# Алгоритмы сравнения и анализа строк

- Улучшение алгоритмов – за счет пропусков избыточных сравнений
- Решение задач – в 2 этапа:
    - изучение структуры строки (препроцессинг)
    - собственно обработка, на основе информации о структуре
- Наиболее известны две схемы предварительного анализа – нахождение граней и *Z*-блоков строк.

# Грани строк

- *Определение*. *Грань* строки – любой собственный префикс, равный суффиксу.
  - «Собственный» исключает грань, совпадающую со всей строкой
  - Любая непустая строка имеет пустую грань (длины 0)
  - Пример: строка *ABAABABAABAAB* содержит непустые грани: *AB* и *ABAAB*
- Особый интерес – наибольшая грань (в предыдущем примере *ABAAB*).
- Строка *ABAABAAB* имеет такие же грани, но здесь наибольшая грань *ABAAB* частично перекрывается
- Наивный алгоритм: сравнение префиксов *S*[0..*i*-1] с суффиксами *S*[*n*-*i*..*n*-1] при *i* = *n*-1, n-2, …, 1. Результат дает их первое совпадение
  - Сложность – $O(n^2) \sim$ (n-1) + (n-2) + … + 2 + 1

# Наибольшая грань – наивный алгоритм

```
Naive-Max-Border (S)
{
  n = strlen (S); br = 0;
  for  (i = n - 1; !br && i; --i)
  {  // i – предполагаемая длина грани
    j = 0;
    while  (j < i && S[j] == S[n - i + j])  ++j;
    if  (j == i)  br = i;
  }
  return  br;
}
```

# Массив граней (border array)

- Массив *bp*[0..*n*-1] для строки *S* содержит длины наибольших граней всех ее подстрок *S*[0..*i*], *i* = 0, 1, …, *n*-1, то есть префиксов («префикс-функция») *S*.
- Пример – массив граней строки $A\ B\ A\ A\ B\ A\ B\ A\ A\ B\ A\ A\ B$:

  0 0 1 1 2 3 2 3 4 5 6 4 5

- Важная роль в приложениях к исследованию строк
- С применением алгоритма *Naive-Max-Border* сложность вычисления массива *bp*[0..*n*-1] – $O(n^3)$
- Неприемлемо для ускорения наивного поиска образца в тексте, работающего за квадратичное время в худшем случае.
- Более эффективный алгоритм – на основе свойств граней

# Свойства граней и их массива

*ABAAABACBCAABAAAB*

1) $bp[0] = 0$ (строка длины 1 имеет лишь собственную подстроку ε).
2) Если $S[0..i]$ ($0 < i < n$) имеет грань $k > 0$, то $S[0..i-1]$ имеет грань $k - 1$. Таким образом, $bp[\mathrm{i}] \le bp[i\text{-}1] + 1$.
3) $bp[\mathrm{i}] = bp[i\text{-}1] + 1 \Leftrightarrow S[i] = S[bp[i\text{-}1]]$ ($bp[i\text{-}1]$ – позиция справа от префикса $S[0..bp[i\text{-}1]\text{-}1]$ – наибольшей грани подстроки $S[0..i\text{-}1]$).
4) Если $b$ – грань $S$, а $b'$ – грань $b$, то $b'$ есть грань $S$. Отсюда $bp[bp[i]\text{-}1]$ – длина второй по величине грани строки $S[0..i]$ и т.д. Эта строго убывающая последовательность заканчивается нулем.

- Возможность вычисления $bp[i]$ на основе

  $bp[0], bp[1], \ldots, bp[i\text{-}1]$

# Вычисление $bp[i]$ по $bp[0], \ldots, bp[i\text{-}1]$

- Согласно (2), положительное $bp[i]$ получается увеличением на 1 длины некоторой грани предыдущей строки – $S[0..i\text{-}1]$.
- Если $S[i] = S[bp[i\text{-}1]]$, то (3) $bp[\mathrm{i}] = bp[i\text{-}1] + 1$.
- Иначе рассмотреть вторую по величине грань в $S[0..i\text{-}1]$, то есть (4) проверить равенство $S[i] = S[bp[bp[i\text{-}1]\text{-}1]]$. Если выполнено, взять $bp[\mathrm{i}] = bp[bp[i\text{-}1]\text{-}1] + 1$.
- При необходимости рассмотреть следующие по уменьшающейся величине грани подстроки $S[0..i\text{-}1]$.
- Процесс останавливается при достижении равенства либо когда очередная грань окажется пустой. В последнем случае в качестве $bp[\mathrm{i}]$ взять 1 или 0, в зависимости от истинности $S[i] = S[0]$.

