Поиск строк. Обозначения и базовые понятия

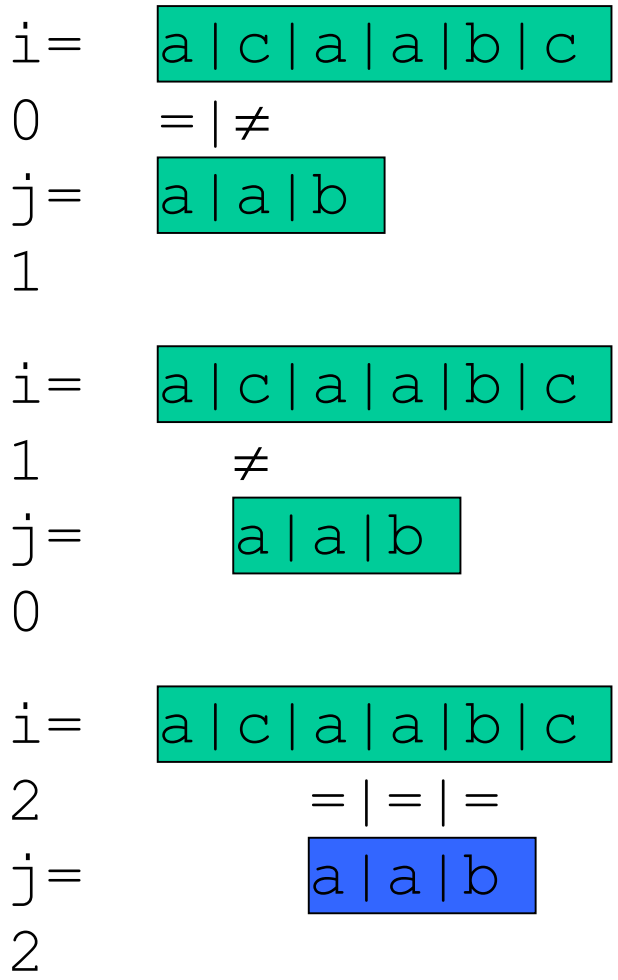
- T – *текст* (строка), в котором осуществляется поиск. N – длина текста.
- P – *образец* (искомая подстрока). M – длина образца.
- $\sum$ – *алфавит* (множество символов, образующих текст и образец).
- $\sum^*$ – множество всех возможных строк алфавита $\sum$, включая пустую строку ε.
- Строка z – *префикс* (начало) строки x, если $x = zy$, для некоторого $y \in \sum^*$.
- Строка z – *суффикс* (конец) строки x, если $x = yz$, для некоторого $y \in \sum^*$.

Простой («наивный», «грубой силы») алгоритм

```
int SubStringNaive(char T[], int N, char P[], int M) {
int i,j;
for(i = 0; i < N - M; i++) {
  for(j = 0; j < M && P[j] == T[i+j]; j++);
  if (j == M) return i;
  }
}

// O(N*M)      // O(N)
```

