Лекция 12

АЛГОРИТМЫ ПОИСКА

План лекции

- Поиск в массивах и списках
  - Линейный поиск
  - Бинарный поиск
- Поиск подстроки
  - Наивный поиск подстроки
  - Алгоритм Рабина-Карпа
  - Алгоритм Бойера-Мура
  - Алгоритм Кнута-Мориса-Прата

Поиск в массивах и списках

- Значения элементов массива (списка) делятся на ключ и произвольные данные
  `struct KeyData { K key; T data; };`
- Ключ можно рассматривать как значение функции T -> K, которая вычисляет ключ key на основании (сколь угодно сложного) анализа данных data
- Алгоритм поиска в массиве (списке) находит индекс элемента массива (адрес элемента списка), имеющего заданный ключ

Линейный (последовательный) поиск

- Последовательный просмотр ячеек
- Останов, если найден нужный ключ или кончились ячейки
- Число сравнений в худшем случае O(число ячеек)

- Условия применимости
  - Либо отсутствует линейный порядок на множестве ключей
  - Либо время поиска не существенно с точки зрения программиста (число ячеек заведомо невелико, 1-кратный поиск, и т.п.)

- Многократный поиск в большом числе ячеек – либо сортировка + бинарный поиск для массива, либо ДДП

Линейный поиск в списке

```
place_t linear_search (list_t L, K key)
{
    place_t p;
    for (p = begin(L); p != end(); p = next(p))
        if (get_value(p).key == key) return p;
    return end(); // элемент не найден
}
```

// Как для массива?

Бинарный поиск в упорядоченном массиве

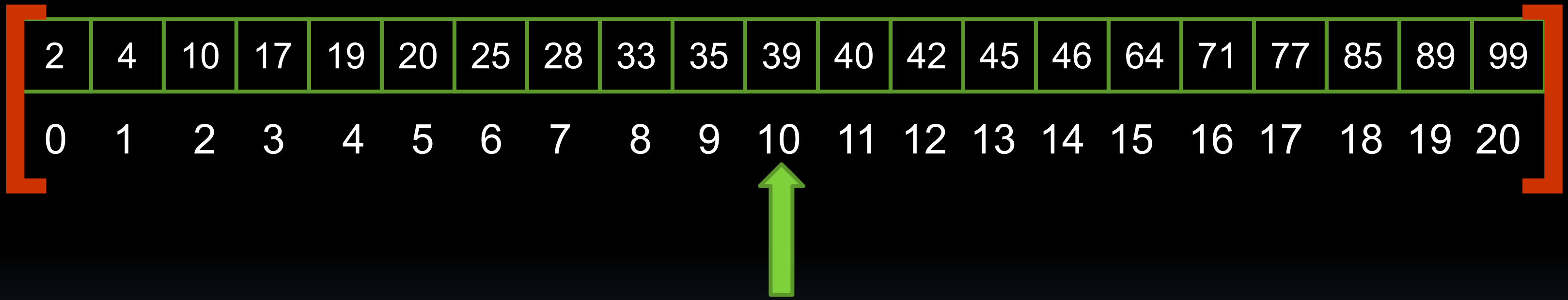
- На каждом шаге делим массив пополам и на следующем шаге продолжаем поиск в той половине, которая должна содержать искомый элемент
- Применяется к упорядоченным массивам
- Число сравнений в худшем случае O (log2(размер массива))
- Требуется линейный порядок на множестве ключей
- Применяется к большим массивам

Бинарный поиск в упорядоченном массиве

```
int binary_search(const struct KeyData A[], int N, K key)
{
     int L = 0, R = N-1;
     do {
          int M = (L+R)/2;
          if (key == A[M].key) return M;
          if (A[M].key < key)
               L = M + 1;
          else
               R = M - 1;
     } while (L <= R);
      return -1;
}
// Почему число сравнений O(log2(N))?
```

Бинарный поиск в упорядоченном массиве

X = 33



2	4	10	17	19	20	25	28	33	35	39	40	42	45	46	64	71	77	85	89	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

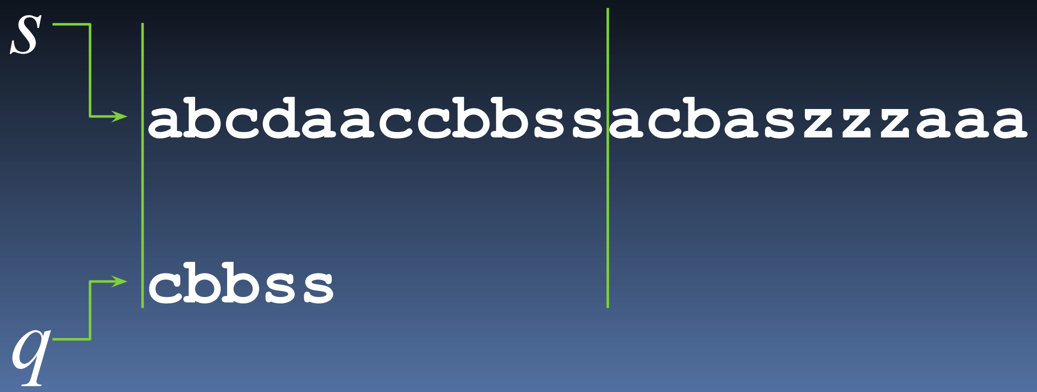
План лекции

- Поиск в массивах и списках
  - Линейный поиск
  - Бинарный поиск
- Поиск подстроки
  - Наивный поиск подстроки
  - Алгоритм Рабина-Карпа
  - Алгоритм Бойера-Мура
  - Алгоритм Кнута-Мориса-Прата

Поиск подстроки

- Даны строка s из N элементов (текст) и строка q из M <= N элементов (образец)
- Требуется найти индекс k, указывающего на начало первого вхождения образца q в текст s

  q[0] = s[k], q[1] = s[k+1], ..., q[M−1] = s[k+M − 1]



