```
int main(int argc, char** argv)
{
  char c = 0;
  char* commands = "ads pq"; // key commands: "left,right,rotate,confirm,pause,quit"
  int speed = 2; // sets max moves per row
  int moves_to_go = 2;
  int full = 0; // whether board is full
  init(); // initialize board an tetrominoes


  cur =
  state[
  state[

  system
  printf

  while
    {
      dr

    // process user action
    c = getchar();   // get new action
    if (c == commands[0] && !intersect(cur, state[0]-1, state[1])) state[0]--; // move left
    if (c == commands[1] && !intersect(cur, state[0]+1, state[1])) state[0]++; // move right
    if (c == commands[2] && !intersect(cur->rotated, state[0], state[1])) cur = cur->rotated;
    if (c == commands[3]) moves_to_go=0;

    // scroll down
    if (!moves_to_go--)
      {
        if (intersect(cur,state[0],state[1]+1)) // if tetromino intersected with sth
          {
            cramp_tetromino();
            remove_complete_lines();
            cur = &tetrominoes[rand() % NUM_POSES];
            state[0] = (WIDTH - cur->width)/2;
```

MAC122 - PRINCÍPIOS DE DESENVOLVIMENTO DE ALGORITMOS

Busca de palavra em texto

Qual a saída?

```
1  typedef char* string
2  int func (string s) {
3    int i, n = 0;
4    for (i=0; s[i] != '\0'; i++)
5       if (s[i] == ' ') n++;
6    return n;
7  }
8
9  int main() {
10   char *t = "MAC 122";
11   printf("%d", func(t));
12   return 0;
13 }
```

Qual a saída?

```
1  typedef char* string
2  int func (string s) {
3    int i, n = 0;
4    for (i=0; s[i] != '\0'; i++)
5       if (s[i] == ' ') n++;
6    return n;
7  }
8
9  int main() {
10   char *t = "MAC 122";
11   printf("%d", func(t));
12   return 0;
13 }
```

```
1
```

Revisão: Tabela ISO-8859-1 (latin1)

32		64	@	96	`	162	¢	194	Â	226	â
33	!	65	A	97	a	163	£	195	Ã	227	ã
34	"	66	B	98	b	164	¤	196	Ä	228	ä
35	#	67	C	99	c	165	¥	197	Å	229	å
36	$	68	D	100	d	166	¦	198	Æ	230	æ
37	%	69	E	101	e	167	§	199	Ç	231	ç
38	&	70	F	102	f	168	¨	200	È	232	è
39	'	71	G	103	g	169	©	201	É	233	é
40	(	72	H	104	h	170	ª	202	Ê	234	ê
41	)	73	I	105	i	171	«	203	Ë	235	ë
42	*	74	J	106	j	172	¬	204	Ì	236	ì
43	+	75	K	107	k	173		205	Í	237	í
44	,	76	L	108	l	174	®	206	Î	238	î
45	-	77	M	109	m	175	¯	207	Ï	239	ï
46	.	78	N	110	n	176	°	208	Ð	240	ð
47	/	79	O	111	o	177	±	209	Ñ	241	ñ
48	0	80	P	112	p	178	²	210	Ò	242	ò
49	1	81	Q	113	q	179	³	211	Ó	243	ó
50	2	82	R	114	r	180	´	212	Ô	244	ô
51	3	83	S	115	s	181	µ	213	Õ	245	õ
52	4	84	T	116	t	182	¶	214	Ö	246	ö
53	5	85	U	117	u	183	·	215	×	247	÷
54	6	86	V	118	v	184	¸	216	Ø	248	ø
55	7	87	W	119	w	185	¹	217	Ù	249	ù
56	8	88	X	120	x	186	º	218	Ú	250	ú
57	9	89	Y	121	y	187	»	219	Û	251	û
58	:	90	Z	122	z	188	¼	220	Ü	252	ü
59	;	91	[	123	{	189	½	221	Ý	253	ý
60	<	92	\	124	|	190	¾	222	Þ	254	þ
61	=	93	]	125	}	191	¿	223	ß	255	ÿ
62	>	94	^	126	~	192	À	224	à		
63	?	95	_	161	¡	193	Á	225	á		

Revisão: Strings

`char *s;`
Cadeia de caracteres terminada em `'\0'`

Uma string é armazenada em um vetor e representada por um ponteiro para o primeiro caractere

Tamanho ou comprimento de uma string: número de caracteres menos um (sem contar o `'\0'`)

```
1 /* string de tamanho 6 contendo "MAC122" */
2 char s1[] = {'M', 'A', 'C', '1', '2', '2', '\0', 'P', '1'};
3 /* string de tamanho 5 contendo "M A C" */
4 char s2[] = {'M', ' ', 'A', ' ', 'C', '\0'};
5 /* não é string */
6 char s3[] = {'M', 'A', 'C'};
```

Strings constantes

Representadas por uma sequência de caracteres delimitados por aspas

`(const char *) "string"`

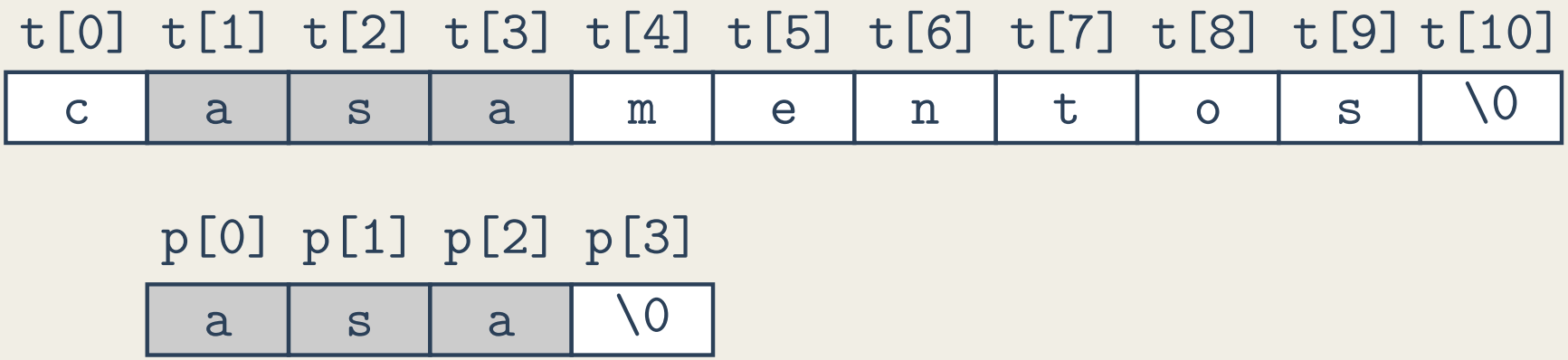
```
1 char *s = "MAC 122";
2 /* s[0] = 'M', s[1] = 'A', s[2] = 'C', s[3] = ' ',
3    s[4] = '1', s[5] = '2', s[6] = '2', s[7] = '\0'  */
```

Revisão: Biblioteca string

`unsigned int strlen(char *s)`
Retorna tamanho de uma string
`string strcpy(char *s, char *t)`
Copia string `t` na string `s`
`int strcmp(char *s, char *t)`
Compara strings de acordo com ordem lexicográfica `int strncmp(char *s, char *t, int n)`
Compara strings de acordo com ordem lexicográfica considerando no máximo `n` caracteres

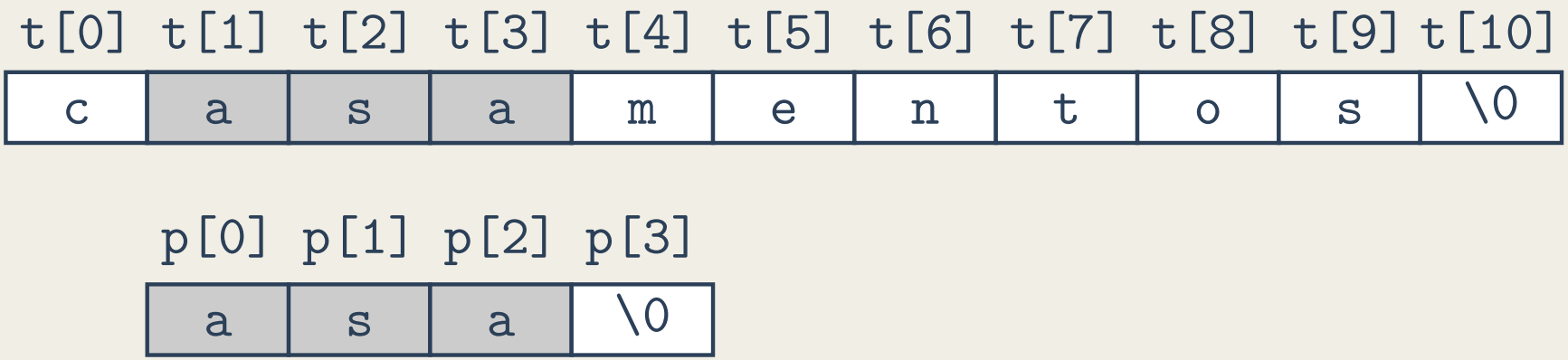
Busca de palavra/padrão em texto

Dadas: strings `p[0..m]` (com `p[m]='\0'`) e `t[0..n]` (com `t[n]='\0'`), encontrar ocorrências de palavra ou padrão `p` em texto `t`



Busca de palavra/padrão em texto

Dadas: strings `p[0..m]` (com `p[m]='\0'`) e `t[0..n]` (com `t[n]='\0'`), encontrar ocorrências de palavra ou padrão `p` em texto `t`



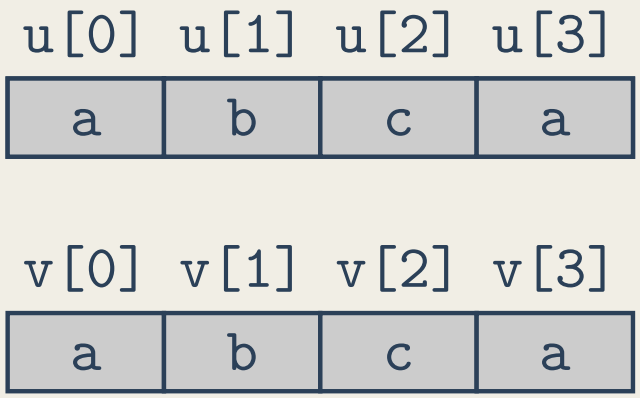
Vamos omitir o caractere de término (`\0`) em diante

Casamento de vetores

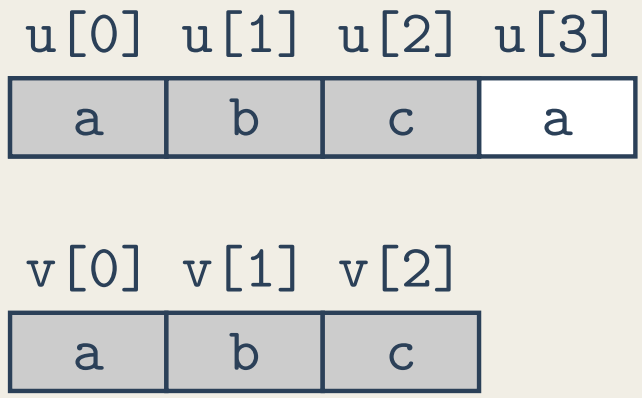
Dois vetores `a[0..m-1]` e `b[0..n-1]` *casam* se `m=n` e

`a[0]=b[0], a[1]=b[1], ..., a[n-1]=b[n-1] .`

Casam:



Não casam:

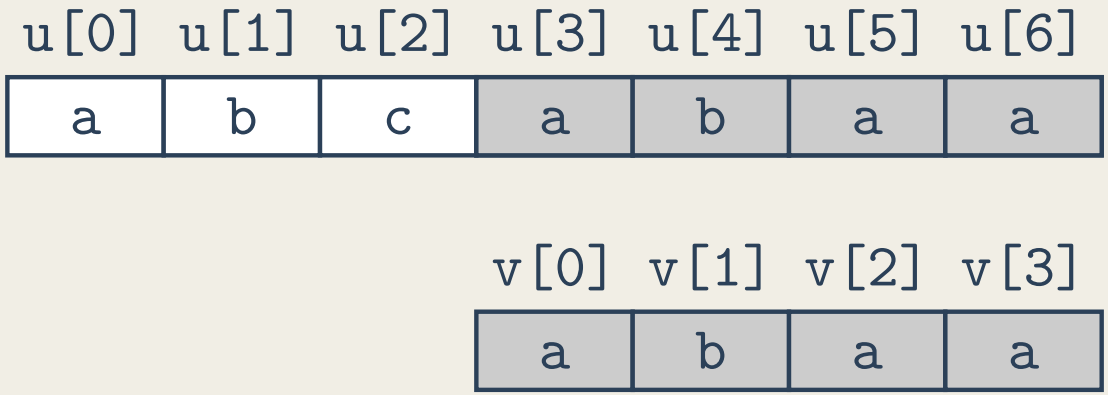


Terminologia

Sufixo

O vetor `a[0..m-1]` é sufixo do vetor `b[0..n-1]`, se `a[0..m-1]` *casa* com `b[n-m..n-1]`.

Note: se `m > n` então `a` não é sufixo de `b`.

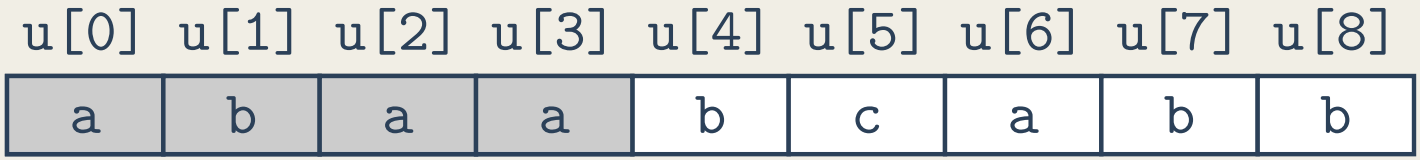


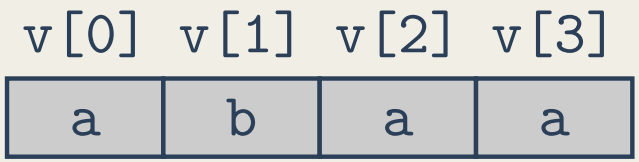
Terminologia

Prefixo

O vetor `a[0..m-1]` é prefixo de um vetor `b[0..n-1]`, se `a[0..m-1]` *casa* com `b[0..m-1]`.

Note: se `m > n` então `a` não é prefixo de `b`.

