Правило хорошего суффикса

- Суффикс *P* совпал с подстрокой в *T*, а символ *y* – нет

```
T      XXXXXXXXXxXXXXXXXXXXXX          (X - ∀ символы)
P      XzXXXXXXXyXXXX
P              XzXXXXXXXyXXXX
```

- Ближайшая слева копия «хорошего» суффикса занимает его место
- В *T* не будет пропущено ни одного вхождения образца
- Более эффективно: такая копия, что $z \neq y$ (либо *z* не существует) – *сильное правило хорошего суффикса*
- Сильное правило дает линейную оценку алгоритма Бойера-Мура в наихудшем случае
- Слабое правило использовалось в ранних версиях алгоритма

Формулировка правила

- Пусть есть несовпадение в позиции *k* образца *P* или образец совпал (тогда *k* = -1)
- Если 1) *k* = *m*-1, то хорошего суффикса нет – сдвинуть образец на 1 позицию
- Иначе 2) найти в *P* предпоследнее вхождение *w* подстроки *P*[*k*+1..*m*-1] {, предшествующий символ *z* которого не равен *P*[*k*] – *сильное правило*}

Сдвинуть *P* вправо, чтобы копия суффикса заняла его место

```
T       XXXXXXXXXxXXXXXXXXXXXX
P       XzXXXXXXXyXXXX
P               XzXXXXXXXyXXXX
```

Формулировка правила – продолжение

- Пусть такой подстроки *w* нет (при *k* = -1 это заведомо известно)
  - 3) Сдвинуть *P* минимально так, что собственный префикс *P* займет место равного ему суффикса в *P*[*k*+1..*m*-1]:

```
T       XXXXXXxXXXXXXXXXXXX
P       XXXXXXyXXXX
P                XXXXXXyXXXX
```

  - Если 4) такого суффикса нет, передвинуть *P* на *m* позиций
- В 3)–4) перемещение определяется гранью образца *P*
- Не обязательно наибольшей – она не должна превышать длины хорошего суффикса
- В случае 4) эта грань просто равна 0

Препроцессинг для слабого правила

- Для применений варианта 2) – массив *ns*[0..*m*-1] («nearest suffixes») координат ближайших повторных вхождений суффиксов
- Для каждой позиции образца *k* он содержит позицию *j* начала ближайшей слева копии суффикса длины *m*-*k*-1
- Вычисляется за линейное время из массива *bs*, *j*-й элемент которого содержит длину наибольшей грани суффикса *P*[*j*..*m*-1]

```
P      XXXXXXXXXXXXyXXXXXXX

        ↑j          ↑k       ↑m
```

- Сам массив *bs* также вычисляется за линейное время (алгоритм *Suffix-Border-Array*)
- Запись осуществляется в позиции $k > j \Longrightarrow$ совмещение массивов *bs* и *ns* невозможно

Алгоритм *BS-to-NS*

```
BS-to-NS (bs, ns, m)
{
   for  (k = 0; k < m; ++k)  ns[k] = -1;  // Фиктивное значение
   for  (j = 0; j < m - 1; ++j)
   {              // Порядок просмотра bs гарантирует сохранение
   if (bs[j])  //  позиций самых правых копий суффиксов
      {
         k = m - bs[j] - 1; ns[k] = j;
      }
   }
}
```

Препроцессинг для сильного правила

- Массив *nsh*[0..*m*-1] координат ближайших повторных вхождений суффиксов с дополнительным требованием
- Для каждой позиции образца *k* содержит позицию *j* ближайшей слева копии суффикса длины *m*-*k*-1, причем *P*[*j*-1] ≠ *P*[*k*]
- Вычисляется за линейное время из массива *bsm*, *j*-й элемент которого содержит длину наибольшей грани суффикса *P*[*j*..*m*-1] с дополнительным условием о непродолжимости частей граней к началу строки

```
P      XzXXXXXXXXXXyXXXXXXX

         ↑j         ↑k       ↑m
```

- Сам массив *bsm* вычисляется за линейное время (алгоритм *BS-to-BSM*)
- Массив *nsh* можно получить из массива *bsm* выполнением предыдущего алгоритма *BS-to-NS* (*bsm*, *nsh*, *m*)

Препроцессинг для вариантов 3)–4)

- Пусть в *P* не окажется повторных вхождений *w* хорошего суффикса
- Сдвиг образца – на основе его наибольшей грани, не превосходящей длины хорошего суффикса (*m*-*k*-1)

```
T        XXXXXXxXXXXXXXXXXXX
P        XXXXXXyXXXX
P                XXXXXXyXXXX
```

- Сдвиг можно вычислить как *m* - *br*[*k*], где *br*[*k*] – наибольшая грань *P*, не превосходящая *m*-*k*-1 (*br* – «borders, restricted»)
- Массив *br* вычисляется из массива граней *bs*
- Сложность линейна по *m*: каждый элемент *br*[*k*] получает значение ровно один раз; массивы можно совместить