Наивный (прямой) поиск подстроки

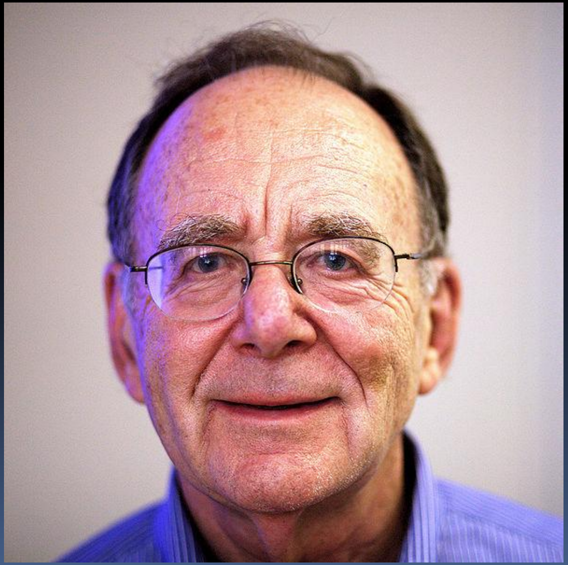
- Шаг 1
  - «Прикладываем» левый край образца к левому краю текста, К = 0
- Шаг 2
  - Проверяем, входит ли образец в текст, начиная с К-й позиции, последовательным сравнением символов образца q[j] с символами текста s[K+j] слева направо
- Шаг 3
  - Если имеем M совпадений, то образец в тексте найден – конец работы
  - Если K+M >= N, то образец не найден – конец работы
  - Иначе K = K+1 и переходим к шагу 2
- В худшем случае O((N - M)*M) сравнений

Прямой поиск подстроки

```
int  naive_substring_search(
    const char s[],  int N, const char q[], int M)
{
    int  k; // смещение образца по тексту
    for (k = 0; k < N-M; ++k)
    {
        int j; // смещение по образцу
        for (j = 0; s[k+j]==q[j]; ++j)
            if (j == M-1) return k; // нашли
    }
    return -1;          // не нашли
}
```

Алгоритм Рабина-Карпа

- Майкл Рабин р. 1932
- Ричард Карп р. 1935
- Karp, Richard M.
  Rabin, Michael O.
  Efficient randomized pattern-matching algorithms // IBM Journal of Research and Development Vol. 31 (2), pp. 249—260, March 1987



Алгоритм Рабина-Карпа

- Быстрый поиск *нескольких* образцов в одном тексте
- Уменьшение числа сравнений в наивном поиске подстроки за счёт использования хэш-функции (разновидность контрольной суммы)
- Хэш-функции преобразуют строки (в общем случае – данные) в числовые значения – т.н. хэш-значения
- Алгоритм Р.-К. использует тот факт, что одна и та же хэш-функция преобразует одинаковые строки в одинаковые хэш-значения

Алгоритм Рабина-Карпа

- Шаг 1
  - Прикладываем левый край образца к левому краю текста, К = 0
  - Вычисляем хэш-значения hq и hs для q и для s[0…M-1]
- Шаг 2
  - Если hq == hs, то проверяем, входит ли образец в текст, начиная с К-й позиции, последовательным сравнением символов образца q[j] с символами текста s[K+j] слева направо, j=0…M-1
- Шаг 3
  - Если имеем M совпадений, то образец в тексте найден – конец работы
  - Если K+M >= N, то образец в тексте не найден – конец работы
  - Иначе вычисляем hs для s[K+1…K+M], используя hs для s[K…K+M-1], К = К+1 и переходим к шагу 2

Алгоритм Рабина-Карпа

```
int  rk_substring_search(
      const char s[],  int N, const char q[], int M)
{
      int k; // смещение образца по тексту
      int hs = rk_hash(s, M);
      int hq = rk_hash(q, M);
      for (k = 0; k < N-M; ++k)
      {
            int j; // смещение по образцу
            if (hs == hq) for (j = 0; s[k+j]==q[j]; ++j)
                  if (j == M-1) return k; // нашли
            // время работы rk_hash_update должно быть O(1)
            hs = rk_hash_update(hs, s[k], s[k+M], M);
      }
      return -1;          // не нашли
}
```

Простая хэш-функция

```
// hs = s[0]+s[1]+…+s[M-1]
// чем плоха такая хэш-функция?

int  rk_hash(const char s[], int M)
{
    int h = 0, i;
    for (i = 0; i < M; ++i) h += s[i];
}

int rk_hash_update(int h, char out, char in, int M)
{
    return h-out+in; // M не используется
}
```

Улучшенная хэш-функция

```
static const int  rk_hash_p         = "хорошее" простое число;
static const int  rk_hash_n         = 256; // число символов в алфавите
// hs = ( s[0]*n^0+s[1]*n^1+…+s[M-1]*n^(M-1) ) mod p
```

```
int  rk_hash(const char s[], int M)
{
    int h = 0, w = 1, i;
    for (i = 0; i < M; ++i)
        h += (w*s[i])%rk_hash_p,
        w = (w*rk_hash_n)%rk_hash_p;
}
```

```
int rk_hash_update(int h, char out, char in, int M)
{
    static int nM = 0;
    if (nM == 0) nM = (rk_hash_n M-1 ) mod rk_hash_p;
    return ((h-out)/rk_hash_n + in*nM)%rk_hash_p;
}
```

Анализ алгоритма Рабина-Карпа

- Число сравнений зависит от сочетания хэш-функции, текста и образца
- В худшем случае O((N - M)*M)
  - Приведите пример хэш-функции
- В "среднем" O(N) сравнений
  - Приведите сочетание хэш-функции и текста, для которых число сравнений = O(N) и не зависит от образца