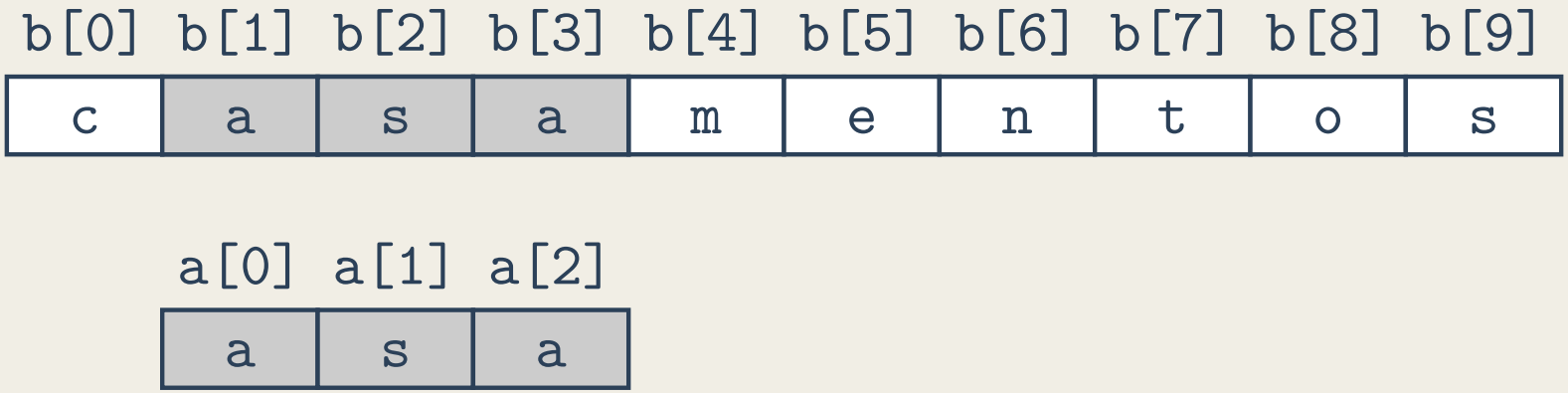




Ocorrência

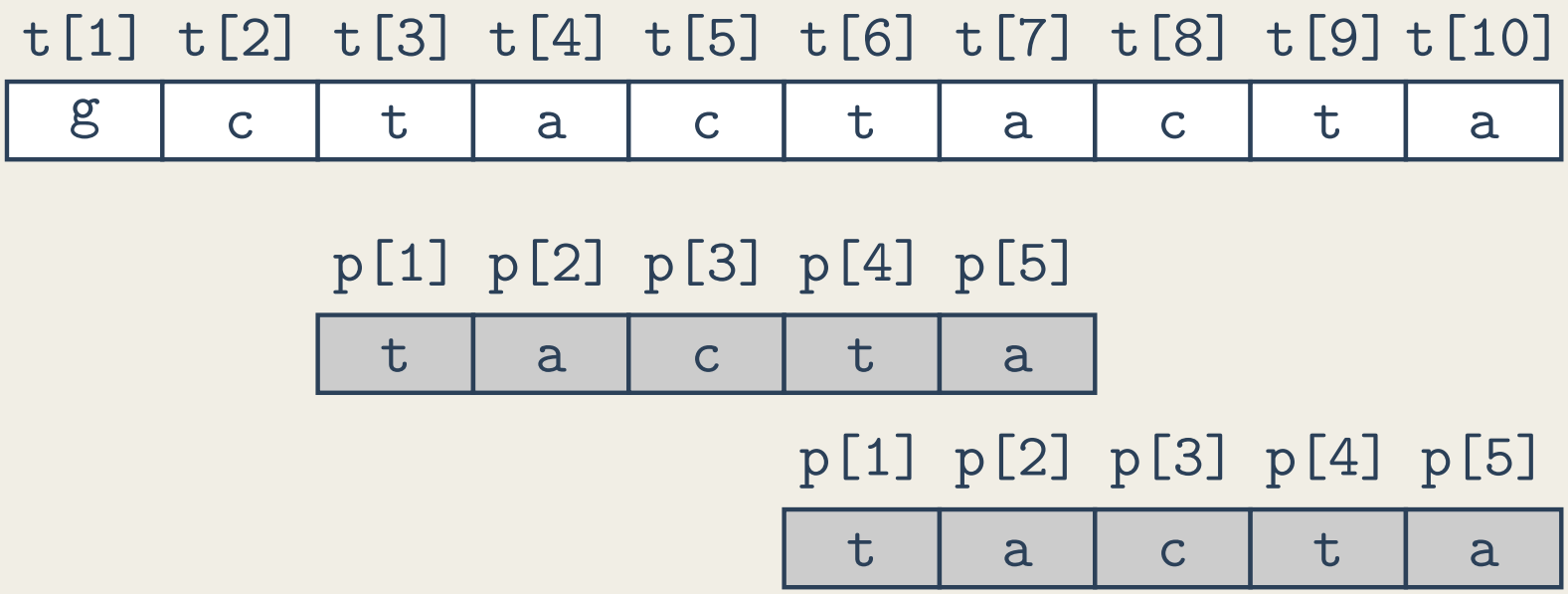
Uma ocorrência de `a[0..m-1]` em `b[0..n-1]` é um valor de `k` tal que `a[0..m-1]` é *prefixo* de `b[k..k+m]`.

Alternativa: Uma ocorrência de `a[0..m-1]` em `b[0..n-1]` é um valor de `k` tal que `a[0..m-1]` é sufixo de `b[0..k-1]`.



Problema computacional

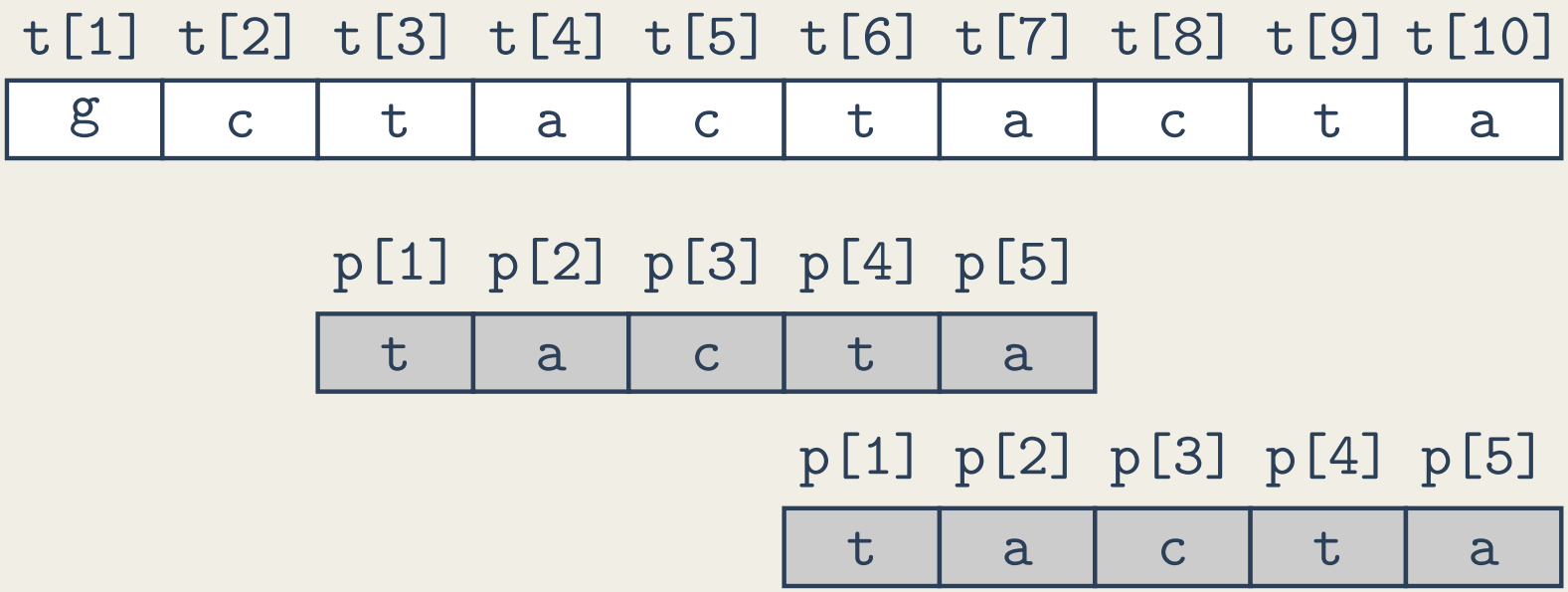
Encontrar as ocorrências de um vetor `p[0..m-1]`, chamado palavra ou padrão, em um vetor `t[0..n-1]`, chamado texto.



Definição simplificada

Problema computacional

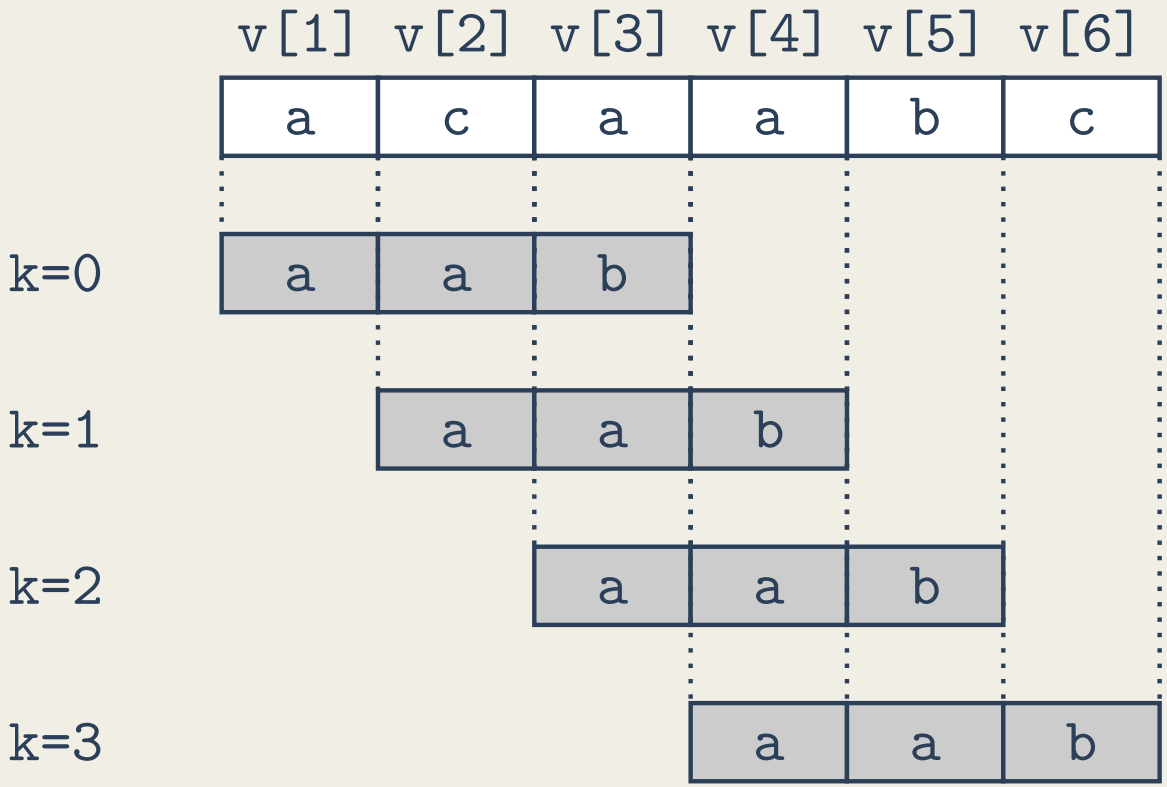
Dados vetores `p[0..m-1]` e `t[0..n-1]` encontrar o número de ocorrências distintas de `p` em `t`.



Algoritmo trivial

Algoritmo trivial

Verificar para `k=0,1,...,n-m` se `p` é prefixo de `t[k..n-1]`



- Verificação pode ser feita da esquerda para direita ou vice-versa

Algoritmo trivial

```
1  // Recebe strings p e t e devolve o numero de ocorrências
       de p em t.
2  int busca_em_texto_trivial (string p, string t)
3  {
4      int i, k, m, n, o = 0;
5      m = strlen(p); n = strlen(t);
6      for (k = 0; k <= n-m; k++) {
7          // p[0..m-1] casa com t[k..k+m]?
8          // comparando da esquerda para direita
9          i = 0;
10         while (i < m && p[i] == t[i+k]) i++;
11         if (i >= m) o++;
12     }
13     return o;
14 }
```

Complexidade de tempo (pior caso)?

Algoritmo trivial

```
1  // Recebe strings p e t e devolve o numero de ocorrências
       de p em t.
2  int busca_em_texto_trivial (string p, string t)
3  {
4      int i, k, m, n, o = 0;
5     m = strlen(p); n = strlen(t);
6      for (k = 0; k <= n-m; k++) {
7          // p[0..m-1] casa com t[k..k+m]?
8          // comparando da esquerda para direita
9          i = 0;
10         while (i < m && p[i] == t[i+k]) i++;
11         if (i >= m) o++;
12     }
13     return o;
14 }
```

Complexidade de tempo (pior caso)? $O(mn)$.

Algoritmo trivial (implementação alternativa)

```
1  // Recebe strings p e t e devolve o numero de ocorrências
       de p em t.
2  int busca_em_texto_trivial2 (string p, string t)
3  {
4      int i, k, m, n, o = 0;
5     m = strlen(p); n = strlen(t);
6      for (k = 0; k <= n-m; k++) {
7          // p[0..m-1] casa com t[k..k+m]?
8          // comparando da direita para esquerda
9          i = m-1;
10         while (i >= 0 && p[i] == t[i+k]) i--;
11         if (i < 0) o++;
12     }
13     return o;
14 }
```

Complexidade de tempo?

Algoritmo trivial (implementação alternativa)

```
1  // Recebe strings p e t e devolve o numero de ocorrências
       de p em t.
2  int busca_em_texto_trivial2 (string p, string t)
3  {
4      int i, k, m, n, o = 0;
5      m = strlen(p); n = strlen(t);
6      for (k = 0; k <= n-m; k++) {
7          // p[0..m-1] casa com t[k..k+m]?
8          // comparando da direita para esquerda
9          i = m-1;
10         while (i >= 0 && p[i] == t[i+k]) i--;
11         if (i < 0) o++;
12     }
13     return o;
14 }
```

Complexidade de tempo? $O(mn)$.

Algoritmo trivial (implementação alternativa 2)

```
1  // Recebe strings p e t e devolve o numero de ocorrências
       de p em t.
2  int busca_em_texto_trivial3 (string p,
3                               string t)
4  {
5      int r, k, o = 0;
6     m = strlen(p); n = strlen(t);
7      for (k = m; k <= n; k++) {
8         r = 1;
9         while (r <= m && p[m-r] == t[k-r]) r++;
10        if (r > m) o++;
11     }
12     return o;
13 }
```

Complexidade de tempo? $O(mn)$.