Простой алгоритм. Иллюстрация



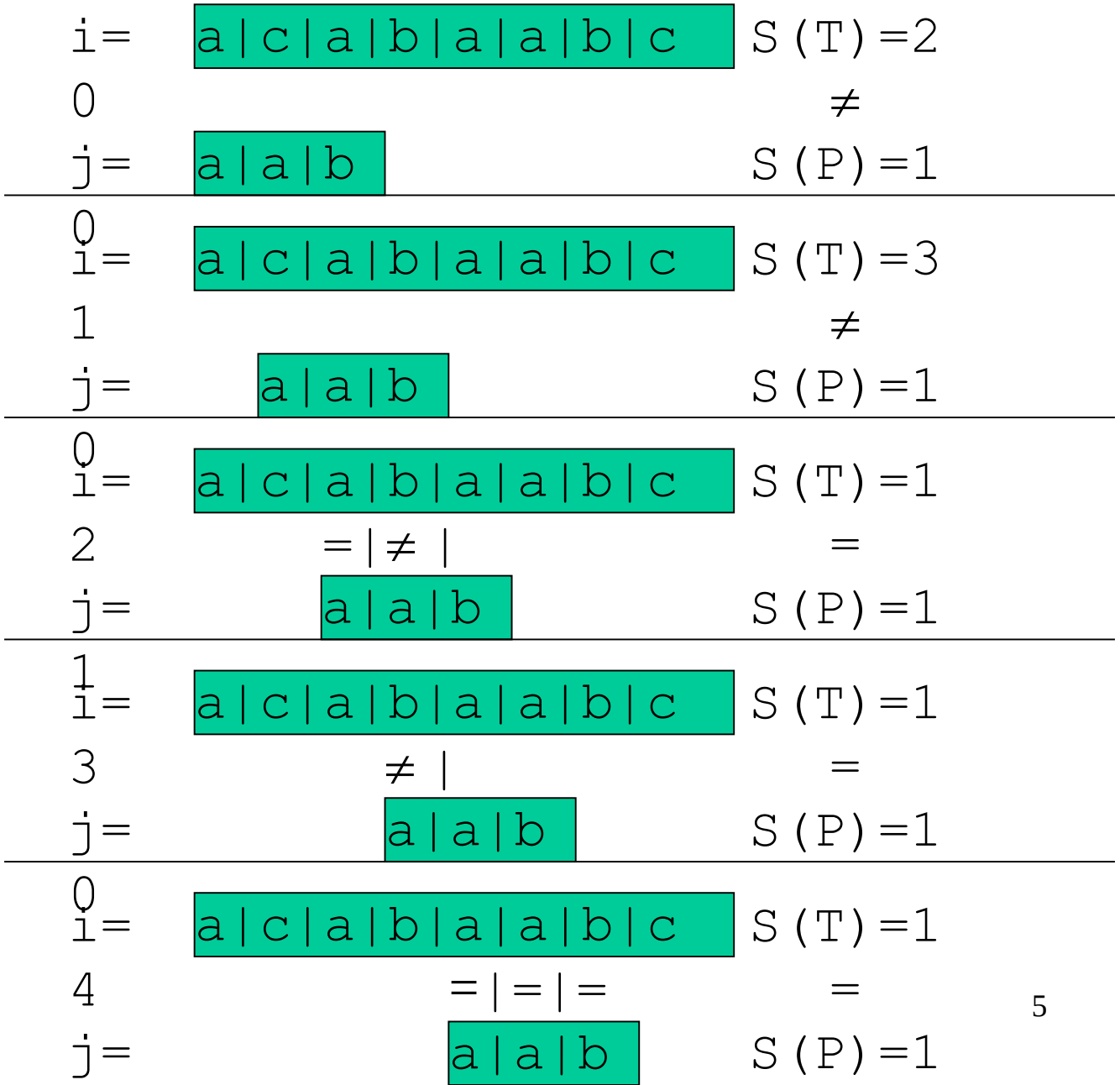
Поиск строк. Алгоритм Рабина-Карпа

Идея этого алгоритма состоит в том, чтобы сократить количество полных проверок идентичности подстрок текста и образца, путем сопоставления им значений, вычисляемых за постоянное время с помощью некоторой функции. Асимптотическая сложность работы такого алгоритма равна $O(N + M)$ в среднем и, к сожалению, $O((N - M + 1)M)$ в худшем. Стоит заметить, что вероятность худшего случая мала.

Алгоритм Рабина-Карпа. Иллюстрация

Пусть a=0,b=1,
c=2, тогда
S(P)=∑P[i]=1,

S(T[j,j+M-1])=
S(T[j-1,j+M-2])-
T[j-1]+T[j+M-1]



Алгоритм Рабина-Карпа. Хеширование. Схема Горнера

Как эффективно сопоставлять число фрагменту текста?

Сопоставление некоторым данным (ключу, строке, файлу и т. п.) числа (или возможно другого «простого» объекта) называют хешированием.