Алгоритм Бойера—Мура

- Robert Stephen Boyer Роберт Стивен Бойер р. ?
- J Strother Moore Джей Стротер Мур р. ?
  - Имя из одной буквы!
- Robert S. Boyer, J S. Moore
  A Fast String Searching Algorithm //
  Communications of the Association for
  Computing Machinery, Vol. 20, No. 10, pp. 762-7

Алгоритм Бойера—Мура

- Улучшение наивного поиска
  - Сравнение текста и образца, начиная с q[M – 1] и s[k + M – 1] в обратном порядке
  - Сдвиг образца на расстояние >= 1
    - Таблица сдвигов по стоп-символам d[c] = безопасный сдвиг образца относительно текста при условии, что s[k+M-1] == c и s[k…k+M-1] != q
    - Таблица сдвигов по суффиксам suffix_shift[j] = min сдвиг образца относительно текста, совмещающий внутреннюю часть образца с просмотренным суффиксом

```
s: * * * * * * * * b * * * * *
q:     * * * b * * *
       ----->* * * b * * *
размер сдвига = d[‘b’] – зависит только от q
```

Алгоритм Бойера-Мура со сдвигом по стоп-символам

- Шаг 1
  - Прикладываем левый край образца к левому краю текста, К = 0
  - Заполняем таблицу сдвигов по стоп-символам d
- Шаг 2
  - Проверяем, входит ли образец в текст, начиная с К-й позиции, последовательным сравнением символов образца q[j] с символами текста s[K+j] справа налево, j=M-1…0
- Шаг 3
  - Если имеем M совпадений, то образец в тексте найден – конец работы
  - Если K+M >= N, то образец в тексте не найден – конец работы
  - Иначе K = K+d[s[K+M-1]] и переходим к шагу 2

Алгоритм Бойера-Мура без сдвига по суффиксам

```
int  bm_substring_search(
	const char s[],  int N, const char q[], int M)
{
	int k; // смещение образца по тексту
	int d[256]; // таблица сдвигов
	bm_init(d, q, M);
	for (k = 0; k < N-M; k+=d[s[k+M-1]])
	{
		int j; // смещение по образцу
		for (j = M-1; s[k+j]==q[j]; --j)
			if (j == 0) return k; // нашли
	}
	return -1;		// не нашли
}
```

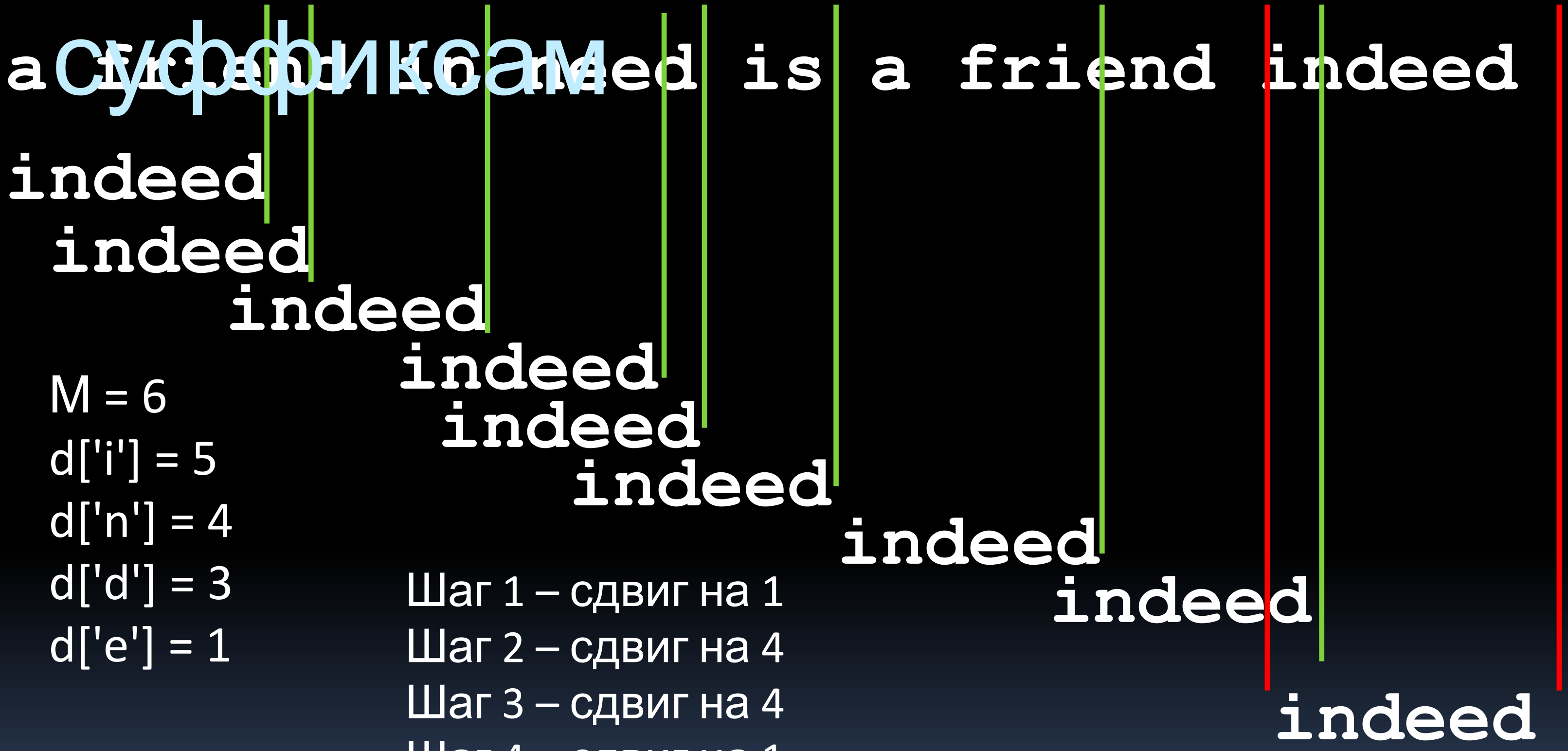
Заполнение таблицы сдвигов по стоп-символам