Препроцессинг - иллюстрация

border < m - k $\Leftrightarrow$ k < m - border

$bs[0]=11$

P XXXXXXXXXXXXXXXXXXXXXXXXXXXX (грань всего P)

k-> | $(m-br[0])$

$br[k]=11$

$bs[m-11]=3$

P XXXXXXXXXXXXXXXXXXXXXXXXXXXX (грань грани P)

k-> |

$br[k]=3$

Алгоритм *BS-to-BR*

```
BS-to-BR (bs, br, m)
{
  currBorder = bs[0]; k = 0;
  while (currBorder)
  { //  k < m - currBorder <=> currBorder < m - k
    for  ( ; k < m - currBorder; ++k) br[k] = currBorder;
    currBorder = bs[k]; // Меньшая грань образца (k = m - currBorder)
  }                     // - грань грани
  for  ( ; k < m; ++k) br[k] = 0;
}
```

Сдвиг по правилу хорошего суффикса

```
// nsx – это ns или nsh (слабое или сильное правило)
GoodSuffix-Shift (nsx, br, PosBad, m)
{
  if  (PosBad == m-1) return 1;  // Хорошего суффикса нет
  if  (PosBad < 0)  return m - br[0];  // Образец совпал – сдвиг по наиб. грани

  CopyPos =  nsx[PosBad];   // Вхождение левой копии суффикса
  if  (CopyPos >= 0)  Shift  =  PosBad - CopyPos + 1;
  else  Shift  =  m - br[PosBad];  // Сдвиг по ограниченной наибольшей грани
  return  Shift;
}
```

Алгоритм Бойера-Мура

```
BM (P, T, h) // h = 1 ~ сильное правило хорошего суффикса
{
  // Препроцессинг
  Init(A); // Формирование алфавита
  Position-List (P, A, pl); // Для правила плохого символа
  m = strlen (P);  n = strlen (T);

  Suffix-Border-Array (P, bs); BS-to-BR (bs, br, m);
  if  (h) BS-to-BSM (bs, bs, m);
  BS-to-NS (bs, nsx, m);
  nTextR = m; //  Правая граница «прикладывания» образца
```

Алгоритм Бойера-Мура - окончание

```
while  (nTextR <= n) //  Поиск вхождений
{
   // Сравнение образца с текстом справа налево
   k = m - 1; i = nTextR - 1;
   for  ( ; k >= 0; --k, --i) if  (P[k] != T[i]) break;
   // Результаты сравнения
   if  (k < 0) printf ("Вхождение с позиции %d\n", i + 1);
   // Продвижение по наиболее эффективному правилу
   nShift = Max (BadChar-Shift (pl, T[i], k), GoodSuffix-Shift (nsx, br, k, m) );
   nTextR += nShift;
}
}
```

Алгоритм Карпа-Рабина

- Вместо посимвольного сравнения – двоичная арифметика – учет архитектуры компьютеров
- Замена подстрок числами и сравнение этих чисел (константное время)
- Алгоритм работает за линейное время, но с (малой) возможностью ошибки
- Если уточнять вхождения образца, то сложность составит в худшем случае (при большом числе вхождений) $\Theta(mn)$
- Редко используется непосредственно, но имеет теоретическую значимость как альтернативный подход
- С практической точки зрения: распространяется на другие задачи, например, поиск k образцов за время $O(n)$

Общий подход

- Каждой строке x (на алфавите A) ставится в соответствие *сигнатура* $h(x)$ – целое число (фактически это хеш-функция)
- Количество строк существенно превышает границу целых чисел в процессоре – возможны коллизии
- Для успешного применения сигнатур при поиске вхождений образца P в текст T требуются следующие условия:
  - малая вероятность совпадения сигнатур разных строк
  - эффективное вычисление сигнатуры
- Для краткости обозначим $h(j) = h(T[j..j+m\text{-}1])$, где $m = |P|$
- В алгоритме эффективно вычисляется $h(j+1)$ на основе $h(j)$
- Упрощение постановки задачи: $A = \{‘0’, ‘1’\}$

Взаимно-однозначные сигнатуры

- Пусть P и T – двоичные последовательности
- Перевод в числа (взаимно однозначный):

$H(P) = \sum_{i=0}^{m-1} 2^{m-1-i} P[i]$; $H(j) = \sum_{i=0}^{m-1} 2^{m-1-i} T[j+i]$

- Функция H выдает число по его набору двоичных цифр длины m (с учетом незначащих нулей)
- Позволяет точно сравнивать подстроки (здесь – P и $T[j..j+m\text{-}1]$)
- Схема Горнера (время вычислений $\Theta(m)$):

$H(P) = P[m\text{-}1] + 2(P[m\text{-}2] + 2(P[m\text{-}3] + \ldots + 2(P[1] + 2P[0])\ldots)$