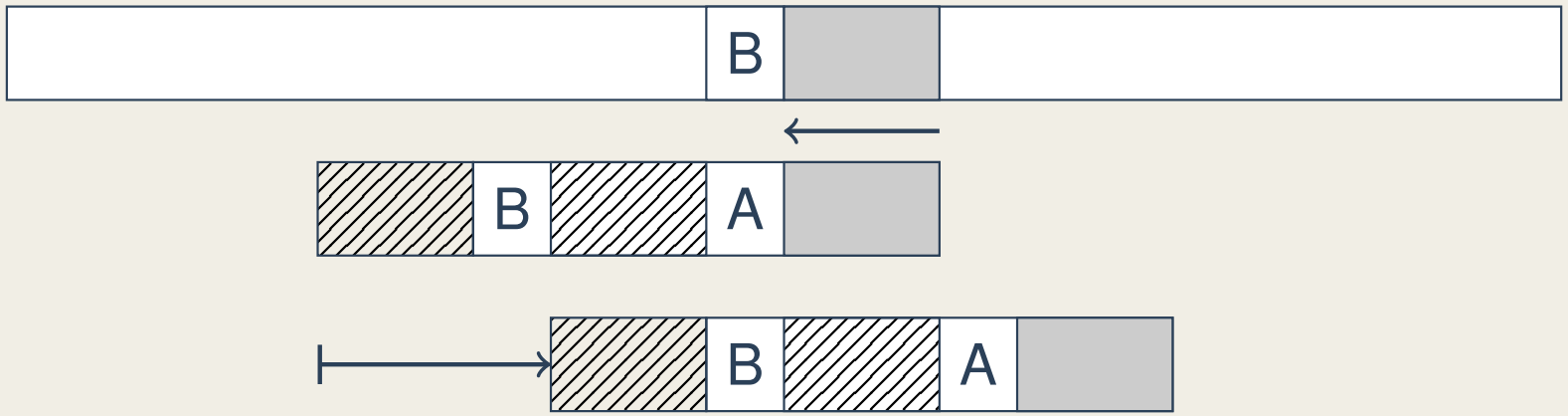
Melhorando o algoritmo trivial

- Algoritmo trivial faz deslocamentos unitários após cada comparação bem ou mal sucedida
- Quantidade de caracteres (símbolos) é pequena
- Palavra é em geral muito menor que texto
- Proposta: preprocessar palavra para determinar deslocamentos não unitários

A regra do caractere ruim

Pré-condição: casamento parcial (pela direita) de `p[i+1..m-1]` e `t[i+k+1..k+m-1]` e desacordo em `p[i]=A`$\neq$`t[i+k]=B`

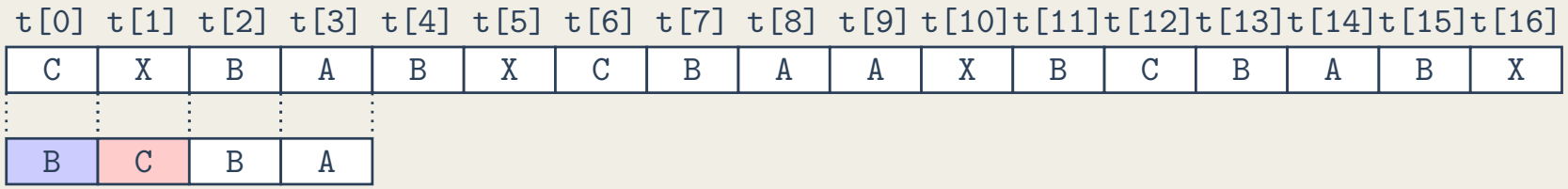
$\Rightarrow$ Alinhar `t[i+k]` com última ocorrência de `B` em `p[0..m-2]`