- Для каждого символа x из образца
  - Если q[M-1] != x (не последний символ), то d[x] есть расстояние от последнего вхождения x в образец до q[M-1]
  - Если q[M-1] == x (последний символ) и x входит в образец >= 2 раз, то d[x] равно расстоянию от предпоследнего вхождения x до q[M-1]
  - Если q[M-1] == x (последний символ) и x входит в образец 1 раз, то d[x] = M

Пример заполнения таблицы сдвигов по стоп-символам

- Для образца q=“abcabeabce” (M = 10)
  - d['a'] = 3
  - d['b'] = 2
  - d['c'] = 1
  - d['e'] = 4
  - d[x] = 10 для x, не входящих в образец

Пример работы алгоритма Бойера – Мура без сдвигов по суффиксам



```
a friend in need is a friend indeed
indeed
 indeed
     indeed
         indeed
          indeed
             indeed
                   indeed
                        indeed
                             indeed
```

M = 6
d['i'] = 5
d['n'] = 4
d['d'] = 3
d['e'] = 1

Шаг 1 – сдвиг на 1
Шаг 2 – сдвиг на 4
Шаг 3 – сдвиг на 4
Шаг 4 – сдвиг на 1
Шаг 5 – сдвиг на 3
Шаг 6 – сдвиг на 6
Шаг 7 – сдвиг на 5
Шаг 8 – сдвиг на 5

Анализ алгоритма Бойера-Мура

- В лучшем случае O(N/M) сравнений
  - Если последний символ образца всегда попадает на символ текста, не входящий в образец

- В худшем случае O((N - M)*M) сравнений
  - Приведите пример текста и образца для худшего случая

Алгоритм Кнута-Морриса-Пратта

- Donald Knuth Дональд Кнут р. 1938
- Воган Пратт р. 1944
- Джеймс Моррис р. 1941
- Knuth, Donald; Morris, James H., jr; Pratt, Vaughan
  "Fast pattern matching in strings"
  SIAM Journal on Computing
  Vol 6 (2), pp. 323–350, 1977

Алгоритм Кнута-Морриса-Пратта

- Улучшение наивного поиска
  - Каждый символ текста участвует в сравнении <= одного раза
  - Сдвиг выбирается с учётом того, какой именно префикс образца совпал с префиксом текста в окне просмотра

Алгоритм Кнута-Морриса-Пратта

- На сколько позиций можно сдвинуть q относительно s, не пропустив вхождений q в s, если до позиций i и j они совпадают, а в i и j различаются?

```
   0   k         i           N-1
s: b a a b a b a b a c a b a t
q:     a b a b a c a
       0         j M-1
```

Префикс-функция КМП

- Префикс-функция prefix(q, j) строки q
  prefix(q,j) = max { x | q[0..x] = q[j-x..j], x < j }
  prefix(q,0) = 0

- Свойства префикс-функции
  - prefix(q,j) = длина самого длинного префикса строки q[0..j], который != q[0..j] и является суффиксом q[0..j]
  - j-prefix(q,j)+1 = размер безопасного сдвига образца, если q[0..j] совпал с текстом в окне просмотра
  - prefix(q,j) = число сравнений, которые можно не делать после такого сдвига окна просмотра

Префикс-функция КМП

- Пример 1

```
j          0  1  2  3  4  5  6
q[j]           a  b  a  b  a  c  a
j-prefix(q,j)+1    1  2  2  2  2  6  6   prefix(q,j)
0  0  1  2  3  0  1
```

- Пример 2

```
j          0  1  2  3  4  5  6
q[j]           b  a  a  a  a  a  a
j-prefix(q,j)+1    1  2  3  4  5  6  7   prefix(q,j)
0  0  0  0  0  0  0
```

Алгоритм Кнута-Морриса-Пратта

- Шаг 1
  - Прикладываем левый край образца к левому краю окна просмотра, К = 0, j = 0
  - Вычисляем префикс-функцию образца
- Шаг 2
  - Проверяем, входит ли образец в текст, начиная с К-й позиции, последовательным сравнением символов образца q[j] с символами текста s[K+j] слева направо, j=j…M-1
- Шаг 3
  - Если имеем М совпадений, то образец в тексте найден – конец работы
  - Если K+M >= N, то образец в тексте не найден – конец работы
  - Иначе К = К+j-prefix[j-1], j = prefix[j-1]+1 и переходим к шагу 2

Алгоритм Кнута-Морриса-Пратта

```
int  kmp_substring_search(
      const char s[],  int N, const char q[], int M)
{
      int k = 0; // смещение образца по тексту
      int j = 0; // смещение по образцу
      int p[M+1], prefix = p+1; // таблица сдвигов, C99
      kmp_init(prefix, q, M);
      for (k = 0; k < N-M; k+=j-prefix[j-1])
      {
             for (j = j; s[k+j]==q[j] && j < M; ++j)
                    if (j == M) return k; // нашли
             j = prefix[j-1]+1;
      }
      return -1;          // не нашли
}
```

Алгоритм Кнута-Морриса-Пратта

- В худшем случае O(N) сравнений без учета построения префикс-функции
  - Почему каждый символ текста участвует в сравнении <= 1 раз?
- Опишите работу алгоритма КМП для текста «аааааа...abaaaaaa» и образца «baaaaaa»
  - В чем отличие от работы алгоритма БМ?

Заключение

- Поиск в массивах и списках
  - Линейный поиск
    - списки, массивы, линейная сложность
  - Бинарный поиск
    - упорядоч. массивы, логарифмическая сложность
- Поиск подстроки
  - Наивный поиск подстроки
    - O(N) … O(M*N)
  - Алгоритм Рабина-Карпа
    - O(N) … O(M*N)
  - Алгоритм Бойера-Мура
    - O(N/M) … O(M*N)
  - Алгоритм Кнута-Мориса-Пратта
    - O(N) … O(N+M)

При первом входе в цикл индексы указывают на начала строк и $Eq(i,j) = Eq(1, 1)$, очевидно, истинно.