$H(0) = T[m\text{-}1] + 2(T[m\text{-}2] + 2(T[m\text{-}3] + \ldots + 2(T[1] + 2T[0])\ldots)$

Рекуррентное вычисление сигнатур

```
H(j):     XXXXXXXXXXXXXXXXXXXXX
H(j+1):   XXXXXXXXXXXXXXXXXXXXX
            ↑j        ↑j+m
```

- Рекуррентная формула для последовательного вычисления:

  $H(j+1) = 2\,(H(j) - 2^{m-1}T[j]) + T[j+m]$

- Если 2^{m-1} подготовить заранее, то вычисление $H(j)$, $j > 0$ займет константное время
- Проблема: значения $H(j)$ и $H(P)$ экспоненциально возрастают по m
- При длинных образцах P (например, $m > 64$) они слишком большие для вычисления и сравнения

Метод Карпа-Рабина

- Вероятностный подход, оперирующий ограниченными числами
- Допускаются коллизии сигнатур, однако их вероятность мала
- Важное достижение – строгое обоснование этого факта

Ограниченные сигнатуры

- Основная идея: вместо чисел $H(P)$ и $H(j)$ использовать остатки от их деления на некоторое число
- Модулярная арифметика по модулю q, где q – простое число:

  $h(P) = H(P) \bmod q$; $h(j) = H(j) \bmod q$
- Такая хеш-функция h отображает 2^m всевозможных двоичных строк длины m на множество целых чисел $0..q$-1
- Каждому числу соответствуют в среднем $2^m/q$ двоичных строк
- Вероятность совпадения сигнатур двух различных строк равна $1/q$

  (число равных вариантов / число всех вариантов)

Вычисление ограниченных сигнатур

$H(j+1) = 2\,(H(j) - 2^{m-1}T[j]) + T[j+m]$ (1)

- Модулярная арифметика «сохраняет операции»
- Если • – арифметическая операция (+, -, x), а $\bullet_q$ – она же по модулю q, то для любых целых положительных r и s справедливо $(r \bullet s) \bmod q = (r \bmod q) \bullet_q (s \bmod q)$
- Применяя это свойство к (1), получим:

$h(j+1) = (\,2(h(j) - (2^{m-1} \bmod q)\,T[j]) + T[j+m])\,) \bmod q,$

- При этом справедливо

$(2^{m-1} \bmod q) = 2\,(2^{m-2} \bmod q) \bmod q$

Выбор основания счисления q

- Достаточно малое, чтобы арифметика была эффективной (числа не занимали слишком много разрядов)
- Достаточно большое, чтобы вероятность ложных совпадений образца P и подстроки T не оказалась существенной
- Для положительного q *обозначим* $\pi(q)$ количество простых чисел, не превосходящих q
- *Теорема* (*Карпа-Рабина*). P и T – некоторые строки, причем $mn \geq 29$, где $m = |P|$ и $n = |T|$. Пусть l – положительное число. Если q – случайное простое число $\leq l$, то вероятность ложного совпадения P и T ($h(P) = h(T)$) не превосходит $\pi(mn) / \pi(l)$.

Алгоритм Карпа-Рабина

- Функция *Gorner2Mod* (*S*, *m*, *q*) вычисляет по схеме Горнера значение многочлена степени *m* с коэффициентами *S*[0..*m*-1] по модулю *q* при *x* = 2

Gorner2Mod (S, m, q)

{

res = 0;

for (i = 0; i < m; ++i) res = (res * 2 + S[i]) **mod** q;

return res;

}

Алгоритм Карпа-Рабина – основная функция

```
KR (P, T, q)
{
  // Инициализация
  m = strlen(P); p2m = 1;  // Для вычисления p2m = 2**(m-1) mod q
  for (i = 0; i < m-1; ++i) p2m  = (p2m * 2) mod q;
  hp = Gorner2Mod (P, m, q);  ht = Gorner2Mod (T, m, q);
```

Алгоритм Карпа-Рабина - окончание

```
for (j = 0;  j <= n – m; ++j)  // Поиск вхождений
  {
     if (ht == hp)
     { // Уточнить, действительно ли совпали строки
        k = 0;
        while  (k < m && P[k] == T[j+k])  ++k;
         if (k == m) printf ("Вхождение с позиции %d\n", j);
     }
     ht = ( (ht – p2m * T[j]) * 2 + T[j + m] ) mod q;
     if (ht < 0) ht += q;  // Модулярная арифметика
  }
}
```

О реализации

- На заключительной итерации цикла выход символа $T[j + m]$ за пределы текста не приведет к катастрофе, если строка завершается нулевым кодом
- В программной реализации необходимо предусмотреть преобразование символов ‘0’, ‘1’ в двоичные числа
- Умножения на 2 могут быть эффективно выполнены сдвигами