображаете букву 'a' в ноль, не надо так!

3. Чтобы было мало коллизий необходимо брать большое P, но чтобы вычисления помещались в long long, необходимо чтобы $P \sim 10^9$. Таким образом, хорошими простыми числами являются $10^9+7, 10^9+9$. Рекомендую найти своё простое число в районе 10^9 и использовать его на олимпиадах.

4. Если сравнений подстрок больше, чем 10^5 (например, $10^5 \log 10^5$), и у вас есть есть подозрение на коллизии (например, при замене Q/P случаются разные WA), то для каждой строки храните не один хеш, а два (суммы разным модулям).

   Пример, где хеш по двойному модулю необходим – это если у вас есть множество из n строк и вы хотите проверить, есть ли в нём пара равных. Тогда вы засунете хеши всех строк в set и проверите их на равенство, **но неявно вы сравните каждую строку с каждой**, то есть у вас будет n^2 сравнений на равенство.

4.1.4 Хеши подстрок

Помимо сравнения строк, можно сравнивать и подстроки. Для этого запомним не только хеш строки, но и хеш всех ее префиксов. Тогда
$hash(s[0 \dots r]) - hash(s[0 \dots l-1]) = Q^{l-1}s_l + Q^{l+1}s_{l+1} + \dots + Q^{r-1}s_r =$
$= Q^{l-1}(s_l + Qs_{l+1} + \dots + s_r Q^{r-l-1}) = Q^{l-1} \cdot hash(s[l \dots r])$
То есть мы можем получить хеш произвольной подстроки, домноженной на Q^{l-1}. Не забудьте, что все вычисления происходят по простому модулю, значит разница $hash(s[0 \dots r]) - hash(s[0 \dots l-1])$ **может быть отрицательной**, так что не забудьте в таком случае добавить P. Далее, если мы хотим сравнить две подстроки на равенство, то есть два способа:

1. Сложный, но удобный. Можно делить число $Q^{l-1} \cdot hash(s[l \dots r])$ по простому модулю P на Q^{l-1}. Это будет занимать $O(\log P)$ или $O(1)$ и $O(n \log P)$ предпросчета.

2. Простой, но иногда неудобный. Пусть есть $Q^{l_1-1} \cdot hash(s[l_1 \dots r_1])$ и $Q^{l_2-1} \cdot hash(t[l_2 \dots r_2])$. Мы хотим понять, равны ли $s[l_1 \cdots r_1]$ и $t[l_2 \cdots r_2]$.

   Тогда домножим первое число на величину Q^{l_2-1}, а вторую – на Q^{l_1-1}. Теперь получится: $Q^A \cdot hash(s[l_1 \dots r_1])$ и $Q^A \cdot hash(t[l_2 \dots r_2])$, где $A = l_1 + l_2 - 2$. То есть теперь если $hash(s[l_1 \dots r_1]) = hash(t[l_2 \dots r_2])$, то и $Q^A \cdot hash(s[l_1 \dots r_1]) = Q^A \cdot hash(t[l_2 \dots r_2])$; в обратную сторону равенство тоже верно.

4.1.5 Задачи

Предположим, вы забыли как считать z-функцию. Тогда будем делать бинпоиск по длине z-блока для каждой позиции и проверять на равенство подстроки.

```
const int P = 1e9 + 7;
const int Q = 543;

const int N = 1e5 + 1;

int degq[N], h[N];

bool is_equal(int l1, int r1, int l2, int r2) {
  int h1 = h[r1] - (l1 == 0 ? 0 : h[l1 - 1]);
  if (h1 < 0) h1 += P;
  int h2 = h[r2] - (l2 == 0 ? 0 : h[l2 - 1]);
  if (h2 < 0) h2 += P;
  return h1 * 1LL * degq[l2] % P == h2 * 1LL * degq[l1] % P;
}

int main() {
  // h(s1s2s3...) = s1 + s2*Q + s3*Q^2 + ...
  degq[0] = 1;
  for (int i = 1; i < N; ++i)
    degq[i] = degq[i - 1] * 1LL * Q % P;

  string s;
  cin >> s;

  int n = s.size();

  for (int i = 0; i < n; ++i) {
    h[i] = ((i == 0 ? 0 : h[i - 1]) + s[i] * 1LL * degq[i]) % P;
  }

  for (int i = 0; i < n; ++i) {
    int l = 0, r = n - i + 1;

    while (r - l > 1) { // [l; r)
      int m = (r + l) / 2;

      if (is_equal(0, m - 1, i, i + m - 1))
        l = m;
      else
        r = m;
    }

    cout << l << ' ';
  }
}
```

4.2 Хеши для множеств и мультимножеств

Пусть мы теперь хотим сравнивать не строки на равенство за $O(1)$, а множества или мультимножества (множества, в которых каждый элемент может встречаться больше одного раза).

Способ 1: Каждому числу x сопоставим случайное число $r(x)$. Важно, чтобы это сопоставление было одинаковое для x всегда.

Хешом множества $s = \{s_1, s_2, \ldots, s_n\}$ назовем значение выражения

$$h(s) = r(s_1) \oplus r(s_2) \oplus \ldots \oplus \ldots r(s_n)$$

Тогда вставка и удаление из хеша – это $h(s + s_{n+1}) = h(s) \oplus r(s_{n+1})$ и $h(s \setminus s_k) = h(s) \oplus r(s_k)$. Это очень простой и легко пишущийся хеш, рекомендую.

Способ 2: В прошлом способе мы не могли дважды добавить одно и то же число, это проблема. Теперь хешом множества $s = \{s_1, s_2, \ldots, s_n\}$ назовем следующую величину:

$$h(s) = (Q^{s_1} + Q^{s_2} + \ldots + Q^{s_n})\%P$$

Тут тоже нечего комментировать: добавление элемента во множество — это $+=$, удаление – это $-=$.

4.3 Хеши для табличек

Аналогично полиномиальному хешу для строки введем полиномиальный хеш для таблицы:

$$h\left(\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix}\right) = \sum_{ij} Q_1^i Q_2^j a_{ij}$$

Не сложно заметить, что с помощью этого хеша лего вычислять хеши подматриц, если сохранить все префиксные хеши, всё делается аналогично обычному полиномиальному хешу.

4.4 Взлом хеша

Об этой теме можно говорить очень много, буквально этой теме посвящен огромный раздел в Компьютерной Безопасности. О прикладных аспектах (с точки зрения ломания решений на кф) можно прочитать в этом посте на codeforces.