На каждом проходе цикла указатель i сдвигается на одну позицию строки вперед без возвратов.

Пока очередные символы совпадают, внутренний цикл не выполняется и j просто увеличивается синхронно с i, что обеспечивает сохранение условия

$$Eq(i\text{нов}, j\text{нов}) = Eq(i+1, j+1)$$

без сдвига образеца относительно строки.

При несовпадении очередных символов надо сдвинуть образец так, чтобы некоторый dj-префикс q продолжал совпадать с dj-суффиксом просмотренной строки $s\ [1\ ..\ i]$, тем самым сохраняя инвариант $Eq\ (i_{нов}, j_{нов}) = Eq\ (i + 1, dj + 1)$ для следующей итерации цикла.

Изменение соответствия позиций с (i, j) на $(i+1, dj+1)$ означает сдвиг q относительно s на $D = j - dj > 0$ позиций вперед.
Отсюда $dj < j$. Если таких dj-префиксов можно указать несколько, надо выбрать из них наибольший по длине, чтобы сдвиг D был кратчайшим.
Если таких префиксов нет, возьмем $dj = 0$, так как $Eq(i+1, 1)$ всегда истинно. Это соответствует сдвигу образеца на $D=j$, к позиции $s[i+1]$; т. е. следующее сравнение начнется со следующей непрочитанной позиции
строки, имея «нулевую историю» совпадений.

До сдвига $\mathit{pref}\,(q, j-1)$ совпадает с $\mathit{suff}\,(\mathit{pref}\,(s, i-1), dj-1)$.
Чтобы сдвиг образеца на $D=j-dj$ был перспективен, префикс
$\mathit{pref}\,(q, j-D-1) = \mathit{pref}\,(q, dj-1)$ должен совпадать с суффиксом
$\mathit{suff}\,(\mathit{pref}\,(s, i-1), dj-1)$, с которым до сдвига совпадал
$\mathit{suff}\,(\mathit{pref}\,(q, j-1), dj-1)$.
Отсюда $\mathit{pref}(q, dj-1) = \mathit{suff}\,(\mathit{pref}\,(q, j-1), dj-1)$, т. е.

$$q[1...dj-1] = q[j-dj+1\ ...\ j-1]. \qquad (7)$$

Это условие необходимо для перспективности сдвига на $D = j - dj$,
но еще не достаточно;
из сравнения нам еще известно, что $s[i]$ не совпадает с $q[j]$.
Поэтому если $q[dj] = q[j]$, то сдвиг бесперспективен.
Сделаем соответствующее уточнение в формуле (7):

$$q[1...dj-1] = q[j-dj+1\ ...\ j-1] \text{ и } q[dj] \neq q[j] \qquad (8)$$

Добавив теперь условие максимальности длины префикса dj,
выразим зависимость dj от j следующей *префикс-функцией:*

$$d\,[j] = max\{d \mid d < j \text{ и } q\,[1...d-1] = q\,[j-d+1\ ...\ j-1] \text{ и } q\,[d] \neq q\,[j]\}. \qquad (9)$$

Как можно видеть, префикс-функция зависит только от образеца q, но не от строки s, поэтому она может быть вычислена ещё до начала поиска и задана в алгоритме таблицей значений.

Однако зависимость dj от строки все же имеется:
если $q[dj] \neq s[i]$, то сдвиг тоже заведомо бесперспективен.
В этом случае вычисленное $d[j]$ следует отвергнуть и
Так как $j > dj$, все длины перспективных префиксов q
образуют последовательность, убывающую до нуля:
$d[j] > d[d[j]] > d[d[d[j]]] > ... > d[...d[j]...] = 0.$

Выбором подходящего dj, с учетом всего сказанного, занимается внутренний цикл КМП-алгоритма.
Ниже приведены значения префикс-функции и величины сдвига для образца *ababaca* из примера выше: - по формуле (9)

j:	1	2	3	4	5	6	7	
$q[j]$:	a	b	a	b	a	c	a	
$d[j]$:	0	1	0	1	0	4	0	- по фломуле (9)
$D=j-d[j]$:	1	1	3	3	5	2	7	

Пример

$1 \qquad\qquad\qquad i \qquad\qquad\qquad N$

$s\colon b\ a\ \underline{a\ b\ a\ b\ a}\ b\ a\ c\ a\ b\ a\ b$

$\qquad\qquad\qquad\ \wedge$

$q\colon \quad \underline{a\ b\ a}\ b\ a\ c\ a$

$\qquad 1 \qquad {}^\wedge d \quad j\ \ M$

$q\colon j\text{-}d \qquad \underline{a\ b\ a}\ b\ a\ c\ a$

$\qquad\qquad 1 \quad {}^\wedge d \qquad M$

- $Eq(i,j)$: $j = 6, d[j] = 4$
- $pref(q,3) = suff(pref(s,i\text{-}1),3)$
- d-1 = 3 – длина совпадения
- условие (8) выполнено и $q[d] = s[i]$
- сдвиг на j-d = 2 совмещает префикс с суфиксом
- d-префикс совпадает $Eq(i+1,d+1)$
- продолжаем сравнение с $(i+1,d+1)$

Допустим, что для всех позиций k образеца, предшествующих и включая i, $d[k]$ уже вычислены и $d[i] = j+1$.

Это означает, что $pref(q, j) = suff(pref(q, i), j)$.

