Алгоритм Кнута-Морриса-Пратта

- Наиболее известный алгоритм с линейной сложностью
- Редко используется на практике, т.к.
- Уступает по эффективности другим классическим алгоритмам
- Просто объясняется
- Легко обосновывается линейная оценка
- Имеет модификацию *online*
- Создает основу для алгоритма Ахо-Корасик, вычисляющего все вхождения в текст любого образца из заданного набора

Общая идея

- Схема соответствует наивному алгоритму
- Сдвиг образца по тексту – более чем на одну позицию
- Используется массив граней

Схема алгоритма – ускорение продвижения

- Пусть при «прикладывании» образца *P* к тексту *T* совпали *k* символов (весь образец или его часть)
- Продвижение на 1 позицию (как в наивном алгоритме)

```
T          XXXXXXXXXXXXXXXXXXXX
P[0..k-1]  XXXXXXXXXXX
P[0..k-1]   XXXXXXXXXXX   - необходима грань длины k-1
```

- Продвижение на несколько позиций

```
P[0..k-1]  XXXXXXXXXXX
P[0..k-1]      XXXXXXXXXXX   - необходима грань
```

- Рекомендуемое перемещение: *k* - *bp*[*k*-1]

Схема алгоритма – сокращение сравнений

```
T            XXXXXXXXXXXXXXXXXXXX
P[0..k-1]    XXXXXXXXXX
P[0..k-1]        XXXXXXXXXXX
```

- После продвижения сравнению с текстом подлежат лишь символы образца справа от префикса длины $bp[k\text{-}1]$
- При меньших гранях продвижение быстрее, но
- Чем больше грань, тем меньше потребуется сравнений
- Предыдущее сравнение было неуспешным из-за несовпадения $P[k]$ с некоторым символом текста $T[i]$ (либо k – длина образца)
- После перемещения с тем же $T[i]$ сравнивается символ $P[bp[k\text{-}1]]$
- По тексту – поступательное движение (i не убывает)

Модифицированный массив граней

- Полезно гарантировать, что вновь сравниваемый с $T[i]$ символ образца $P[bp[k\text{-}1]]$ будет отличен от $P[k]$
- Гарантию дает использование модифицированного массива граней *bpm* вместо *bp*
- Он определяется дополнительным условием: $P[bpm[k\text{-}1]] \neq P[k]$ для всех k
- Если образец совпадет полностью, фиксируется вхождение, а продвижение выполняется аналогично – k заменяется на $bpm[k\text{-}1]$

Алгоритм – инициализация

```
KMP (P, T)
{
   Prefix-Border-Array (P, bpm);  // Построение массива граней
   m = strlen (P);   n = strlen (T);
   BP-to-BPM (bpm, bpm, m);  // Модифицированный массив граней
   k = 0;  // Текущий индекс в образце
```

Алгоритм – основная часть

```
// Цикл по символам текста T
for  (i = 0; i < n; ++i)
{  // Быстрые продвижения при фиксированном i
   while (k && P[k] != T[i]) k = bpm[k-1];
   // «Честное» сравнение очередной пары символов
   if (P[k] == T[i]) ++k;
   if (k == m)
   {
      printf ("Вхождение с позиции %d\n", i - k + 1);
      k = bpm[k-1];
   }
}
}
```

Алгоритм КМП – оценка сложности

- Построение массивов граней – $\Theta(m)$
- Цикл *for* повторяется n раз, что при отсутствии внутреннего цикла составило бы $\Theta(n)$
- Вложенный *while* обещает квадратичную сложность, однако целесообразно оценить его общее количество выполнений
- Переменная k принимает неотрицательные значения и начинает с нуля. Увеличивается на ≤ 1 на каждом шаге *for*
- Каждый шаг *while* уменьшает k на ≥ 1
- Общее число уменьшений (шагов цикла *while* за все время) $\leq$ общего числа увеличений (n), что дает $O(n)$
- Суммарная сложность алгоритма равна $\Theta(m+n)$