$\Rightarrow$ Se `B` não ocorrer em `p[0..m-2]`, alinhe palavra com `t[i+k+1]`

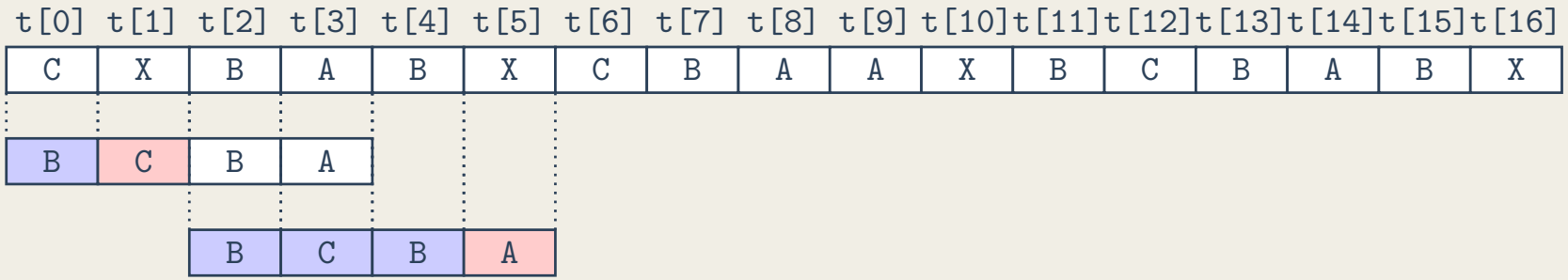
A regra do caractere ruim

Exemplo



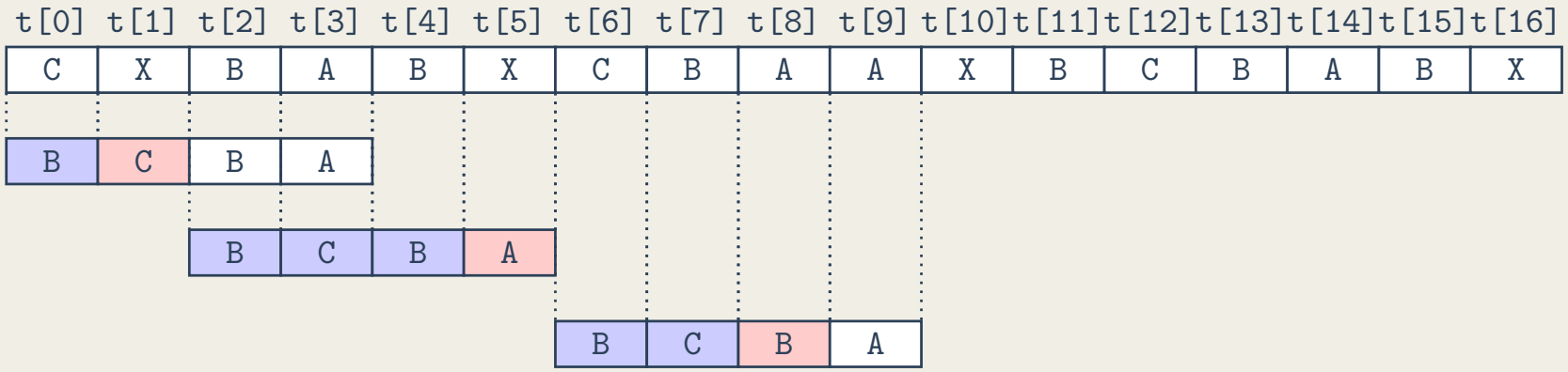
A regra do caractere ruim

Exemplo



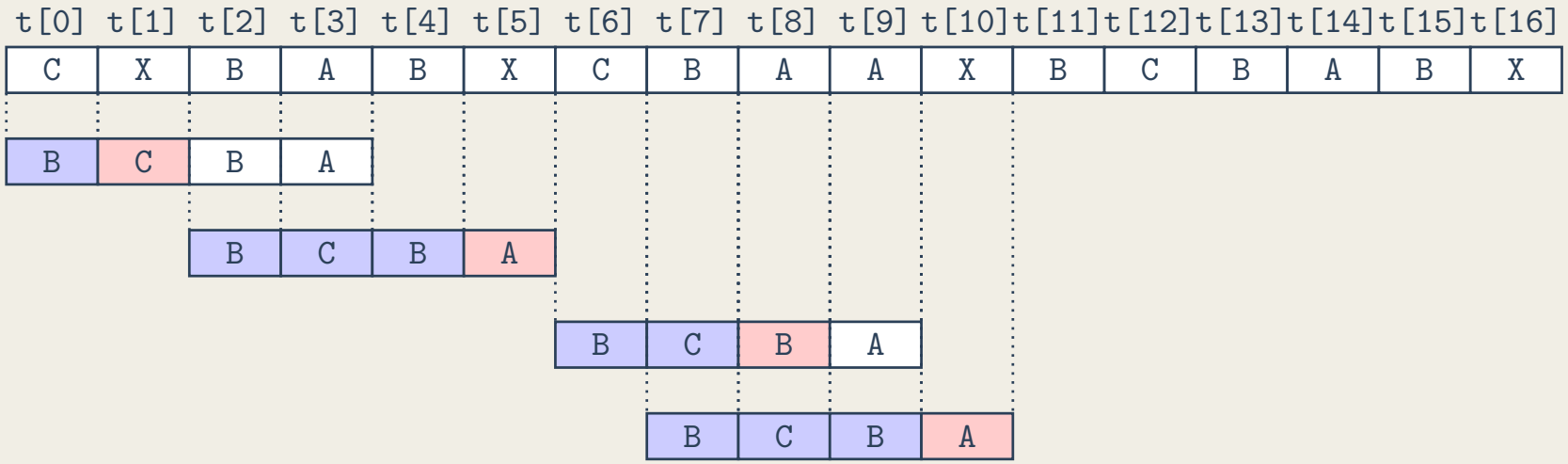
A regra do caractere ruim

Exemplo



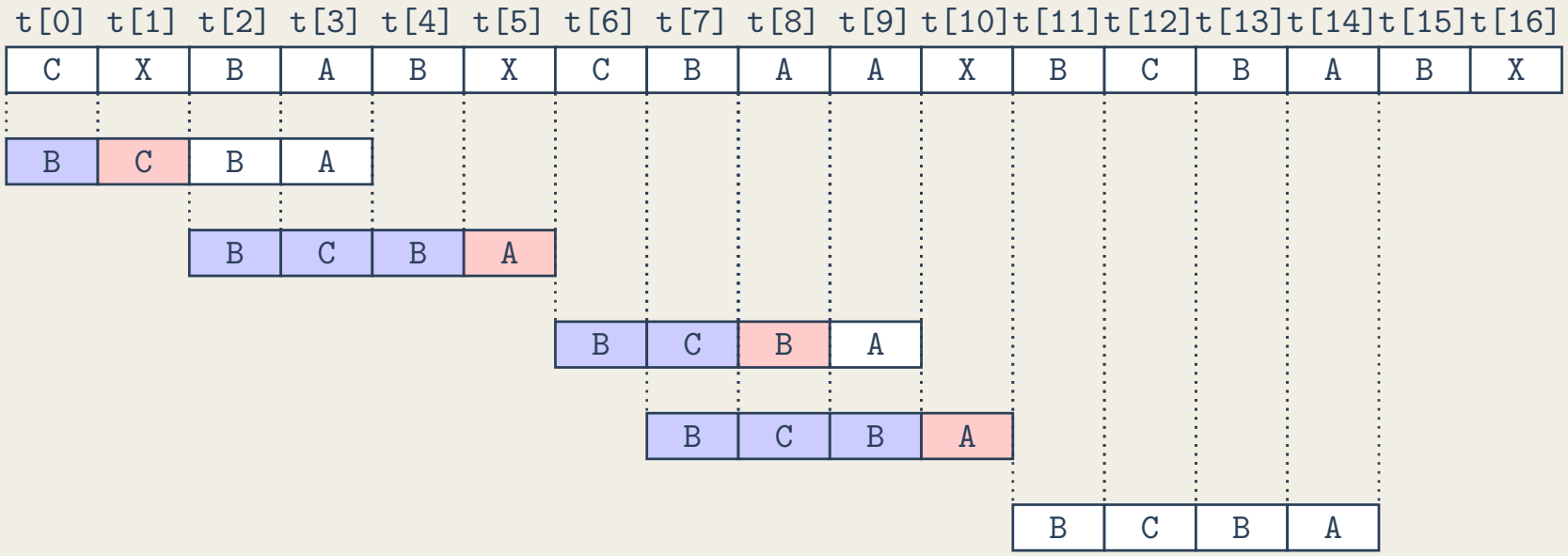
A regra do caractere ruim

Exemplo



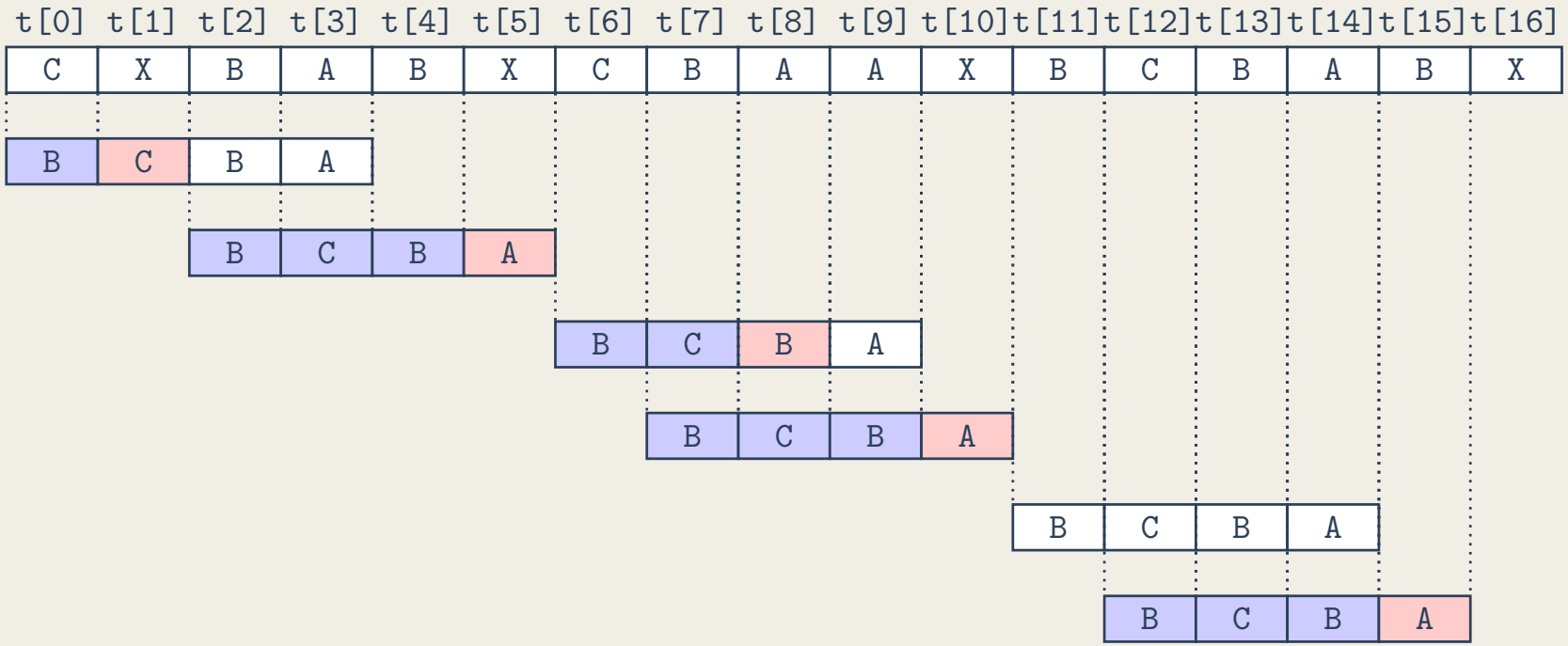
A regra do caractere ruim

Exemplo



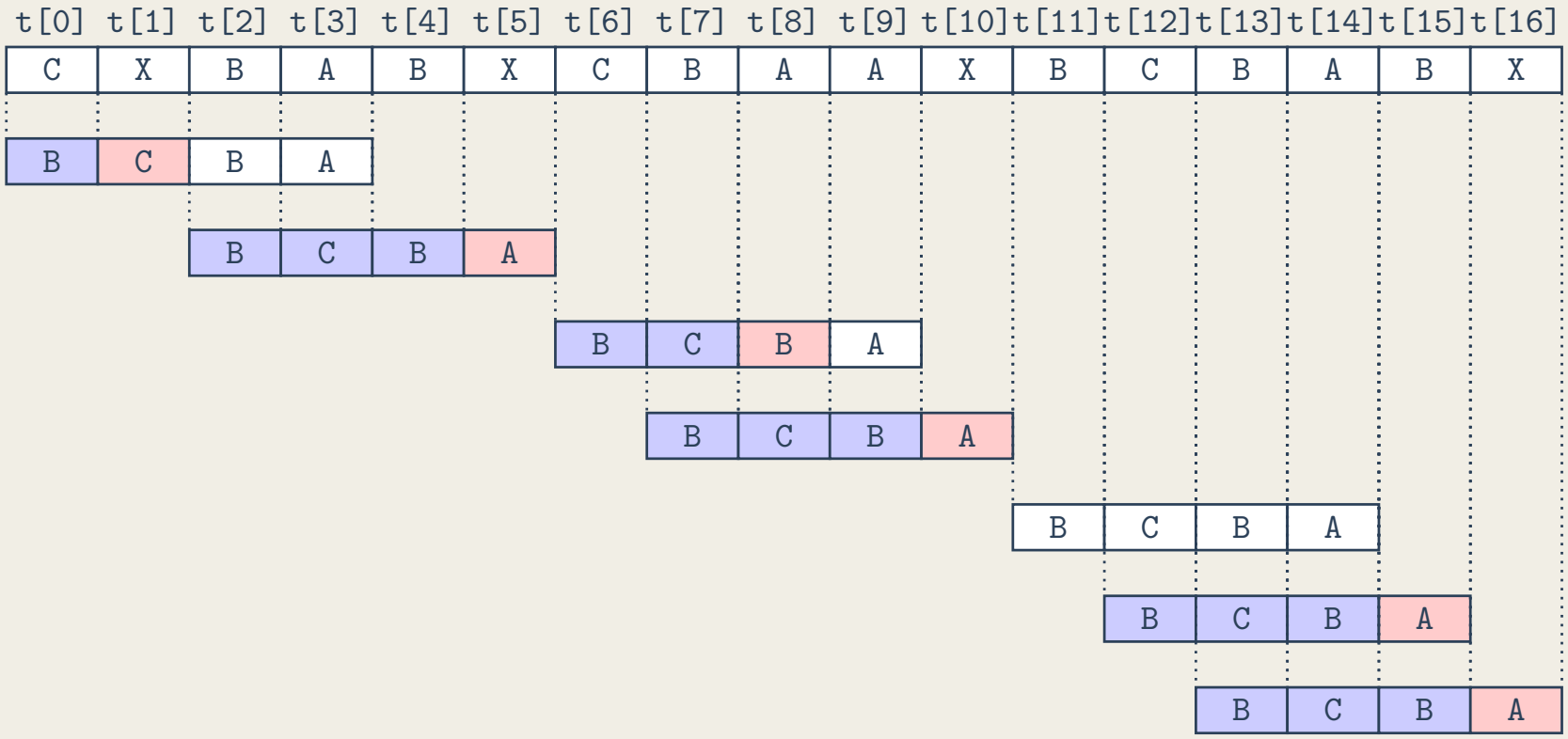
A regra do caractere ruim

Exemplo



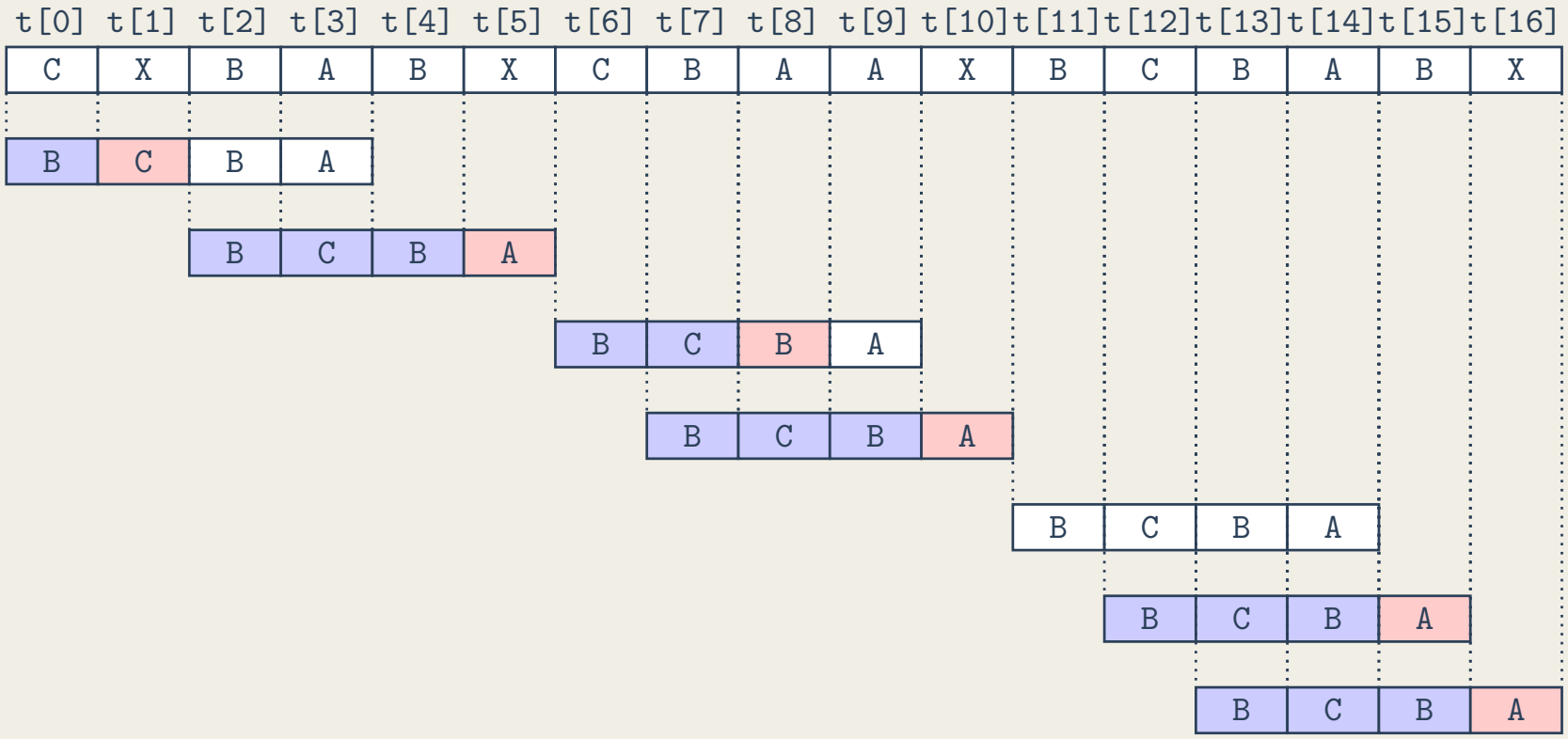
A regra do caractere ruim

Exemplo



A regra do caractere ruim

Exemplo



A regra do caractere ruim

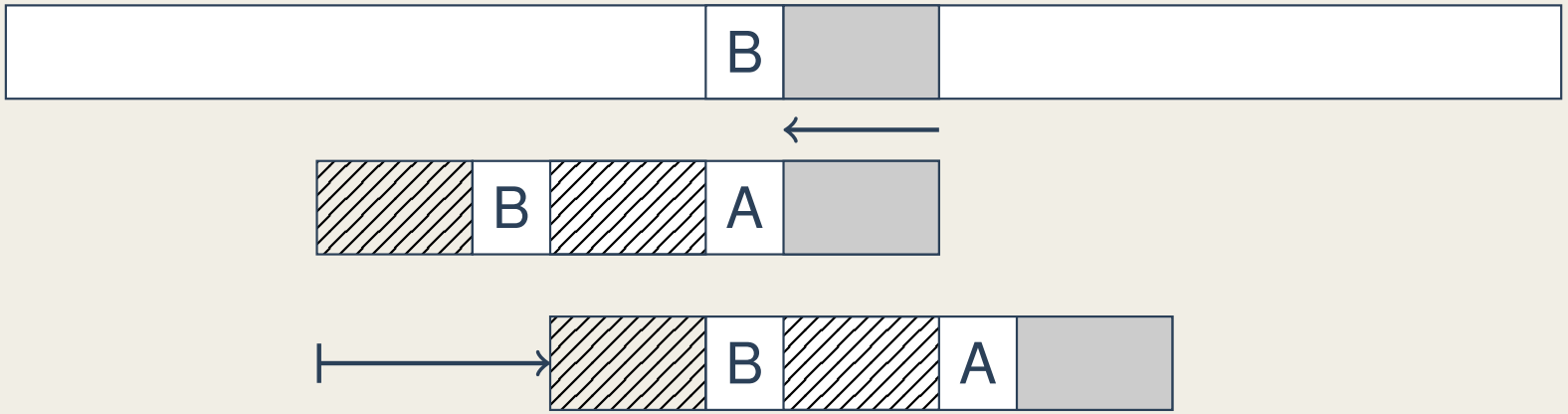
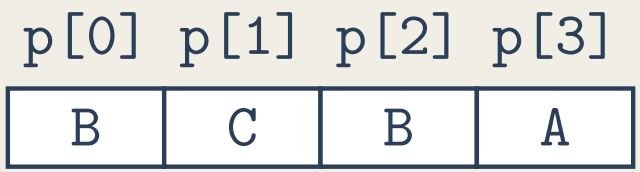


Tabela de deslocamentos

$$\texttt{T1}[c] = \begin{cases} \texttt{m} - (\arg\max_j \texttt{p}[j] + 1) = c, & \text{se } c \text{ ocorre em } \texttt{p}[0..\texttt{m}-2], \\ \texttt{m}, & \text{caso contrário.} \end{cases}$$

A regra do caractere ruim

```
1 void preprocessa1 (string p, int m, int T1[])
2 {
3   int j, m;
4   for (j = 0; j < 127; j++) T1[j] = m;
5   for (j = 0; j < m-1; j++) T1[p[j]] = m - j - 1;
6 }
```



c	...	!	"	...	=	>	?	@	A	B	C	D	E	F	G	...
T1[c]	...	4	4	...	4	4	4	4	4	1	2	4	4	4	4	...

A regra do caractere ruim

```
1 void preprocessa1 (string p, int m, int T1[])
2 {
3   int j, m;
4   for (j = 0; j < 127; j++) T1[j] = m;
5   for (j = 0; j < m-1; j++) T1[p[j]] = m - j - 1;
6 }
```

p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]	p[7]
G	C	A	G	A	G	A	G

c	A	C	G	T
T1[c]	?	?	?	?

A regra do caractere ruim

```
1 void preprocessa1 (string p, int m, int T1[])
2 {
3   int j, m;
4   for (j = 0; j < 127; j++) T1[j] = m;
5   for (j = 0; j < m-1; j++) T1[p[j]] = m - j - 1;
6 }
```

p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]	p[7]
G	C	A	G	A	G	A	G

c	A	C	G	T
T1[c]	1	6	2	8

Busca com deslocamento pela regra do caractere ruim

```
1  int busca_em_texto_cr (string p, string t)
2  {
3      int i, j, k, m, n, o = 0;    int T1[127];
4     m = strlen(p); n = strlen(t);
5      /* Pre-processamento */
6      preprocessa1(p, T1);
7      /* Busca */
8      k = 0;
9      while (k <= n-m) {
10         for (i = m-1; i >= 0 && p[i] == t[i+k]; i--);
11         if (i == -1)  { o++; j = 1; } /* ocorrência */
12         } else { /* p[i] != t[k+i] não casam */
13             j = T1[t[i+k]] - m + i + 1;
14             if (j < 1) j = 1;
15         }
16         k += j; /* desloca janela da palavra em j passos*/
17     }
18     return o;
19 }
```

Busca com deslocamento pela regra do caractere ruim

```
1  int busca_em_texto_cr (string p, string t)
2  {
3     int i, j, k, m, n, o = 0;   int T1[127];
4     m = strlen(p); n = strlen(t);
5     preprocessa1(p, m, T1);
6     k = 0;
7     while (k <= n-m) {
8        for (i = m-1; i >= 0 && p[i] == t[i+k]; i--);
9        if (i == -1)  { o++; j = 1; }
10       } else {
11          j = T1[t[i+k]] - m + i + 1;
12          if (j < 1) j = 1;
13       }
14       k += j;
15    }  return o;
16 }
```

Complexidade (pior caso)?

Busca com deslocamento pela regra do caractere ruim

```
1  int busca_em_texto_cr (string p, string t)
2  {
3     int i, j, k, m, n, o = 0;   int T1[127];
4     m = strlen(p); n = strlen(t);
5     preprocessa1(p, m, T1);
6     k = 0;
7     while (k <= n-m) {
8        for (i = m-1; i >= 0 && p[i] == t[i+k]; i--);
9        if (i == -1)  { o++; j = 1; }
10       } else {
11          j = T1[t[i+k]] - m + i + 1;
12          if (j < 1) j = 1;
13       }
14       k += j;
15    }  return o;
16 }
```

Complexidade (pior caso)? $O(m + 127) + O(mn)$

Melhor caso?