# Алгоритм вычисления массива граней

```
Prefix-Border-Array (S, bp)
{
  n = strlen (S); bp[0] = 0;
  for  (i = 1; i < n; ++i)
  {  // i –длина рассматриваемого префикса
    bpRight = bp[i - 1]; // Позиция справа от предыдущей грани
    while (bpRight  && (S[i] != S[bpRight]) ) bpRight = bp[bpRight - 1];
    // Длина на 1 больше, чем позиция
    if  (S[i] == S[bpRight]) bp[i] = bpRight + 1;
    else  bp[i] = 0;
  }
}
```

# Сложность вычисления массива граней

- Цикл *for* повторяется ровно $n$ – 1 раз, что при отсутствии вложенных циклов составило бы $\Theta(n)$
- Вложенный цикл *while* обещает квадратичную сложность, однако целесообразно оценить его общее количество выполнений
- Переменная *bpRight* принимает неотрицательные значения и начинает с нуля. Увеличивается на $\leq 1$ на каждом шаге *for*, начиная со 2-го
- Каждый шаг *while* уменьшает значение *bpRight* на $\geq 1$
- Общее число уменьшений (шагов цикла *while* за все время) $\leq$ общего числа увеличений, то есть $n$ – 2, что дает $O(n)$
- Таким образом, вычислительная сложность рассматриваемого алгоритма равна $\Theta(n)$

# Массив граней суффиксов

- Массив *bp* содержит длины наибольших граней для всех префиксов строки *S*
- Аналогичная задача: нахождение массива *bs*, содержащего длины наибольших граней для всех суффиксов строки *S*
- Способ решения: переписать *S* в обратном порядке, построить ее массив *bp*, который в обратном порядке записать в *bs*
- Приведем непосредственный алгоритм, сложность аналогичная – $\Theta(n)$

# Вычисление массива граней суффиксов

```
Suffix-Border-Array (S, bs)
{
  n = strlen(S); bs[n - 1] = 0;
  for  (i = n - 2; i >= 0; --i)
  {
    bsLeft = bs[i + 1]; // Позиция с конца слева от предыдущей грани
    while (bsLeft && (S[i] != S[n - bsLeft - 1]) ) bsLeft = bs[n - bsLeft];
    // Длина на 1 больше, чем позиция
     if (S[i] == S[n - bsLeft - 1]) bs[i] = bsLeft + 1;
    else bs[i] = 0;
  }
}
```

# Применение: поиск вхождений образца

- Эффективный поиск вхождений образца *P* в текст *T*
  - Взять символ # не из алфавита *A* и построить строку *S* = *P*#*T*
  - Для строки *S* вычислить массив *bp*
  - В *bp* найти все значения = *m* (длине *P*). Индекс *i* каждого из них (уменьшенный на *m*+1) укажет правую координату вхождения *P* в *T*.
- Симметричный способ
  - Построить строку *T*#*P* и для нее найти массив *bs*
  - В *bs* найти все элементы = длине *P.* Их индексы соответствуют левым координатам вхождений *P* в *T.*
- Сложность обоих алгоритмов – $\Theta(m+n)$
- В массивах *bp* и *bs* здесь достаточно хранить лишь *m элементов*

# Модифицированные массивы граней

- В ряде приложений используются модифицированные массивы граней (например, в алгоритме Кнута-Морриса-Пратта)
- Дополнительное условие: непродолжимость компонент граней
- Для массива *bp*: *bpm*[*i*] – длина такой наибольшей грани подстроки $S[0..i]$, что $S[bpm[i]] \neq S[i+1]$
- Ниже приводится алгоритм вычисления модификации, сложность – $\Theta(n)$

# Построение модифицированного массива

```
Prefix-Border-ArrayM (S, bp, bpm)
{
  n = strlen (S);
  bpm[0] = 0; bpm[n-1] = bp[n-1];
  for  (i = 1; i < n - 1; ++i)
  {  //  Проверка совпадения следующих символов
    if  (bp[i] && (S[bp[i]] == S[i+1]))  bpm[i] = bpm[bp[i] - 1];
    else  bpm[i] = bp[i];
  }
}
```

*Примечание*. Для меньших граней проверка выполняется ранее.