Алгоритмы реального времени

- Алгоритм работает *online* («в реальном времени»), если он считывает текст последовательно без возвратов, при этом на каждый символ тратит константное время (не зависящее от n, m)
- Полезно, например, при больших текстах или маленькой памяти
- Перемещения по тексту могут быть длиннее
- Линейная оценка сложности очевидна по определению
- КМП близок к таковому: текст T просматривается однократно
- Однако $T[i]$ при несовпадениях может использоваться для сравнений несколько раз – $\Omega(m)$ (в соответствии с *while*)
- Идея: «развернуть» цикл *while* в матрицу, чтобы подходящее k вычислять прямым доступом по символу $T[i]$ и исходному k

КМП – в алгоритм реального времени

- Вместо $bpm[0..m\text{-}1]$ строится матрица $bpm2[0..m\text{-}1][0..|A|]$, где $|A|$ – размер алфавита A
- $bpm2[k][x]$ – длина наибольшей грани + ее префикс ограничен в образце справа символом x, т.е. $P[bpm2[k][x]] = x$ ($x \sim T[i]$)
- Есть линейный по m алгоритм ее построения, но зависит от $|A|$
- При несовпадении $P[k]$ и $T[i]$ длина перемещения по тексту вычисляется как k - $bpm2[k\text{-}1][T[i]]$ (вместо цикла *while*)
- В результате символ $T[i]$ автоматически совпадает и пропускается
- Далее сравниваются символы $P[bpm2[k\text{-}1][T[i]]+1]$ и $T[i+1]$
- Символ $T[i]$ может использоваться максимум дважды – для сравнения и для доступа к матрице

Алгоритм Бойера-Мура

- Считается одним из наиболее практичных
- Широкий простор для творчества
- Множество последователей
- Множество модификаций
- Оценка сложности – нетривиальна
- Линейная оценка в худшем случае для некоторых модификаций
- Многочисленные экспериментально-практические исследования
- На практике – ожидаемое время работы *сублинейно* ($o(m + n)$)

Общие идеи

- Последовательно «прикладывает» образец к тексту и сравнивает символы
- Образец сдвигается вправо
- Просмотр символов на стадии сравнения – *справа налево*
- Сдвиг образца по тексту – более чем на одну позицию
- Два собственных правила ускорения сдвигов: «по плохому символу» и «по хорошему суффиксу»

Правило плохого символа

- Сравнение символов – справа налево (от конца *P* к началу)
- Пусть в *P* последний символ *y*, а символ в тексте – *x*

```
T       XXXXXXXXXxXXXXXXXXXXXX          (x ~ любые символы)
P       XXXxXXXXXy
P             XXXxXXXXXy
```

- Можно сразу сдвинуть *P* вправо до крайнего правого *x* в *P*
- Аналогично – для не последнего символа

```
T       XXXXXXXXXxXXXXXXXXXXXX
P       XXXxXXXXXyXX
P             XXXxXXXXXyXX
```

- Сдвиг – до ближайшего слева совпадающего символа

Функция сдвигов

- $R(x)$ – позиция крайнего правого вхождения x в P для $\forall\, x \in A$
- Если x не входит в P, положим $R(x) = -1$
- Функция $R(x)$ вычисляется за однократный просмотр образца, то есть за время $\Theta(m)$, где $m = |P|$
- В предыдущем примере (`XXXxXXXXXyXX`) $R(x) = 3$, $R(y) = 9$
- Правило сдвига по плохому символу, использующее функцию R

Простое правило плохого символа

- Пусть при сопоставлении *P* с участком *T* правые *n*-*k*+1 символов *P* совпали с символами в *T*
- Символ *P*[*k*] не совпадает со своей парой в позиции *i* текста *T*
- Тогда сдвинуть *P* вправо по тексту на max { 1, *k* - *R*(*T*[*i*]) } позиций:
  - Если правое вхождение в *P* символа *T*[*i*] = позиции *j* < *k* (в том числе *j* = -1), то *j*-й символ в *P* передвигается к *i*-му символу в *T*
  - Иначе (при *j* > *k*) *P* сдвигается вправо на одну позицию
- Цель – возможный сдвиг *P* более чем на одну позицию
- Хорошо подходит для учебных целей, однако на практике в общем случае малоэффективно
- При маленьком алфавите (ДНК) весьма вероятно *j* > *k*