Предположим, что строка – это последовательность цифр в 10-й системе исчисления. Каким образом можно быстро вычислять соответствующее число?

$$S = T[i]\cdot 10^{M-1} + T[i+1]\cdot 10^{M-2} \ \ldots + T[i+M-1] \qquad (*)$$

Асимптотическая сложность вычисления формулы (*) – *O(M)*

$$S = (\ldots((T[i]\cdot 10 + T[i+1])\cdot 10 + T[i+2]) \ + \ \ldots \ T[i+M-1] \qquad (**)$$

Вычисление хеш-значения по формуле (**) позволяет вычислять ***очередное*** значение за время, которое не зависит от длины образца

$$S_n = (S_{n-1} - T[i-1]\cdot 10^{M-1})\cdot 10 + T[i+M-1]$$

Поиск строк. Алгоритм Рабина-Карпа

```
#define S 119
#define d 256
int SubStringRK(char T[], int N, char P[], int M) {
int i, j, h = 1, p = 0, t = 0;
for (i = 0; i < M - 1; i++)
  h = (d * h) % S;
for (i = 0; i < M; i++)
  p = (d * p + P[i]) % S,   t = (d * t + T[i]) % S;
for (i = 0; i < N - M; i++) {
  if (p == t) {
    for (j = 0; j < M && P[j] == T[i + j]; j++);
    if (j == M) return i;
    }
  t = (d * (t - h * T[i]) + T[i + M]) % S;
  t = (t < 0) ? t + S : t;
  }
}
```

Поиск строк. Алгоритм «Сдвиг-ИЛИ»

Для образцов небольшой длины (длина не превышает разрядности слова вычислительной системы) достаточно эффективным является именно этот алгоритм. Этот алгоритм использует поразрядные операции. Пусть битовый вектор R имеет размер M. Вектор R_j – это значение вектора R после того, обработан символ $T[j]$. Он содержит информацию обо всех вхождениях префикса образца P, которые оканчиваются в позиции j в тексте T для $i \in [0, M-1]$:

$$R_j[i] = \begin{cases} 0, P[0,i] = T[j-i, j] \\ 1, P[0,i] \neq T[j-i, j] \end{cases}$$

Поиск строк. Алгоритм «Сдвиг-ИЛИ»

Рекурсивно можно вычислить R_{j+1} зная R_j следующим образом. Для каждого $R_j[i] = 0$:

$$R_{j+1}[i] = \begin{cases} 0, P[i+1] = T[j+1] \\ 1, P[i+1] \neq T[j+1] \end{cases}$$

и

$$R_{j+1}[0] = \begin{cases} 0, P[0] = T[j+1] \\ 1, P[0] \neq T[j+1] \end{cases}$$

Если $R_{j+1}[M-1] = 0$, то образец найден. Для быстрого вычисления R необходимо для каждого символа алфавита C вычислить вспомогательный вектор S_C, (размера M) который будет содержать информацию о позиции данного символа в образце P: $S_C[i] = 0$, если $P[i] = C$, при $i \in [0, M-1]$. Зная S_C вычислять R можно за две операции: R_{j+1} = СДВИГ_ВЛЕВО (R_j) ИЛИ $S_{CT[j+1]}$. Асимптотическая сложность предварительной подготовки O($M + D$), а собственно поиска – O(N).

Алгоритм «Сдвиг-ИЛИ». Иллюстрация

Пусть $\Sigma=\{0,1,\ldots,9\}$, тогда $|SC|=10$.