Busca com deslocamento pela regra do caractere ruim

```
1  int busca_em_texto_cr (string p, string t)
2  {
3      int i, j, k, m, n, o = 0;    int T1[127];
4     m = strlen(p); n = strlen(t);
5      preprocessa1(p, m, T1);
6      k = 0;
7      while (k <= n-m) {
8          for (i = m-1; i >= 0 && p[i] == t[i+k]; i--);
9          if (i == -1)  { o++; j = 1; }
10         } else {
11             j = T1[t[i+k]] - m + i + 1;
12             if (j < 1) j = 1;
13         }
14         k += j;
15     }   return o;
16 }
```

Complexidade (pior caso)? $O(m + 127) + O(mn)$

Melhor caso? $O(n/m)$

A regra do sufixo bom: Caso 1

Pré-condição: casamento parcial (pela direita) de `p[i+1..m-1]` e `t[i+k+1..k+m-1]` e desacordo em `p[i]=A`$\neq$`t[i+k]=B`

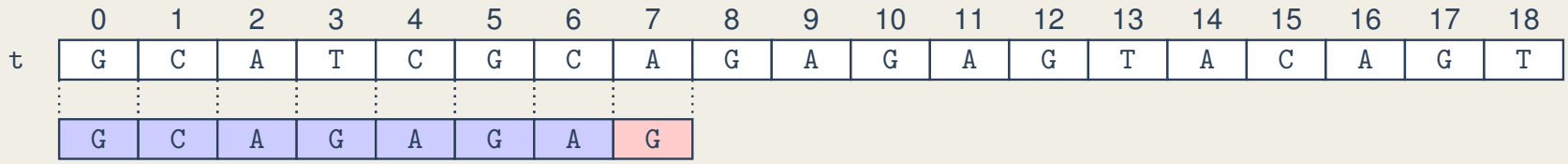
$\Rightarrow$ Alinhe `t[i+k+1..k+m-1]` com ocorrência anterior de `p[i+1..m-1]` em `p` que é precedida por caractere distinto de `p[i]`.

Exemplo

	0	1	2	3	4	5	6	7	8	9	10	11	12
t	X	M	A	N	P	A	**N**	**A**	**M**	P	N	A	M
p		A	N	A	M	P	**N**	**A**	**M**				
p						A	**N**	**A**	**M**	**P**	**N**	**A**	**M**