Сравним $q[i + 1]$ и $q[j + 1]$:

если они равны, то $pref(q, j + 1) = suff(pref(q, i+ 1), j + 1)$,

т. е. $d[i + 1] = j + 2$;

если они не равны, то выберем для испытаний следующий по длине префикс q, являющийся суффиксом $pref(q, i)$, т. е. $d[j]$.

```
int  seek_substring_KMP  (char s[],   char q[]){
int  i, j, N, M; N = strlen(s); M = strlen(q);
int *d =(int*)malloc(M*sizeof(int));/*динамический массив длины M+1*/
/* Вычисление префикс-функции */
i=0;   j=-l;   d[0]=-l;
while   (i  < M-l)   {
while((j>=0) && (q[j]!=q[i]))
    j = d[j];
i++; j++;
if(q[i]==q[j]) d[i]   = d[j];
else d[i]= j;
}
/* поиск */
for(i=0,j=0;(i<=N-l)&&(j<=M-l); i++,j++)
while((j>=0)&&(q[j]!=s[i]))
j   = d[j];
free (d);  /* освобождение памяти массива d */
if (j==M) return i-j;
else /* i==N */  return -1;
}
```

Алгоритм Рабина -- Карпа поиска подстроки

- Майкл Рабин, Ричард Карп 1987
- Уменьшение числа сравнений в наивном поиске подстроки за счёт использования кольцевой хэш-функции (разновидность контрольной суммы)
- Пусть строка и образец состоят из символов алфавита A.
- Каждый символ этого алфавита будем считать d-ичной цифрой,
- где d = □A□. Строку из k символов можно рассматривать как
- запись d-ичного k-значного числа.
- Тогда поиск образца в строке сводится к серии из N – M
- сравнений числа, представляющего образец, с числами,
- представляющими подстроки s длины M.
- Cравнение чисел может быть выполнено за время,
- пропорциональное M, и тогда эффективность поиска будет
- O(N + M) .
- Для начала предположим, что A = {0,1, ..., 9}. Число, десятичной
- записью которого является образец q, обозначим через tq.
- Аналогично, обозначим через tk число, десятичной записью
- которого является подстрока s[k ... k + M – 1].

- Подстрока s[k ... k + M – 1] совпадает с образцом q тогда и
- Только тогда, когда tq = tk .

- По схеме Горнера значения tq и t1 можно вычислить за время, пропорциональное M
- Временно забудем о том, что вычисления могут привести к очень большим числам.
- Из tk (1 < k <= N – M) за константное время можно вычислить tk+1 по схеме Горнера

- Чтобы получить t[k+1] из t[k], надо удалить последнее слагаемое из формулы (10) (т. е. вычесть 10^{M-1}*s[k]), результат умножить на 10 и добавить к нему s[k+M]
- В результате получим следующее рекуррентное соотношение:

$$t_{k+1} = 10 \cdot (t_k - 10^{M-1} \cdot s[k]) + s[k+M].$$

k=2, M=4 s = 1 2 3 4 5 6 7

t_2= 5+10·(4+10·(3+10·2)))= 2345

t_3= 6+10·(5+10·(4+10·3)))= 3456

$2345-10^3 \cdot 2=2345-2000=345$

345·10=3450

3450+6=3456

Вычислив все t_k, мы можем по очереди сравнить их с t_q, определив тем самым совпадение или несовпадение образца q с подстроками $s[k \ldots k + M - 1]$.

Время работы этого алгоритма пропорционально N-M.

До сих пор мы не учитывали того, что числа могут быть слишком велики. С этой трудностью можно справиться следующим образом.

Надо проводить вычисления чисел t_q и t_k и вычисления по формуле (11) по модулю фиксированного числа p.

Тогда все числа не превосходят p и действительно могут быть вычислены за время порядка M.

Обычно в качестве p выбирают простое число, для которого $d \cdot p$ помещается в машинное слово, все вычисления в этом случае упрощаются.

Рекуррентная формула (11) приобретает вид:

где .

Из равенства $t_q \equiv t_k (mod\ p)$ еще не следует, что $t_q = t_k$ и, стало быть, что $q = s[k \dots k + M - 1]$.

В этом случае надо для надежности проверить совпадение образеца и подстроки.

Алгоритм А5:

- **вход**: q - образец, s - строка, M - длина образеца, N - длина строки,

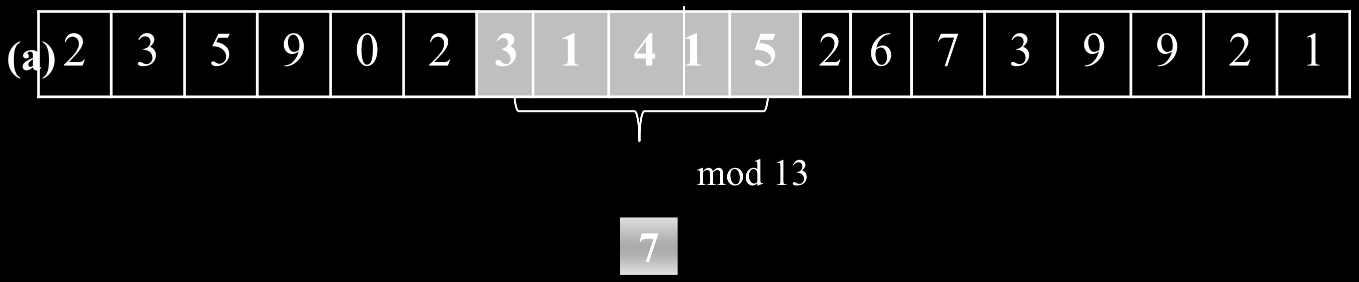
$M < N$, d - число символов в алфавите.