Для образца P="1230239"

SC=
1110111 | 0
1111110 | 1
1101101 | 2
1011011 | 3
1111111 | 4
1111111 | 5
1111111 | 6
1111111 | 7
1111111 | 8
0111111 | 9

T 1|2|3|1|2|3|0|2|3|9

R 1|1|1|1|1|1|1

i=0 R = R<<1 | SC[1]
R 1|1|1|1|1|1|0

i=1 R = R<<1 | SC[2]
R 1|1|1|1|1|0|1

i=2 R = R<<1 | SC[3]
R 1|1|1|1|0|1|1

i=3 R = R<<1 | SC[1]
R 1|1|1|1|1|1|0

i=4 R = R<<1 | SC[2]
R 1|1|1|1|1|0|1

i=5 R = R<<1 | SC[3]
R 1|1|1|1|0|1|1

• • •

i=9 R = R<<1 | SC[9]
R 0|1|1|1|1|1|1

Поиск строк. Алгоритм «Сдвиг-ИЛИ»

```
int SearchStringSO(char * T, int N, char * P, int M) {
unsigned int SC[256], R = 0xFFFFFFFF, Mask = 0;
int i;
for (i = 0; i < 256; i++) // O(D)
  SC[i] = 0xFFFFFFFF; //SC[i] = 11111111111111..1
for (i = 0; i < M; i++) // O(M)
  SC[(unsigned int)P[i]] &= ~(1<<i);//  11..1011..11
Mask = ~(1 << (M-1));
for (i = 0; i < N; i++) { O(N)
  R = (R << 1) | SC[(unsigned int)T[i]];
  if (Mask >= R) return (i - M + 1);
  }
return -1;
}
// O(D + M + N) => O(D + N)
```

Поиск строк. Алгоритм Кнута-Морриса-Пратта

Этот алгоритм основывается на предварительном создании префикс-функции (массива), ассоциированной с образцом и несущей информацию о том, где в образце повторно встречаются его же различные префиксы. Использование этой информации позволяет избежать проверки заведомо недопустимых сдвигов.

Если P – образец длиной M, то

префикс-функция f: $\{0, 1, 2, \ldots, M-1\} \rightarrow \{0, 1, 2, \ldots, M-1\}$

определяется следующим образом:

$f[q] = \max \{k : k < q$ и префикс P_k является суффиксом $P_q\}$,

т.е. $f[q]$ – длина (индекс крайнего правого символа) *наибольшего* префикса P, являющегося *собственным* суффиксом P_q. Время предварительной подготовки пропорционально длине образца, время поиска составляет $O(N + M)$ как в худшем случае, так и в среднем.

Алгоритм хорош для бинарных файлов с регулярной структурой и менее эффективен для поиска в текстовых файлах.

Поиск строк. Алгоритм Кнута-Морриса-Пратта

```
-1 -1 -1 0 1 2 -1 0 1
```

```
3412312301231239 12
  1231239 12
3412312301231239 12
    123123912
```

i = 8, j = 5,(2,-1)

```
-1 -1 0 1 2 3 -1
```

```
34121211212122
  1212122
34121211212122
    12122122
34121211212122
      1212122
```

i = 7, j = 4,(2,0,-1)

Поиск строк. Алгоритм Кнута-Морриса-Пратта

```
int SearchStringKMP(char * T, int N, char * P, int M) {
int * Next, i, j;
Next=(int*)malloc(M*sizeof(int));
for(Next[0] = j = -1, i = 1; i < M; i++) {
  //Префикс-функция (индексы)
  for( ; j > -1 && P[j+1] != P[i]; j = Next[j]);
  if (P[j+1] == P[i]) j++;
  Next[i] = j;
  }
for(j = -1, i = 0; i < N; i++) {
  //пока есть совпадения i и j растут одновременно
  for( ; j > -1 && P[j+1] != T[i]; j = Next[j]);//сдвиги
  if (P[j+1] == T[i]) j++;
  if( j == M-1 )  {free(Next);  return i-j;}
  }
free(Next);
return -1;
}
```

Поиск строк. Алгоритм Бойера-Мура

Для длинных образцов, состоящих из достаточно большого алфавита, возможно одним из лучших алгоритмов является алгоритм Бойера-Мура. Его структура аналогична структуре самого простого алгоритма, но предварительно вычисляются две вспомогательные функции (массива), значения которых используются для вычислений очередных значений индекса при перемещении по тексту. Кроме того, в отличие от простого алгоритма сопоставление образца с текстом происходит справа налево.