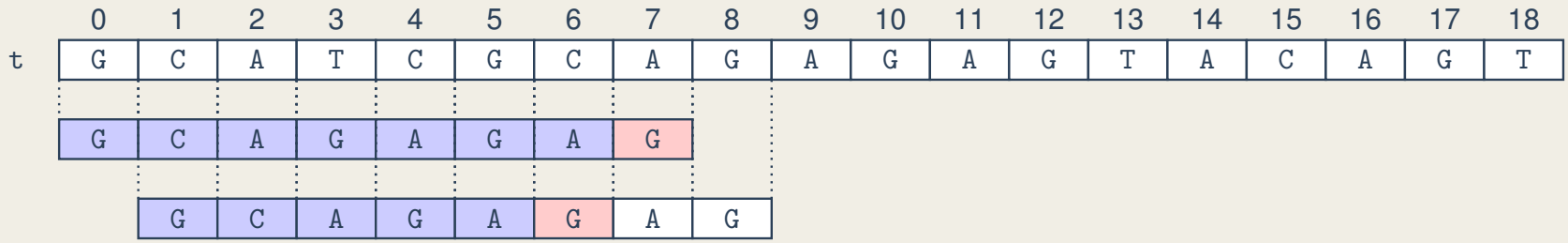
A regra do sufixo bom: Caso 1

Exemplo



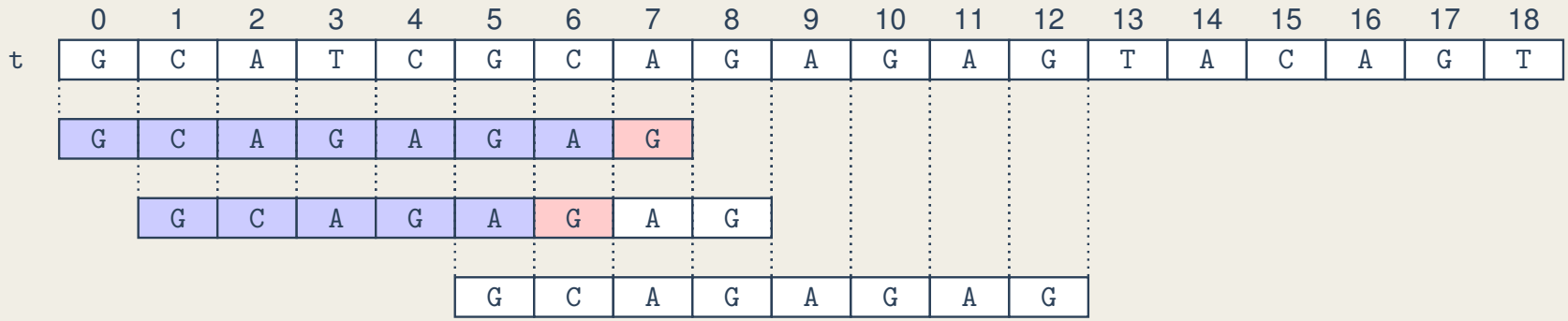
A regra do sufixo bom: Caso 1

Exemplo



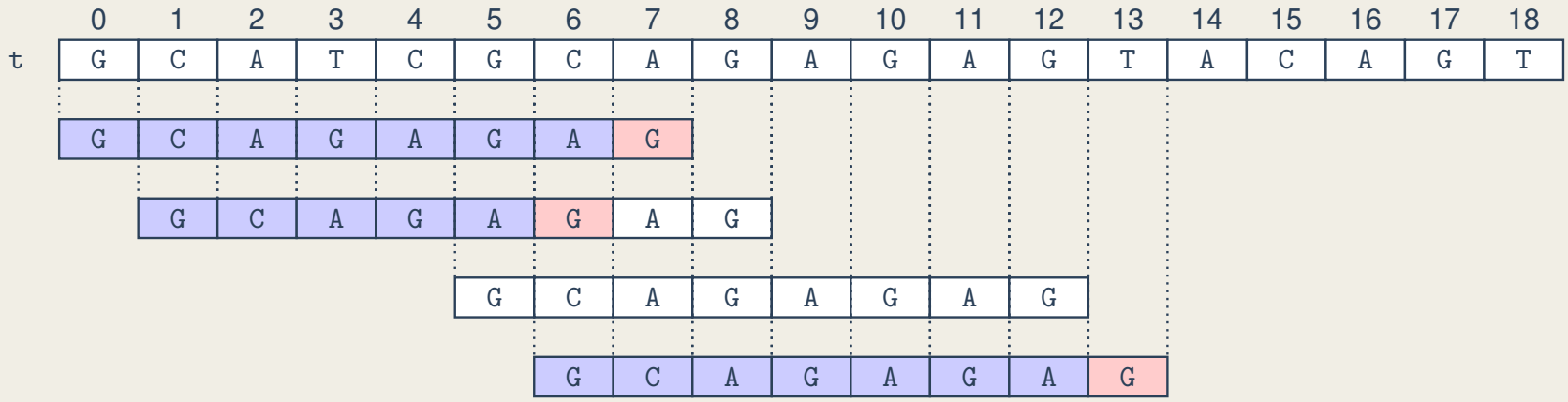
A regra do sufixo bom: Caso 1

Exemplo



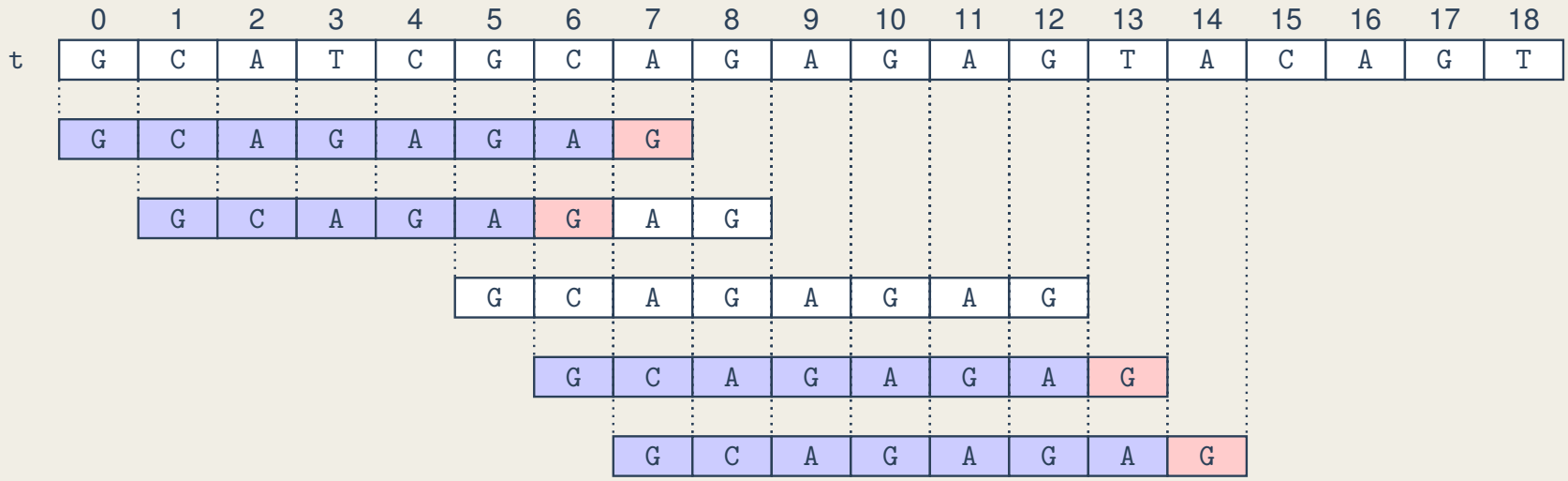
A regra do sufixo bom: Caso 1

Exemplo



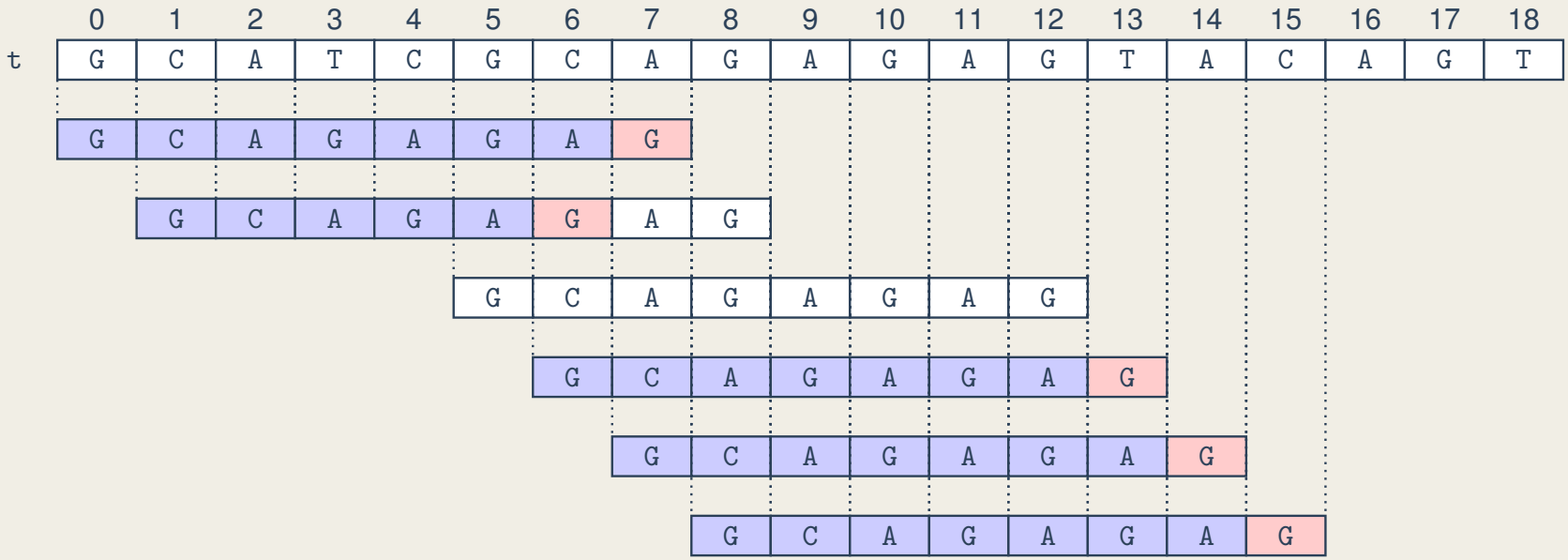
A regra do sufixo bom: Caso 1

Exemplo



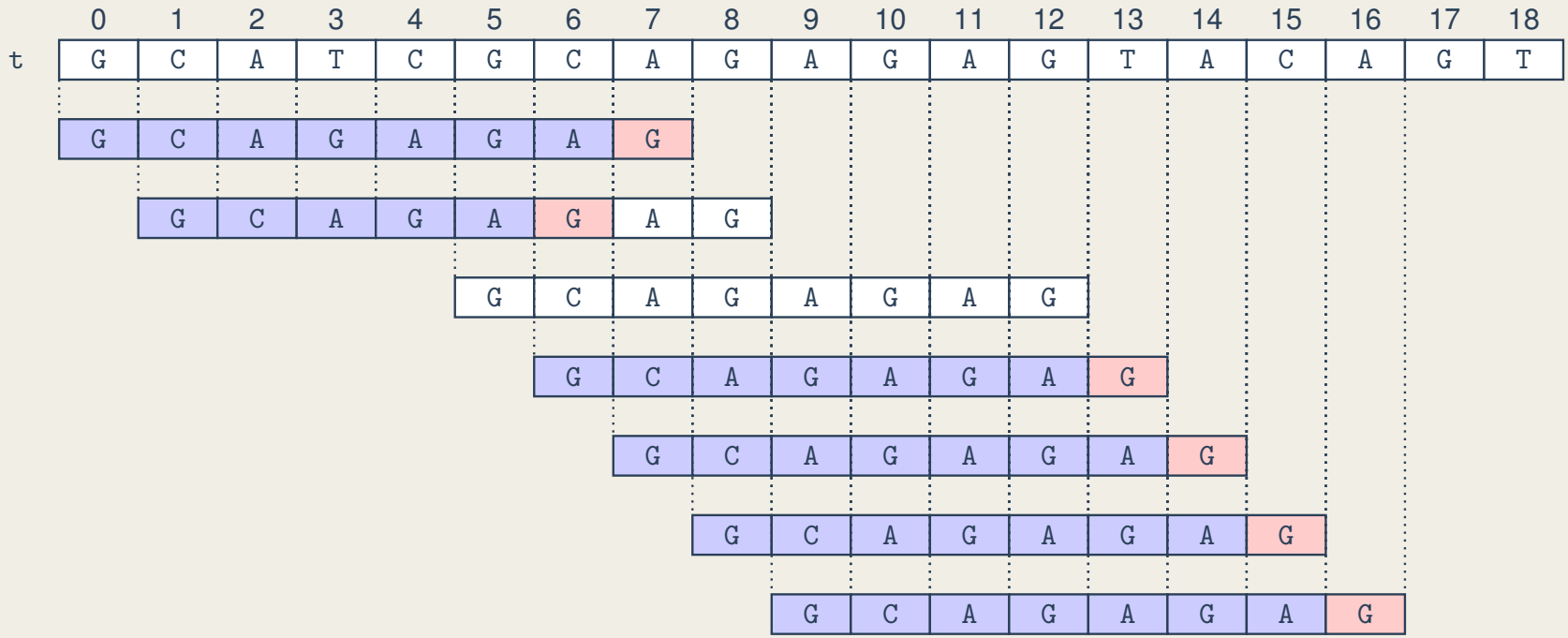
A regra do sufixo bom: Caso 1

Exemplo



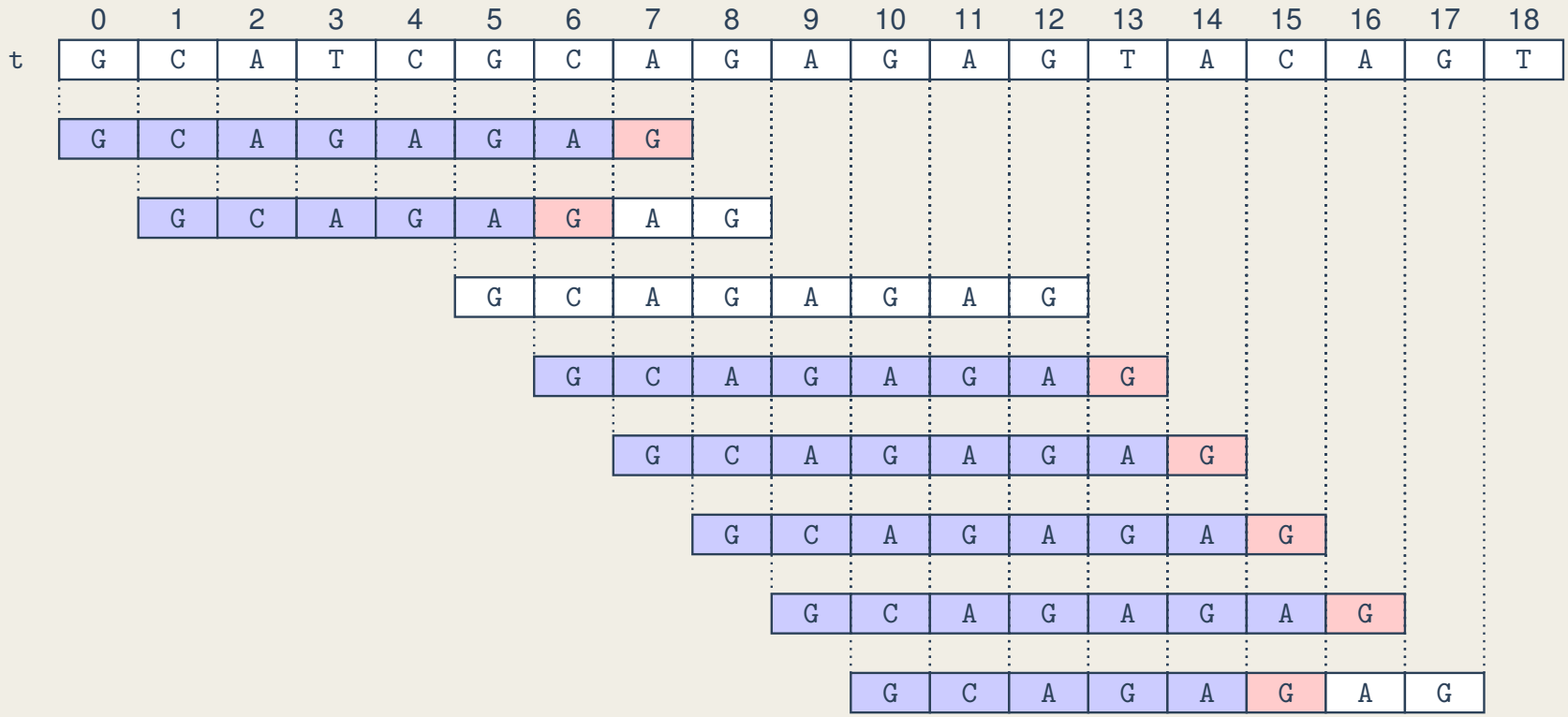
A regra do sufixo bom: Caso 1

Exemplo



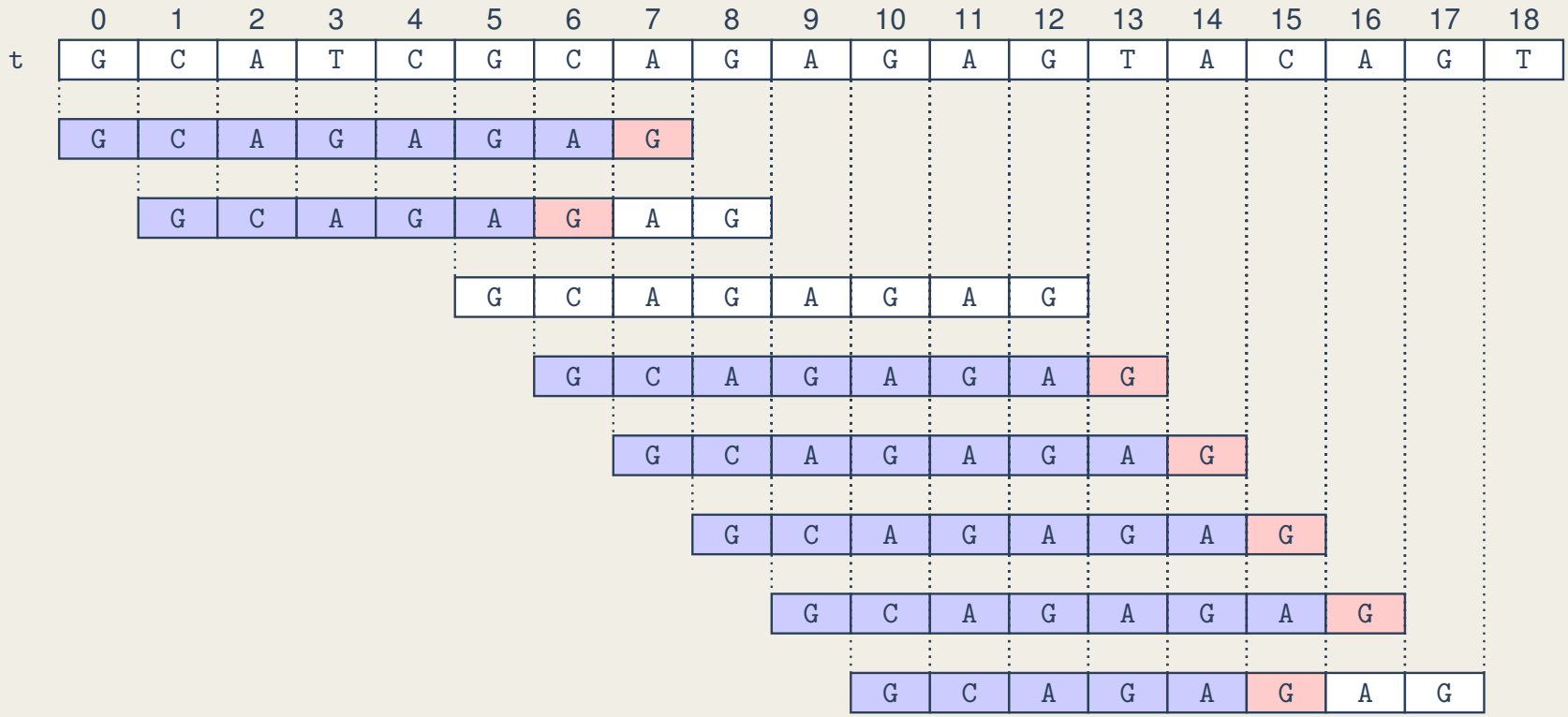
A regra do sufixo bom: Caso 1

Exemplo



A regra do sufixo bom: Caso 1

Exemplo



A regra do sufixo bom: Caso 1

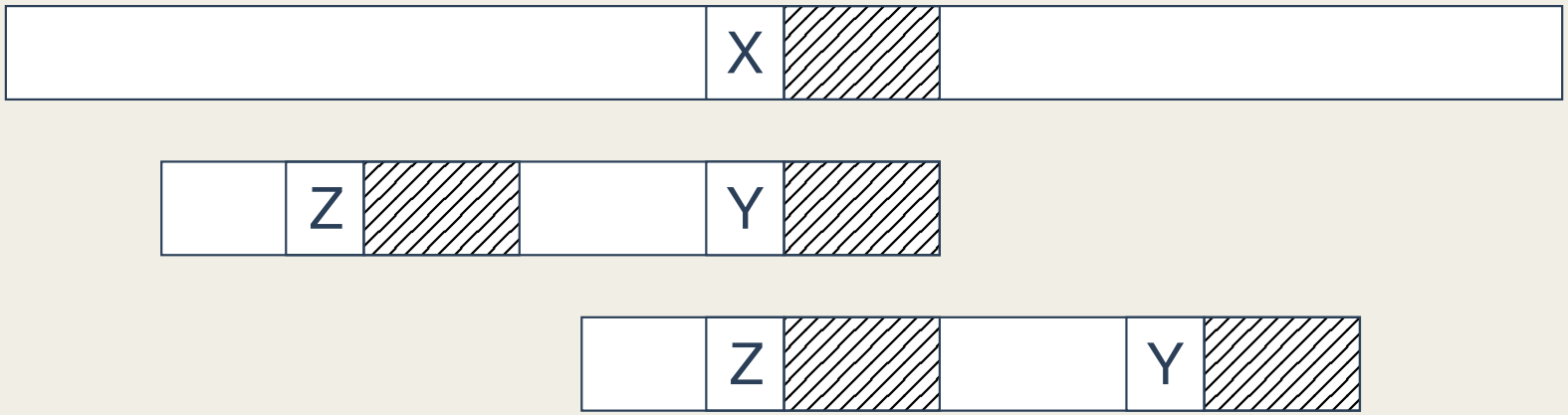


Tabela de deslocamentos

$$\texttt{T2}[i] = \max\{m - j - 1 \mid \texttt{p}[i+1..m-1] \text{ é sufixo de } \texttt{p}[0..j] \text{ e } \texttt{p}[i] \neq \texttt{p}[j - (m - i - 2)]\}$$

$\texttt{T2}[i] = 0$ caso não exista tal j.

A regra do sufixo bom: Caso 1

```
1  void preprocessa2 (string p, int m, int T[])
2  {
3       int i, j, m;
4       for (i = 0; i < m; i++) T[i] = 0;
5       /* T[i] = maior j: p[i+1..m-1] é sufixo de p[0..j] e p[
        i] != p[j-(m-i-2)] */
6       for (j=0; j<m-1; j++) {
7         /* menor i tal que p[i+1..m-1] é sufixo de p[0..j] */
8         for (i=m-1; (j>=m-1-i) && (p[j-(m-1-i)]==p[i]); i--);
9          /* se p[i] != p[j-(m-i-2)], atualiza tabela */
10         if (j>=m-1-i) T[i] = m-j-1; // = i - [j-(m-1-i)]
11      }
12 }
```

	0	1	2	3	4	5
p	C	A	A	B	A	A

i	0	1	2	3	4	5
p[i]	C	A	A	B	A	A
T[i]	0	0	0	3	1	–

A regra do sufixo bom: Caso 1

```
1 void preprocessa2 (string p, int m, int T[])
2 {
3     int i, j;
4     for (i = 0; i < m; i++) T[i] = 0;
5     for (j = 0; j < m-1; j++) {
6         for (i = m-1; (j >= m-1-i) && (p[j-(m-1-i)] == p[i])
      ; i--);
7         if (j >= m-1-i) T[i] = m-j-1; // = i - [j-(m-1-i)]
8     }
9 }
```

i	0	1	2	3	4	5	6	7	8
p[i]	C	A	N	A	M	P	N	A	M
T[i]	?	?	?	?	?	?	?	?	?

A regra do sufixo bom: Caso 1

```
1 void preprocessa2 (string p, int m, int T[])
2 {
3     int i, j;
4     for (i = 0; i < m; i++) T[i] = 0;
5     for (j = 0; j < m-1; j++) {
6         for (i = m-1; (j >= m-1-i) && (p[j-(m-1-i)] == p[i])
      ; i--);
7         if (j >= m-1-i) T[i] = m-j-1; // = i - [j-(m-1-i)]
8     }
9 }
```

i	0	1	2	3	4	5	6	7	8
p[i]	C	A	N	A	M	P	N	A	M
T[i]	0	0	0	0	0	4	0	0	–

A regra do sufixo bom: Caso 2

Pré-condição: casamento parcial (pela direita) de `p[i+1..m-1]` e `t[i+k+1..k+m-1]`, desacordo em `p[i]=A`$\neq$`t[i+k]=B` e caso 1 não aplicável

$\Rightarrow$ Alinhe `t[i+k+1..k+m-1]` com ocorrência de sufixo de `p[i+1..m-1]` em prefixo de `p[0..m-1]`.