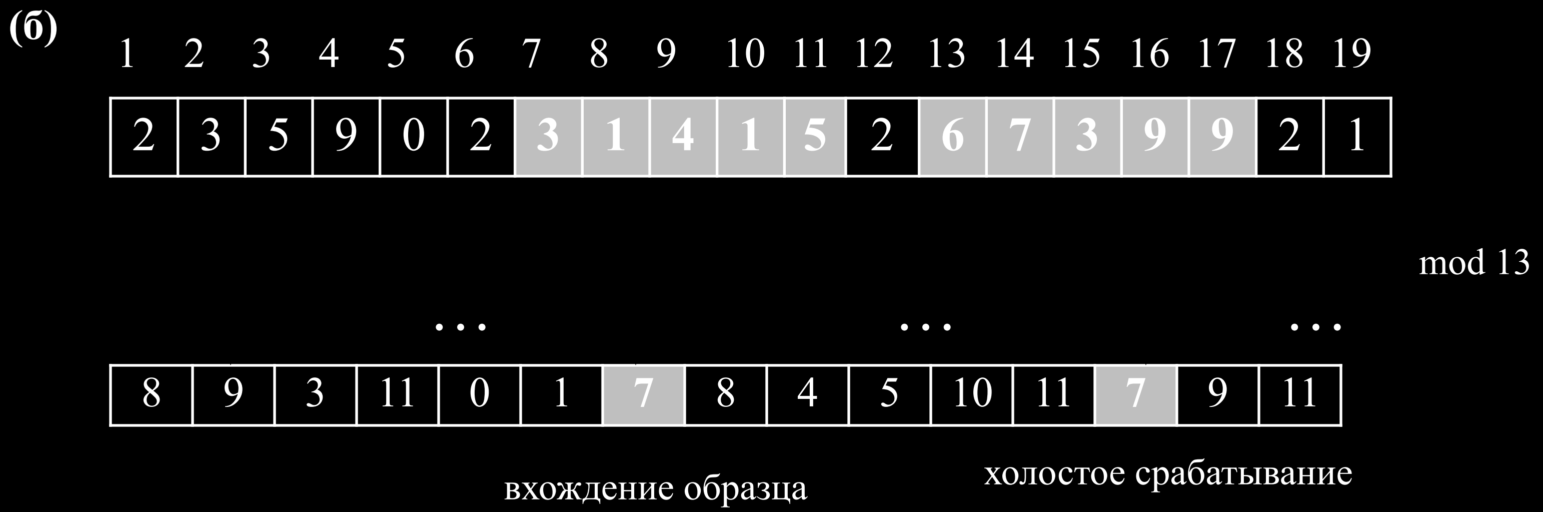
```
По схеме Горнера вычислить числа $t_1$ и $t_k$ по модулю $p$;
цикл по $k$ от 1 до $N – M + 1$
     если $t_q = t_k$ то
       сравнить образец $q$ с подстрокой $s[k … k + M – 1]$;
       если они совпадают, то
             выдать $k$ - результат сравнения;
по формуле (12) вычислить $t_{k+1}$
конец цикла
```

- **выход**: k - позиция начала вхождения образеца в строку.

```
/* $d$ - число символов в алфавите */
/* $p$ - число, по модулю которого производятся вычисления */
/* возвращает смещение вхождения $q$ в $s$ относительно начала строки */
```

```
int Robin_Carp_Matcher(char s[], char q[], int d, int p) {
      int i, h, k, M, N, t_q, t_k;
     N =  strlen(s);  M = strlen(q);
/* вычисление h=(dM-l)mod p */
     h=l;  for(i=l; i<M; i++)
      h=(h*d)%p;
/* вычисление tl и tq по  схеме Горнера */
     t_q = 0; t_k = 0;
     for(i=0; i<M; i++){
      t_q = (d*t_q + q[i])% p;
      t_k = (d*t_k + s[i])% p;
     }
/* сравнение образеца с подстроками и вычисления по формуле (12)*/
     for (k=0; k<=N-M; k++) {
      if (t_q==t_k) {
            for (i=0;(i<M)&&(q[i]==s[k+i]); i++);
            if (i==M) return k;
      }
      if (k<N-M) { /* вычисления по формуле (8) */
            t_k = (d*(t_k-s[k]*h)+ s[k+M])% p;
            if   (t_k < 0)  t_k + = p;
      }
     }
return -1;
}
```

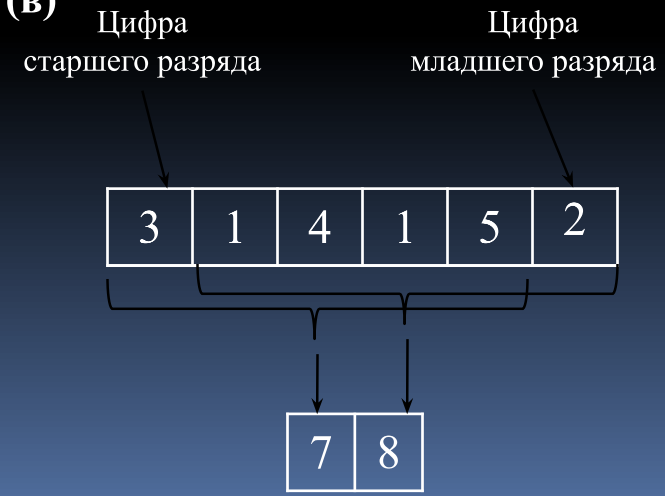




вхождение образца

холостое срабатывание

(в)



Цифра старшего разряда

Цифра младшего разряда

Цифра старшего разряда

Цифра младшего разряда

$$14152 \equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13}$$

$$\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}$$

$$\equiv 8 \pmod{13}$$

Реализация алгоритма Бойера-Мура

```
int  seek_substring_BM(unsigned char s[], unsigned char q[])
{     int d[256];
   int i, j, k, N, M;
   N =  strlen(s);
   M =  strlen(q);
  /* построение d */
   for (i=0; i<256; i++)
   d[i]=M;           /* изначально M во всех позициях */
   for (i=0; i<M-1; i++) /* M - i - 1 - расстояние до конца d */
     d[(unsigned char)q[i]]= M-i-1;/* кроме последнего символа*/
/* поиск */
   i= M-1;
   do {
   j = M-1; /* сравнение строки и образеца */
   k = i;   /* j - по образецу, k - по строке */
   while ((j >= 0) && (q[j] == s[k])) {
      k--; j--;
   }
   if (j < 0) return k+1; /* образец просмотрен полностью */
   i+=d[(unsigned)s[i]];/*сдвиг на расстояние d[s[i]]вправо*/
   } while (i < N);
   return -1;
}
```

Алгоритм Бойера-Мура

Будем последовательно сравнивать образец *q* с подстроками *s*[*i* – *M* + 1..*i*] (в начале *i* = *M*).