Поиск строк. Алгоритм Бойера-Мура

Две вспомогательные функции представляют собой функции: функцию стоп-символа и функцию безопасного суффикса. Их помощь заключается в следующем. Пусть при сравнении на некотором расстоянии образца с текстом выяснилось, что крайняя правая часть совпадает, а затем следует (налево) символ не соответствующий образцу. Функция стоп-символа предлагает сдвинуть образец вправо на такое расстояние, чтобы стоп-символ в тексте оказался напротив крайнего правого вхождения этого символа в образец. Если стоп символа в образце вообще нет, то образец необходимо полностью переместить за стоп-символ. Если стоп символ встречается в образце правее стоп-символа в тексте, то данная функция ничего хорошего не дает, так как, следуя ей, надо было бы сдвинуть образец влево!

С помощью функции безопасного сдвига можно сдвинуть образец вправо настолько, чтобы ближайшее вхождение (правее текущего) безопасного суффикса в образец оказалось напротив безопасного суффикса в тексте.

Алгоритм Бойера-Мура выбирает больший из двух возможных сдвигов. В худшем случае время работы этого алгоритма равно $O((N - M + 1)M + D)$ (D – мощность алфавита), однако на практике он оказывается одним из самых эффективных.

Поиск строк. Алгоритм Бойера-Мура

LO	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
	-1	7	8	5	-1	-1	-1	-1	-1	6
	0	1	2	3	4	5	6	7	8	9
GS	7	7	7	7	7	7	4	4	1	

```
34123123 0123123912
  123123 9 12
34123123 0123123912
         123123912
```

a) i = 2, j = 6

эвристика стоп символа: 6-(-1)=7

эвристика безопасного суффикса: 4

Поиск строк. Алгоритм Бойера-Мура

LO	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
	-1	4	6	-1	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9
GS	7	7	7	7	7	1	1			

```
3412121112122
   1212122

3412121112122
    1212122

3412121112122
     1212122
```

а) i = 3, j = 5

эвристика стоп символа: 5-(4)=1

эвристика безопасного суффикса: 1

б) i = 4, j = 6

эвристика стоп символа: 6-(4)=2

эвристика безопасного суффикса: 1

Поиск строк. Алгоритм Бойера-Мура

```
int SearchStringBM(char * T, int N, char * P, int M) {
int i, j, LO[256], * GS = 0, * Next0 = 0, * Next1 = 0; char * P1 = 0;
for(i = 0; i < 256; i++) LO[i] = -1;
for(i = 0; i < M; i++) LO[P[i]] = i;
Next0 = (int*)malloc(M*sizeof(int));
PrefixFunction(Next0,P,M);
Next1 = (int*)malloc(M*sizeof(int));
P1    = (char*)malloc(M*sizeof(char));
for(i = 0; i < M; i++) P1[M-1-i] = P[i];
PrefixFunction(Next1,P1,M);
GS = (int*)malloc(M*sizeof(int));
for(i = 0; i < M; i++)   GS[i] = M - Next0[M-1];
for(i = 0; i < M; i++) {
  j = M - 1 - Next1[i];
  if(GS[j] > i + 1 - Next1[i])  GS[j] = i + 1 - Next1[i];
  }
free(Next0); free(Next1); free(P1);
for(i = 0; i < N - M; i += (GS[j] > j - LO[T[i+j]]) ? GS[j] : j - LO[T[i+j]]) {
  for(j = M-1; j > -1 && P[j] == T[i+j]; j--);
  if(j == -1) {free(GS); return i;} /*i += GS[0]*/
  }
free(GS);
return -1;
}
```

Поиск строк. Алгоритм Бойера-Мура

264126126

LO	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
	-1	5	7	-1	2	-1	8	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9
GS	7	7	7	7	7	3	3	3	1	