Exemplo

	0	1	2	3	4	5	6	7	8	9	10	11	12
t	X	M	A	N	P	A	**N**	**A**	**M**	P	N	A	M
p				A	M	P	**N**	**A**	**M**				
p								**A**	**M**	**P**	**N**	**A**	**M**

A regra do sufixo bom: Caso 2

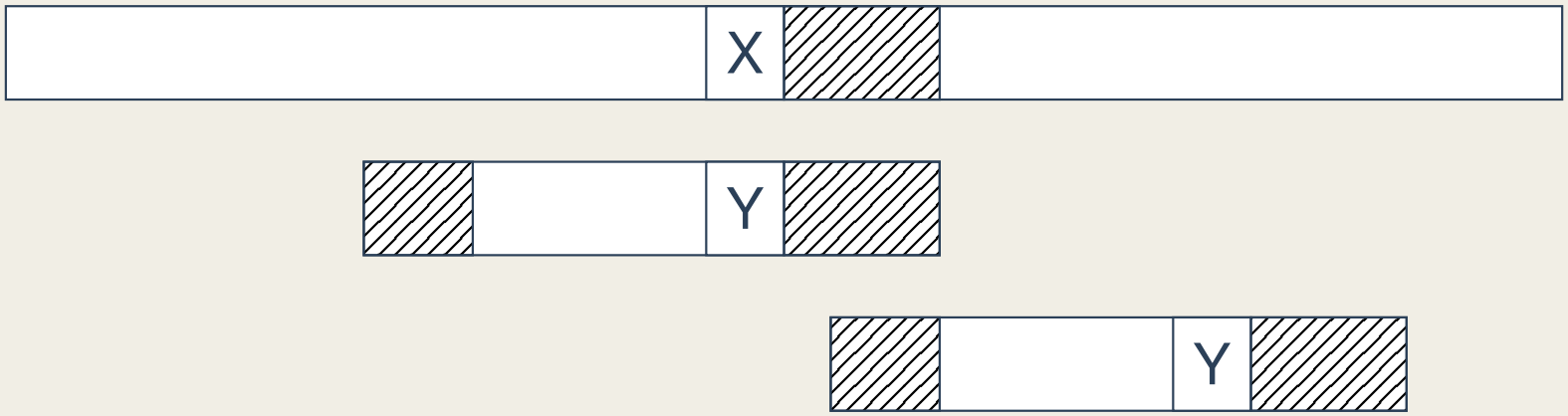


Tabela de deslocamentos

$$\texttt{T2}[i] = \max\{m - j - 1 \mid \texttt{p}[0..j] \text{ é sufixo de } \texttt{p}[i+1..m-1]\}$$

$\texttt{T2}[i] = 0$ caso não exista tal j.

A regra do sufixo bom: Caso 2

```
1  void preprocessa3 (string p, int m, int T[]) {
2      int i, j, k, t;     j = m-1;
3      for (i = m-2; i >= 0; i--) {
4        t = m - i - 1; // tamanho de p[i+1..m-1]
5        // p[0..t-1] = p[i+1..m-1]?
6        for (k = 0; (k < t) && (p[k] == p[i+k+1]); k++);
7        if (k == t) j = t - 1;
8        // sufixo de p[i+1..m-1] é igual a p[0..j]
9        T[i] = m-j-1;
10     }
11 }
```

i	0	1	2	3	4	5	6
p[i]	A	M	C	P	N	A	M
T[i]	5	5	5	5	5	0	–

Busca com deslocamento pela regra do sufixo bom

```
1  int busca_em_texto_sb (string p, string t)
2  {
3     int i, j, k, m, n, o = 0;    int T2[CAP];
4     m = strlen(p); n = strlen(t);
5     preprocessa3 (p, m, T2); // Testa caso 2 primeiro
6     preprocessa2 (p, m, T2);
7     k = 0;
8     while (k <= n-m) {
9        for (i = m-1; i >= 0 && p[i] == t[i+k]; i--);
10       if (i == -1)  { o++; k += 1; }
11       else k += max(T2[i],1);
12    }  return o;
13 }
```

Complexidade (pior caso)?

Busca com deslocamento pela regra do sufixo bom

```
1  int busca_em_texto_sb (string p, string t)
2  {
3      int i, j, k, m, n, o = 0;    int T2[CAP];
4     m = strlen(p); n = strlen(t);
5      preprocessa3 (p, m, T2); // Testa caso 2 primeiro
6      preprocessa2 (p, m, T2);
7      k = 0;
8      while (k <= n-m) {
9          for (i = m-1; i >= 0 && p[i] == t[i+k]; i--);
10         if (i == -1)  { o++; k += 1; }
11         else k += max(T2[i],1);
12     }  return o;
13 }
```

Complexidade (pior caso)? $O(m^2) + O(mn)$

Caso médio: $O(n)$

A regra do sufixo bom: Casos 1 e 2

Tabela de deslocamentos

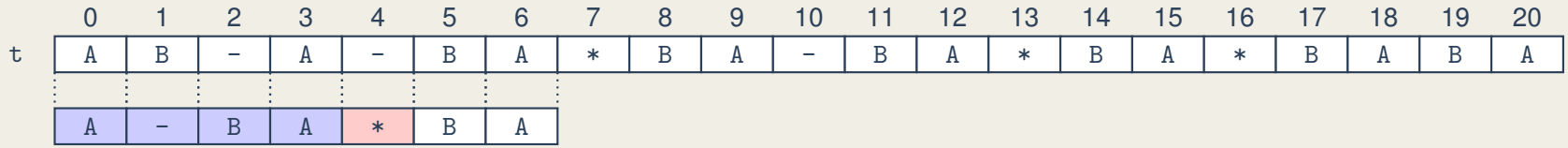
$$\texttt{T2}[\texttt{i}] = \max\Big\{ m - j - 1 \mid \texttt{p}[\texttt{i}+\texttt{1..m}-\texttt{1}] \text{ é sufixo de } \texttt{p}[\texttt{0..j}] \\ \text{ou } \texttt{p}[\texttt{0..j}] \text{ é sufixo de } \texttt{p}[+\texttt{1..m}-\texttt{1}] \Big\} .$$

Exemplo:

i	0	1	2	3	4	5	6
p	A	-	B	A	*	B	A
T2$[i]$	6	6	6	6	3	6	–

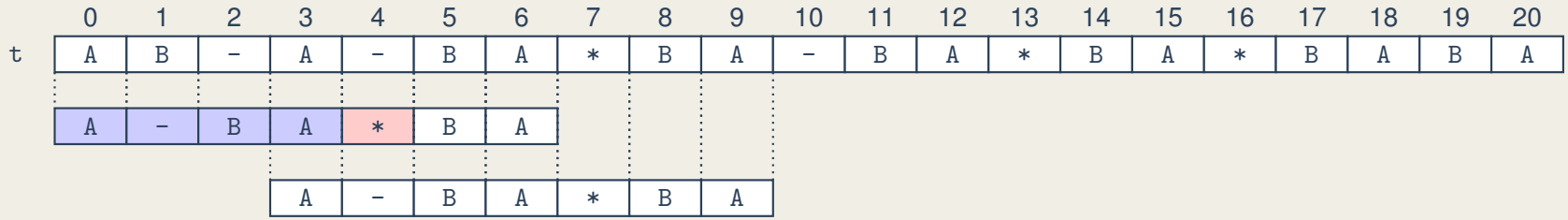
A regra do sufixo bom: Casos 1 e 2

Exemplo



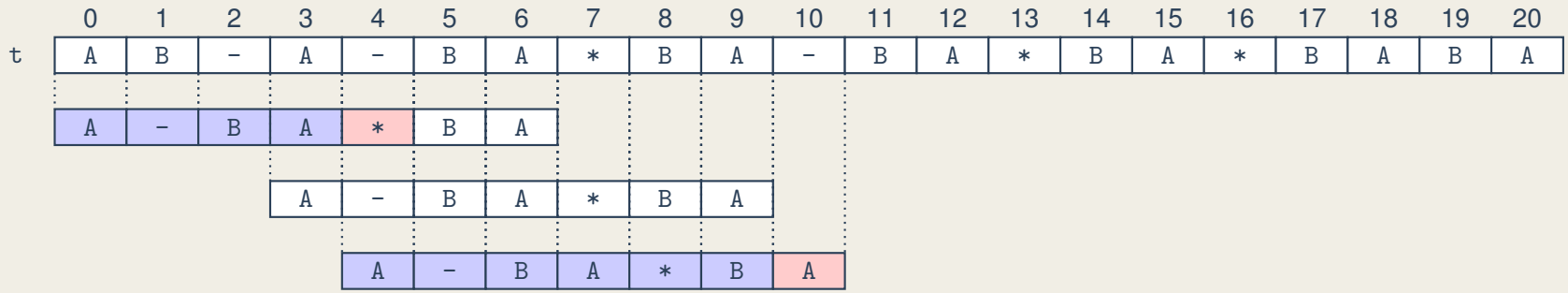
A regra do sufixo bom: Casos 1 e 2

Exemplo



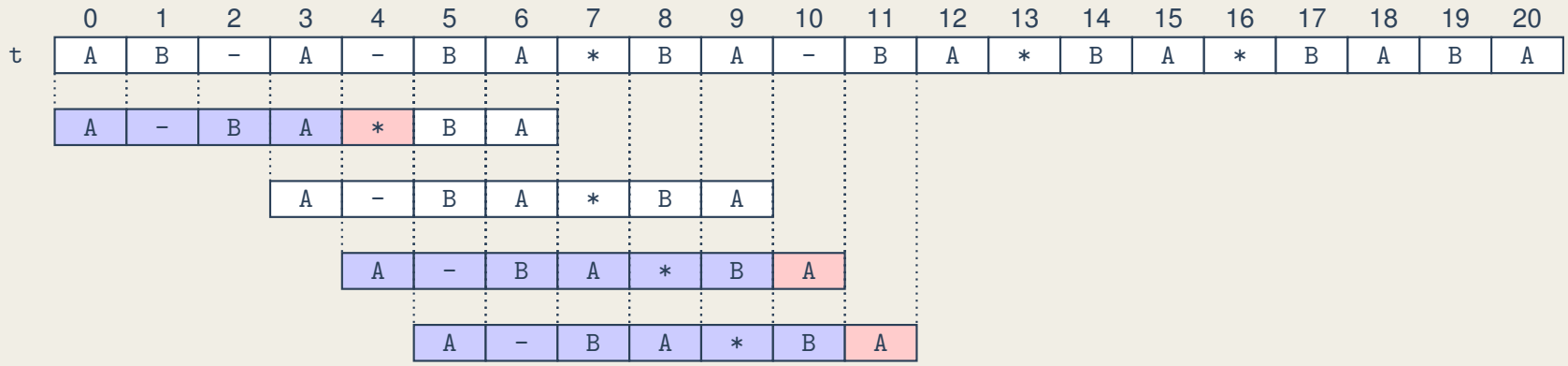
A regra do sufixo bom: Casos 1 e 2

Exemplo



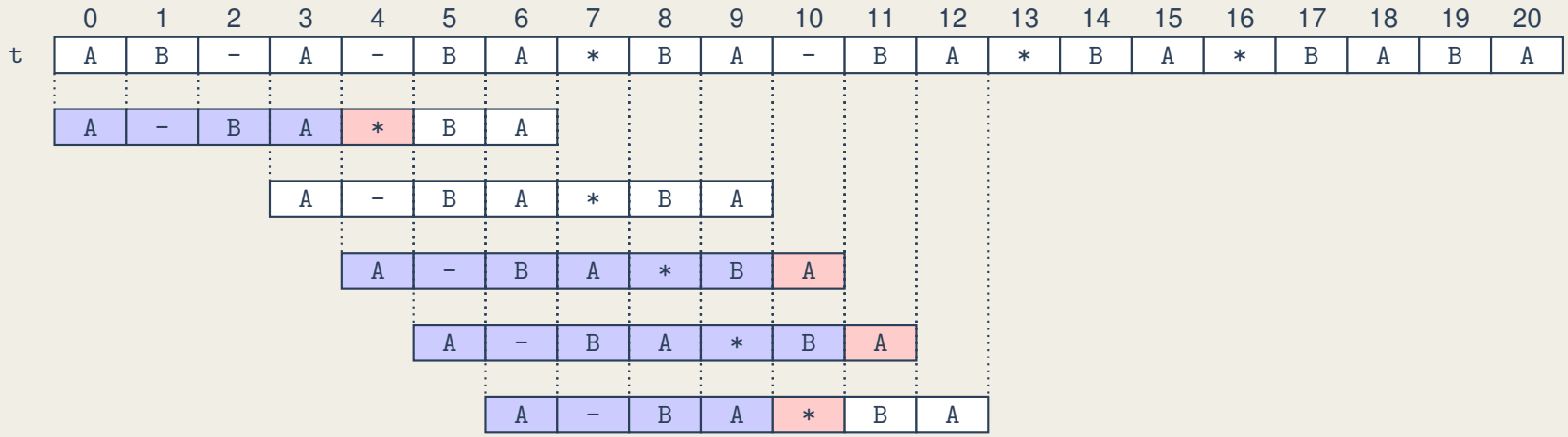
A regra do sufixo bom: Casos 1 e 2

Exemplo



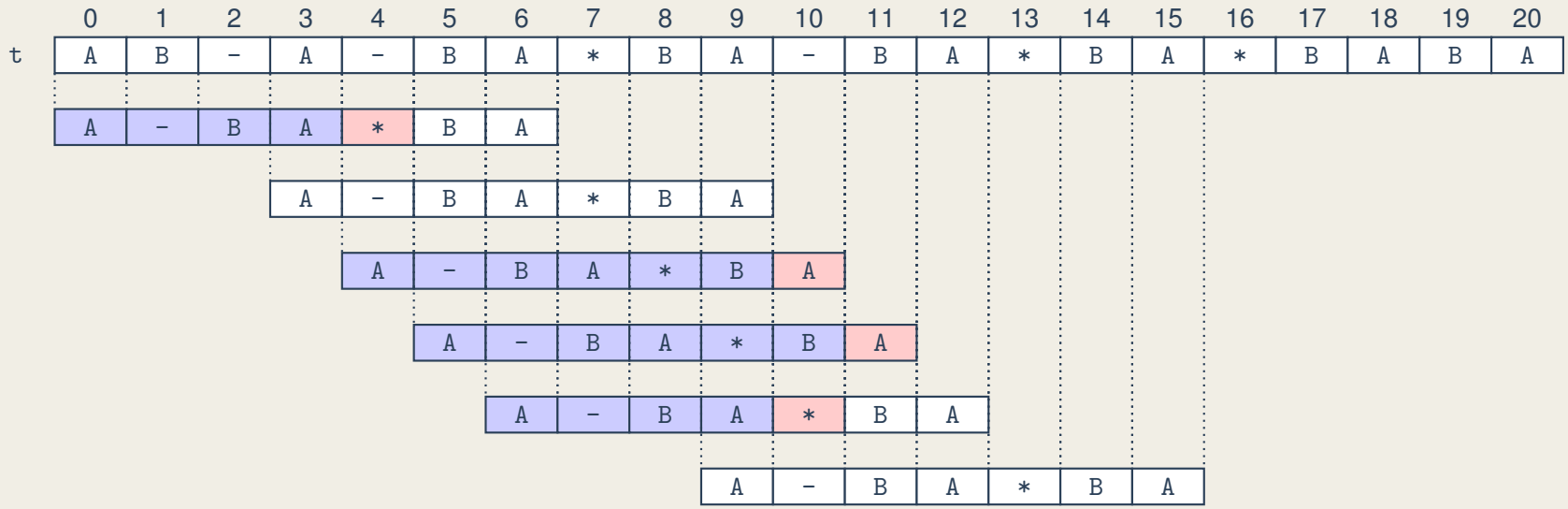
A regra do sufixo bom: Casos 1 e 2

Exemplo



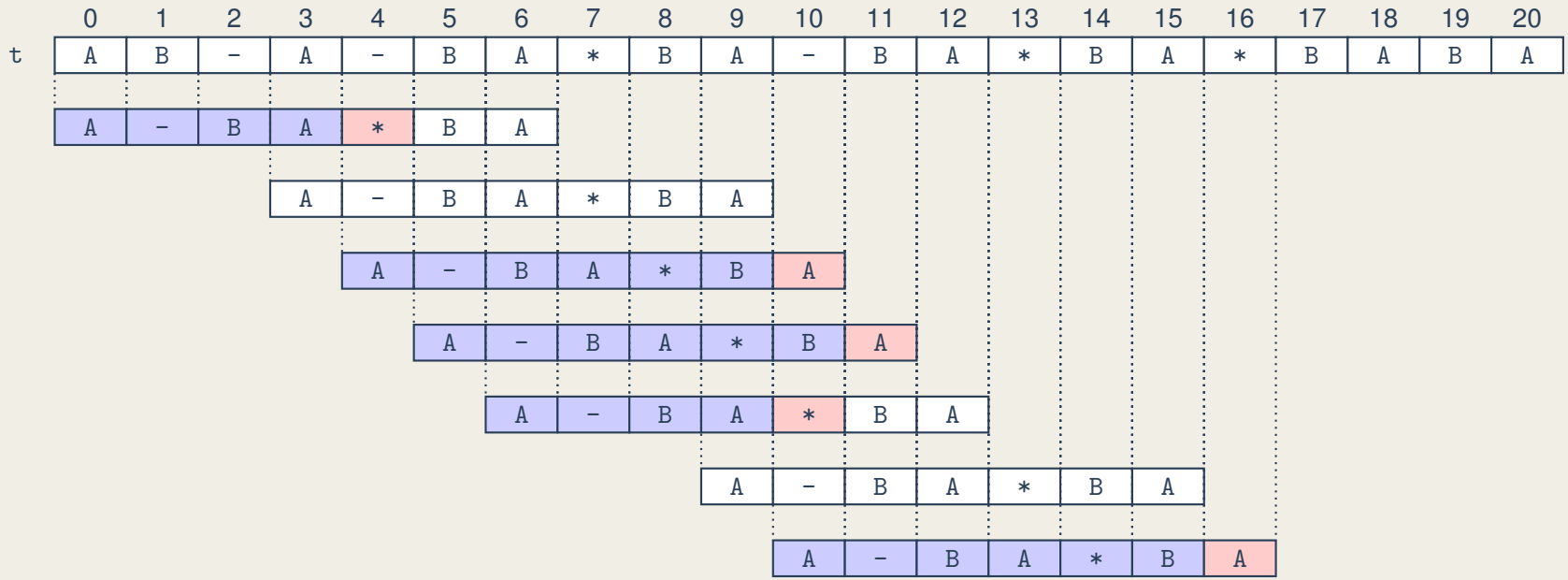
A regra do sufixo bom: Casos 1 e 2

Exemplo



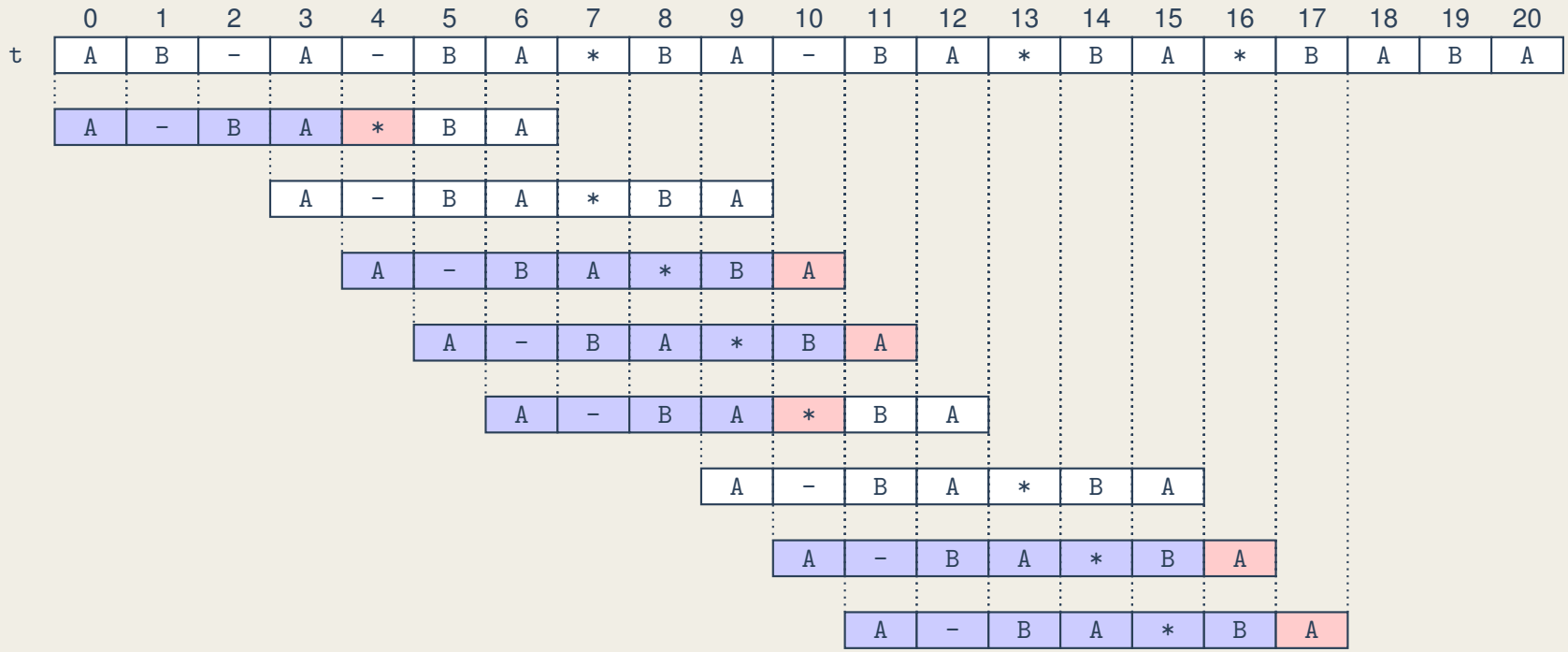
A regra do sufixo bom: Casos 1 e 2

Exemplo



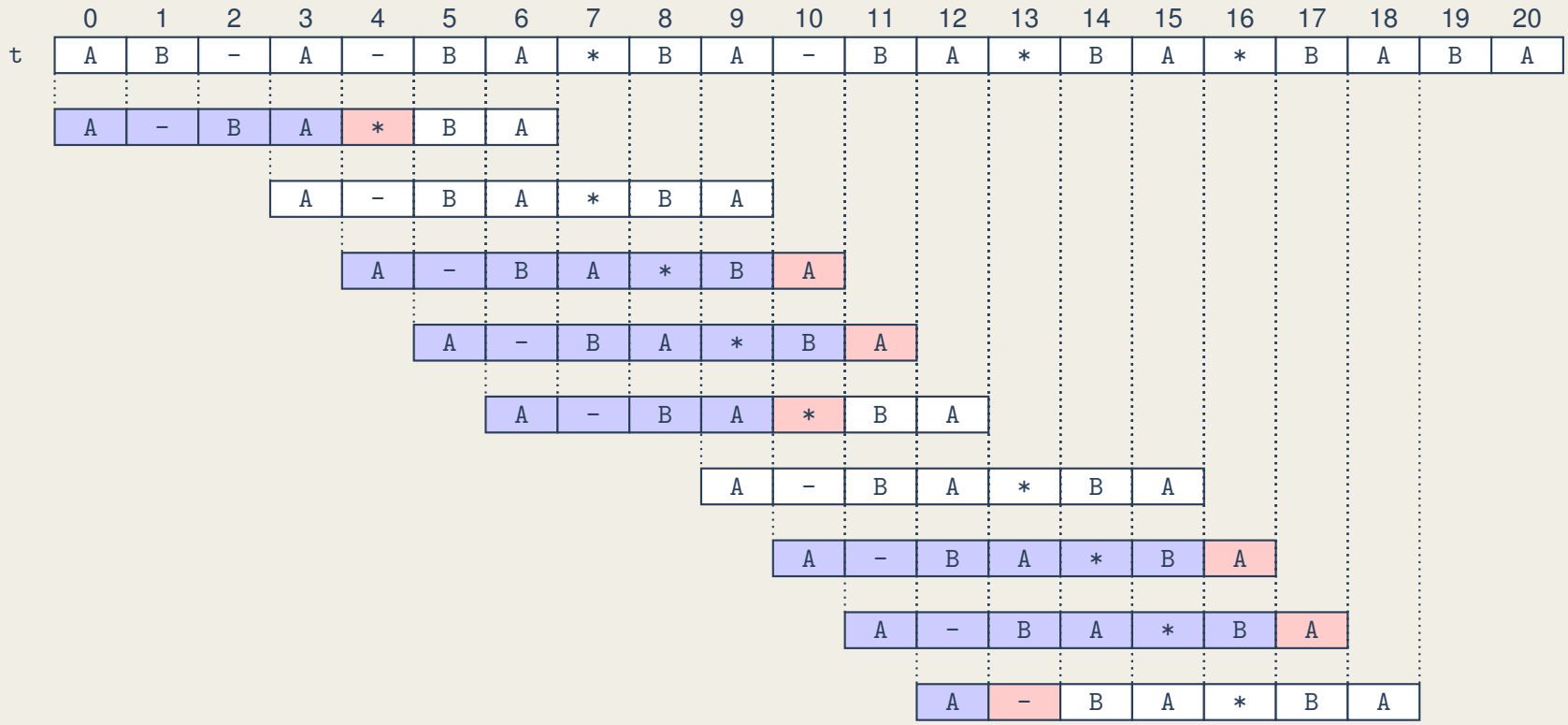
A regra do sufixo bom: Casos 1 e 2

Exemplo



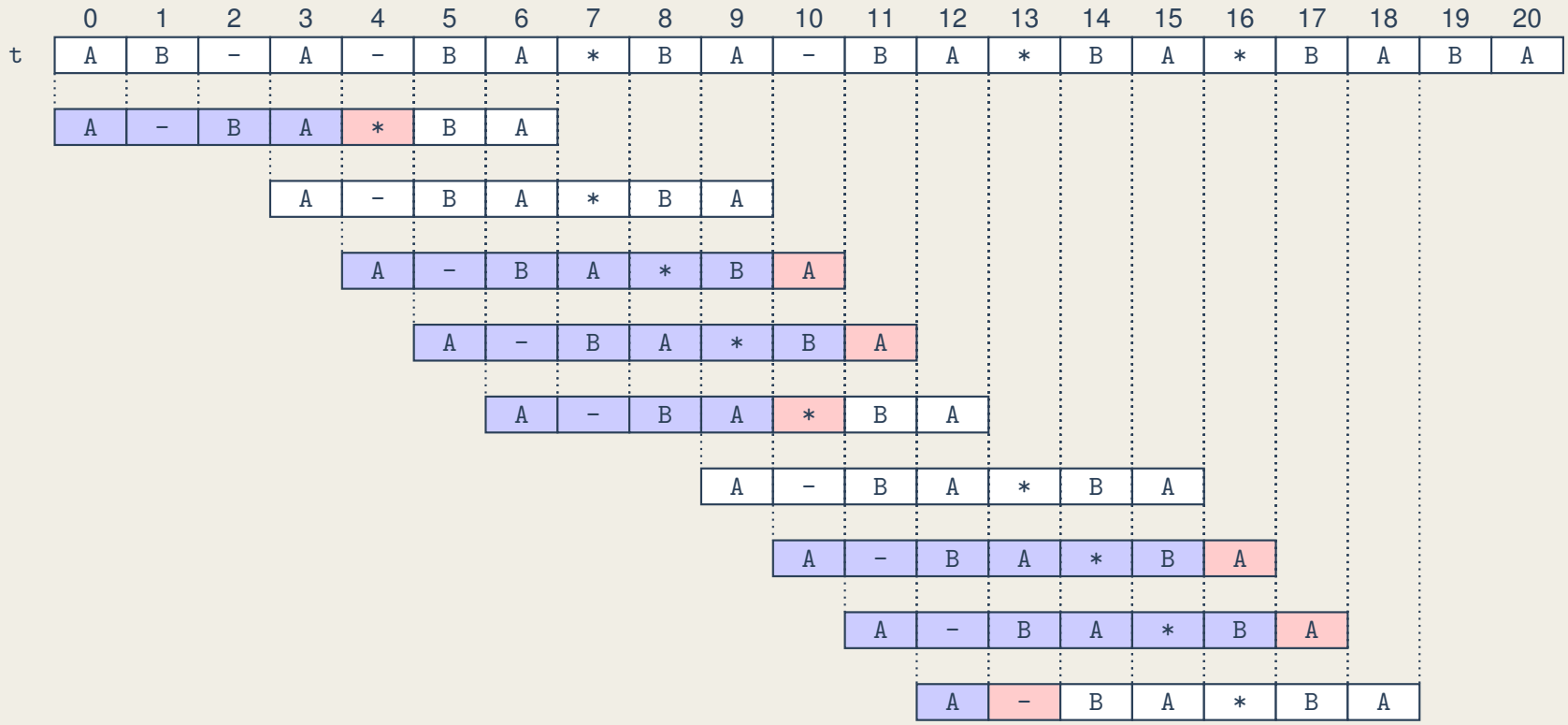
A regra do sufixo bom: Casos 1 e 2

Exemplo



A regra do sufixo bom: Casos 1 e 2

Exemplo



Algoritmo de Boyer-Moore

Escolha `j` como o maior entre `T1` e `T2` e atualize `k += j`

- Pré-processamento: $O(m^2)$
- Busca: $O(mn)$ no pior caso, $O(n)$ na média

Algoritmo de Boyer-Moore

```
1  int busca_em_texto_bm (string p, string t)
2  {
3      int i, j, k, m, n, o = 0;    int T2[CAP];
4     m = strlen(p); n = strlen(t);
5      preprocessa1 (p, m, T1); // Caractere ruim
6      preprocessa3 (p, m, T2); // Sufixo bom, caso 2
7      preprocessa2 (p, m, T2); // Sufixo bom, caso 1
8      k = 0;
9      while (k <= n-m) {
10         for (i = m-1; i >= 0 && p[i] == t[i+k]; i--);
11         if (i == -1)  { o++; k += 1; }
12         else k += max(T2[i],T1[t[i+k]]-m+i+1);
13     }  return o;
14 }
15 }
```

Para casa

Exercícios 5A-5B