Расширенное правило плохого символа

- Пусть при сравнении – несовпадение в позиции *k* образца *P*
- Пусть *x* – соответствующий несовпадающий символ в *T*
- Сдвинуть *P* вправо, совместив с *x* ближайшее его вхождение в *P* слева от позиции *k*
- Если таких вхождений *x* в *P* нет, то *P* сдвигается вправо за позицию несовпадения

```
T   XXXXXXXXXxXXxXXXXXXXX
P   XXXxXXXXXyXXx
P         XXXxXXXXXyXXx
            K=9
```

```
T   XXXXXXXXXxXXxXXXXXXXX
P   XXXzXXXXXyXXx
P             XXXzXXXXXyXXx
               K=9
```

Сравнение 2-х правил

- Простое правило использует $O(|A|)$ памяти для массива R и одно обращение к нему при каждом несовпадении
- Расширенное более эффективно, но требует дополнительной памяти. Может увеличиться и время построения функции сдвигов
- Расширенное правило: $O(m|A|)$ памяти и одно обращение к матрице сдвигов при каждом несовпадении
- Другая реализация: $O(m)$ памяти, но несколько обращений к вспомогательной структуре
- Исходная версия алгоритма Бойера-Мура использовала простое правило плохого символа

Реализация расширенного правила

- Препроцессинг образца: для ∀ позиции k в P и ∀ символа $x \in A$ найти позицию ближайшего вхождения x в P слева от k
- Хранение – двумерный массив $R2$ размера m x $|A|$
- При несовпадении в позиции k образца и несовпадающем в T символе x для определения сдвига выбирается элемент $R2[k, x]$
- Выборка осуществляется быстро
- Размер массива и время его построения могут представляться чрезмерными

Реализация расширенного правила II

- Построение массива *pl*[0..|*A*|-1], который для каждого символа алфавита содержит список позиций его вхождений в образец *P*
- Просмотреть образец справа налево и индекс каждой позиции занести в список, соответствующий ее символу
- В результате непустые списки упорядочены по убыванию
- Построение – за время $\Theta(m)$, суммарная память – в объеме $\Theta(m)$
- При несовпадении в позиции *k* и символе *x* в *T* просмотреть список символа *x*, до достижения элемента, меньшего *k*
- При обратно-упорядоченном списке время просмотра – $O(m-k)$
- До несовпадения должны были совпасть $O(m-k)$ символов справа $\Rightarrow$ время работы возрастает не более чем вдвое в худшем случае
- Для ускорения можно применить двоичный поиск в списке

Формирование массива списков позиций

```
Position-List (S, A, pl)
{
  nA = strlen(A); m = strlen(S);
  for  (i = 0; i < nA; ++i) pl[i] = null; // Можно обнулить быстрее
  for  (k = m-1; k >= 0; --k)
  {
    ich = S[k] – A[0];  //  Индекс от начального символа алфавита
    if ( !pl[ich] ) pl[ich] = new IntList;
    pl[ich] += k;  // Добавить к списку – реализуется типом IntList
  }
}
```

Вычисление сдвига по плохому символу

```
BadChar-Shift (pl, CharBad, PosBad)
{
  if  (PosBad < 0) return 1; // Образец совпал – сдвиг на 1
  nPos = -1;  // Искомая позиция слева от плохого символа
  List = pl[CharBad];  // Список позиций данного символа CharBad
  if  (List)
  {  //  Список не пуст
    nLen = List.Length;  // Длина списка
    //  Ищем элемент, меньший чем плохая позиция
    for  (k = 0; k < nLen; ++k) if  (List[k] < PosBad) { nPos = List[k];  break;  }
  }
  return  PosBad - nPos;
}
```

Поиск вхождений – начальная версия БМ

```
BM (P, T)
{
  Init(A); // Формирование алфавита
  Position-List (P, A, pl);
  m = strlen (P);  n = strlen (T);

  //  Поиск вхождений
  nTextR = m;  //  Правая граница «прикладывания» образца
```

Поиск вхождений – начальная версия БМ

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

Правило плохого символа

- Хорошо работает на практике
- Недостаточно эффективно при маленьком алфавите
- Квадратичная оценка времени работы в худшем случае, например: T = “AAAAAAAAAAAAAAAAAA” и P = “BAAAAA” $\sim \Omega\,(m\,n)$