Введем два рабочих индекса: *j* = *M*, *M* – 1, … , 1 — пробегающий символы образеца, *k* = *i*, … ,*i* – *M*+1 — пробегающий подстроку.

Оба индекса синхронно уменьшаются на каждом шаге.

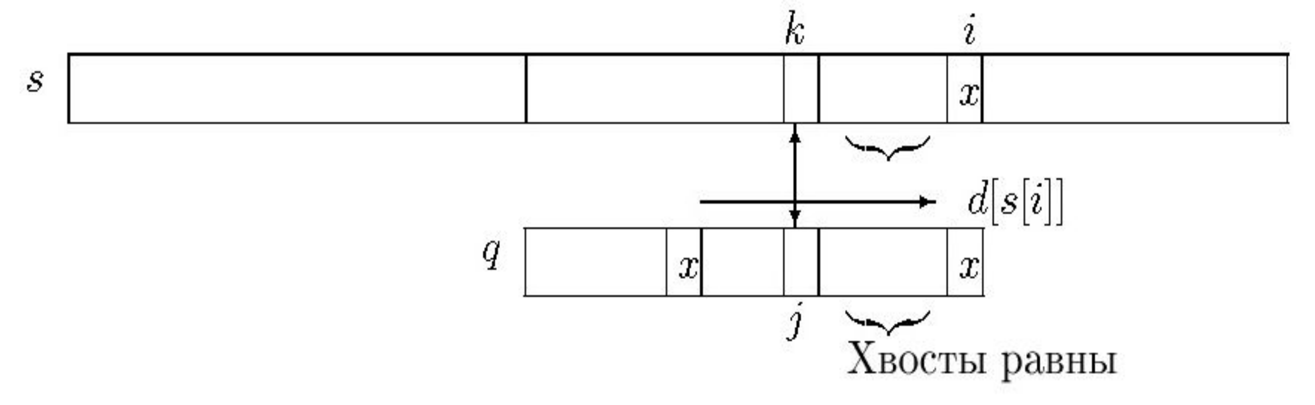
Если все символы *q* совпадают с подстрокой (т. е. *j* доходит до 0), то образец *q* считается найденным в *s* с позиции *k* (*k* = *i* – *M*+1).

Если *q*[*j*]≠*s*[*k*] и *k* = *i*, т. е. расхождение случилось сразу же, в последних позициях, то *q* можно сдвинуть вправо так, чтобы последнее вхождение символа *s*[*i*] в *q* совместилось с *s*[*i*].

Если *q*[*j*] ≠ *s*[*k*] и *k* < *i*. т. е. последние символы совпали, то *q* сдвинется так, чтобы предпоследнее вхождение *s*[*i*] в *q* совместилось с *s*[*i*].

В обоих случаях величина сдвига равна *d*[*s*[*i*]], по построению.

В частности, если *s*[*i*] вообще не встречается в *q*, то смещение происходит сразу на полную длину образеца *M*.



Здесь j = 6 символов строки, следующих за позицией k, уже известны, поэтому можно, не выполняя сравнений, установить, что некоторые последующие сдвиги образца заведомо бесперспективны.
Например, сдвиг на 1 позицию бесперспективен, так как при этом $q[1]$ ='*a*' сравнится с уже известным s[k+1] ='*b*' и совпадения не будет.
А вот сдвиг на 2 позиции сразу отвергнуть нельзя: $q[1...4]$ совпадает с уже известной подстрокой $s[k+2 \ldots k+5]$.
Совпадут ли остальные M - 4 символа, станет известно только при рассмотрении последующих символов s, причем сравнение можно начинать сразу с 5-й позиции образца. Таким образом, при неудаче очередного сравнения надо сдвинуть образец вперед так, чтобы его начало совпало с уже прочитанными символами строки. Если таких сдвигов можно указать несколько, следует выбрать кратчайший из них.

КМП-алгоритм (Кнут, Моррис, Пратт)

Алгоритм А4:

- **вход:** q - образец, s - строка,

```
  $M$ - длина образеца, $N$ - длина строки, $M < N$.
  $i := l; j := 1;$
  пока $(i \le N)$ и $(j \le M)$ цикл
    пока $(j > 0)$ и $s([i] \ne q[j])$ цикл
        $j := dj;$        /*$0 \le dj < j$ */
    конец цикла;
    $i := i + 1;\ j := j + 1;$
  конец цикла;
```

- **выход:** если $j > M$ то образец q найден в позиции $i - M$;

иначе /* $i > N$ */ образец q не найден.

Индекс-указатель *i* пробегает строку *s* без возвратов (что обеспечивает линейность времени работы алгоритма). Индекс *j* синхронно пробегает образец *q*, однако может возвращаться к некоторым предыдущим позициям *dj*. которые будут выбираться так, чтобы обеспечить на всем протяжении алгоритма инвариантность следующего условия *Eq*(*i*,*j*): «все символы образеца, предшествующие позиции *j*, совпадают с таким *же* числом символов строки, предшествующих позиции *i* »:

$Eq(i,j)$:
$1 \le i \le N+1$ и $1 \le j \le M+1$ и $pref(q,j-1)=suff(pref(s,i-1),j-1)$,

где $pref(str,k)=str[1...k]$ – k-префикс str

$suff(str,k)=str[l-k+1...l]$ – k-суффикс $str[]$ (l – длина str)

(пример $str[i..i-1]$=“ ”)

Истинность условия $Eq(i, M+1)$ означает,
что образец q входит в s начиная с позиции $i–M$.
Выполнение условия $Eq(N+1, j)$ при $j < M$ означает,
что образеца в строке нет.