# Структура массивов граней

- Цель – преобразование *bp* в *bpm* без использования строки *S*
- Массив *bp* состоит из *серий* возрастающих на 1 целочисленных подпоследовательностей, возможно, разделенных нулями
- Из серии в *bpm* попадает последний элемент, а каждый предыдущий пересчитывается по меньшей (вложенной) грани
- При преобразовании *bp* в *bpm* вместо сравнения последующих символов грани ($S[bp[i]] = S[i+1]$) можно тестировать $bp[i]$ на продолжение серии ($bp[i] + 1 = bp[i+1]$)

# Примеры

- *S*: C A C Z Z Z C A C A

  *bp*: 0 0 1 0 0 0 1 2 3 2

  *bpm*: 0 0 1 0 0 0 0 0 3 2

- *S*: A B X A B Z M A B X A B Z

  *bp*: 0 0 0 1 2 0 0 1 2 3 4 5 6

  *bpm*: 0 0 0 0 2 0 0 0 0 0 0 2 6

# Преобразование *bp* в *bpm* без S

```
BP-to-BPM (bp, bpm, n)
{
  bpm[0] = 0; bpm[n-1] = bp[n-1];
  for (i = 1; i < n - 1; ++i)
  {
    // Проверка «совпадения следующих символов»
    if (bp[i] && (bp[i] + 1 == bp[i+1]) ) bpm[i] = bpm[bp[i] - 1];
    else bpm[i] = bp[i];
  }
}
```

*Примечание*. Синим выделена измененная часть.

# Задача преобразования *bpm* в *bp*

- Из каждой серии массива *bp* в *bpm* попал последний элемент, а предыдущие пересчитаны по меньшим граням
- Идея обратного преобразования (*bpm* в *bp*) – в восстановлении таких серий, двигаясь от конца к началу
- Нули просто переписываются в *bp*
- После нулей положительный элемент в *bpm* рассматривается как окончание бывшей возрастающей серии, переписывается в *bp*
- На следующих шагах – попытки «достроить» серию в обратном порядке
- Если в аналогичной позиции массива *bpm* окажется больший элемент, то он послужит концом в *bp* предыдущей серии

# Преобразование *bpm* в *bp*

*BPM-to-BP* (*bpm*, *bp*, *n*)

```
{
   bp[n-1] = bpm[n-1]; bp[0] = 0;
   for  (i = n - 2; i > 0; --i) bp[i] = max (bp[i + 1] - 1, bpm[i]);
}
```

- Вывод: массивы *bp* и *bpm* имеют равные выразительные возможности

# Симметричная задача – модифицированный массив для суффиксов

- Для массива *bs*: $bsm[i]$ – длина такой наибольшей грани подстроки $S[i..n\text{-}1]$, что $S[n\text{-}bsm[i]\text{ -}1] \neq S[i\text{-}1]$
- Вариант решения – использовать массивы *bp* и *bpm для обратной строки*, путем их переписывания в обратном порядке
- Ниже приводятся соответствующие непосредственные алгоритмы, сложность – $\Theta(n)$

# Преобразование *bs* в *bsm*

```
BS-to-BSM (bs, bsm, n)
{
  bsm[n-1] = 0; bsm[0] = bs[0];
  for  (i = n - 2; i > 0; --i)
  {
    //  Проверка «совпадения предыдущих символов»
    if  (bs[i] && (bs[i] + 1 == bs[i-1]))  bsm[i] = bsm[n - bs[i]];
    else  bsm[i] = bs[i];
  }
}
```

# Преобразование *bsm* в *bs*

*BSM-to-BS* (*bsm*, *bs*, *n*)

{

   bs[0] = bsm[0]; bs[n-1] = 0;

   **for**  (i = 1; i < n - 1; ++i) bs[i] = max (bs[i - 1] - 1, bsm[i]);

}

# Важное свойство преобразований

- Рассмотренные алгоритмы корректно работают «на месте», то есть в качестве параметров можно передавать

  $bp = bpm$ или $bs = bsm$