

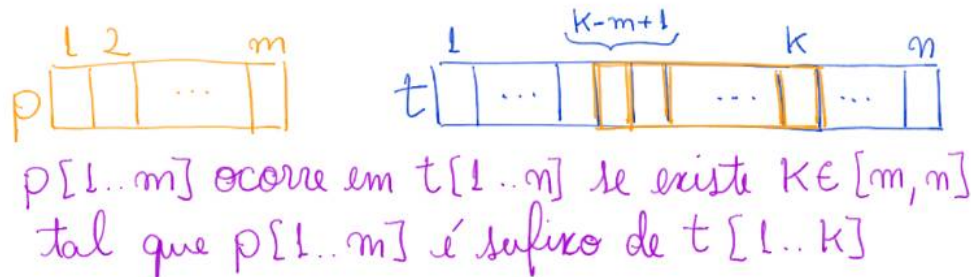
**Busca de palavras em um texto,  
algoritmo de Boyer-Moore (good suffix heuristic)**

**Relembrando a definição do problema**

Considere o problema de encontrar todas as ocorrências de

- uma sequência curta, que chamaremos de palavra,
- em uma sequência longa, que chamaremos de texto.

Mais formalmente, temos que



Para simplificar, vamos tratar do problema de determinar

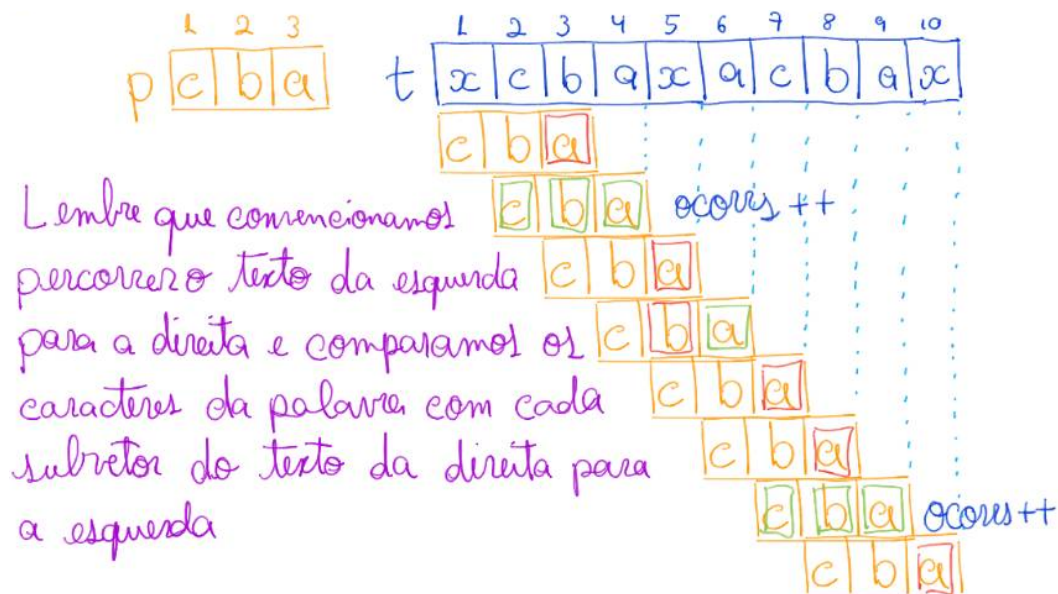
- o número de ocorrências de  $p[1..m]$  em  $t[1..n]$ .

Antes de prosseguir, vamos relembrar algumas convenções:

- Nossos algoritmos vão varrer o texto da esquerda para a direita
  - e cada vez que eles comparam a palavra com um subvetor do texto,
    - vamos varrer esse subvetor da direita para a esquerda.
- Em geral, as duas alternativas são equivalentes,
  - mas a heurística que veremos hoje exige que
    - a comparação seja feita no sentido contrário
      - ao da varredura do texto.
- **Atentar para onde isso será relevante.**

## Algoritmo básico

Exemplo:



Código:

```
// Recebe vetores palavra[1..m] e texto[1..n], com m >= 1 e n >= 0,  
// e devolve o número de ocorrências de palavra em texto.  
int básico(char palavra[], int m, char texto[], int n) {  
    int pos_t, desl_p, ocorrencias;  
    ocorrencias = 0;  
    for (pos_t = m; pos_t <= n; pos_t++) {  
        desl_p = 0;  
        // palavra[1..m] casa com texto[pos_t-m+1 .. pos_t]?  
        while (desl_p < m &&  
            palavra[m - desl_p] == texto[pos_t - desl_p])  
            desl_p++;  
        if (desl_p >= m)  
            ocorrencias++;  
    }  
    return ocorrencias;  
}
```

Eficiência de tempo:

- No pior caso, o tempo é  $O(mn)$ ,
  - pois o laço externo itera  $(n - m + 1)$  vezes
    - e o laço interno itera  $m$  vezes.
- No melhor caso o tempo é  $O(n - m + 1)$ , pois o laço interno nunca itera.

Eficiência de espaço:

- O espaço adicional utilizado é constante.

Agora vamos estudar a segunda heurística do algoritmo de Boyer-Moore

- que utiliza critérios não triviais para avançar o índice `pos_t` no texto
  - com passos maiores que 1 a cada iteração.

## Segundo algoritmo de Boyer-Moore

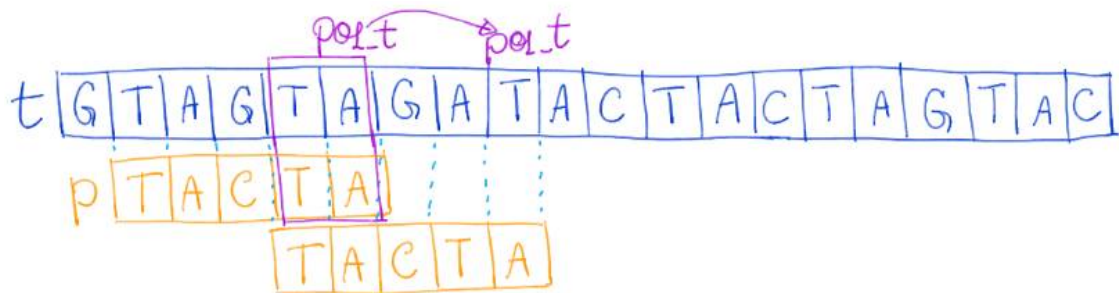
Vamos estudar a segunda heurística do algoritmo de Boyer-Moore,

- conhecida como “good suffix heuristic”.

Nos seguintes exemplos, considere que o algoritmo

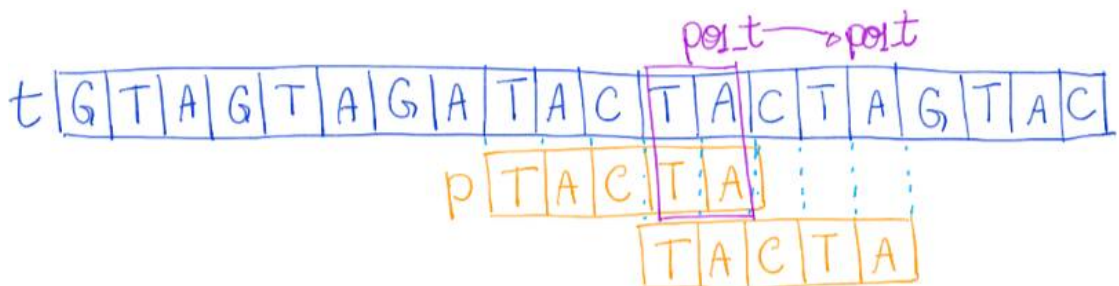
- acabou de testar se  $p[1..m]$  é sufixo de  $t[1..pos\_t]$ 
  - e acabou detectando que  $p[m - desl\_p .. m] = t[pos\_t - desl\_p .. pos\_t]$ .
- Então, responda à pergunta:
  - **Qual a próxima posição da palavra que pode emparelhar**
    - **com o trecho do texto que acabei de detectar?**

Exemplo 1:



$pos\_t$  avançou 3 posições depois de detectar que  $p[4..5] = t[pos\_t-1..pos\_t] = "TA"$ , pois a distância da próxima ocorrência de "TA" até o final de  $p[1..m]$  é 3.

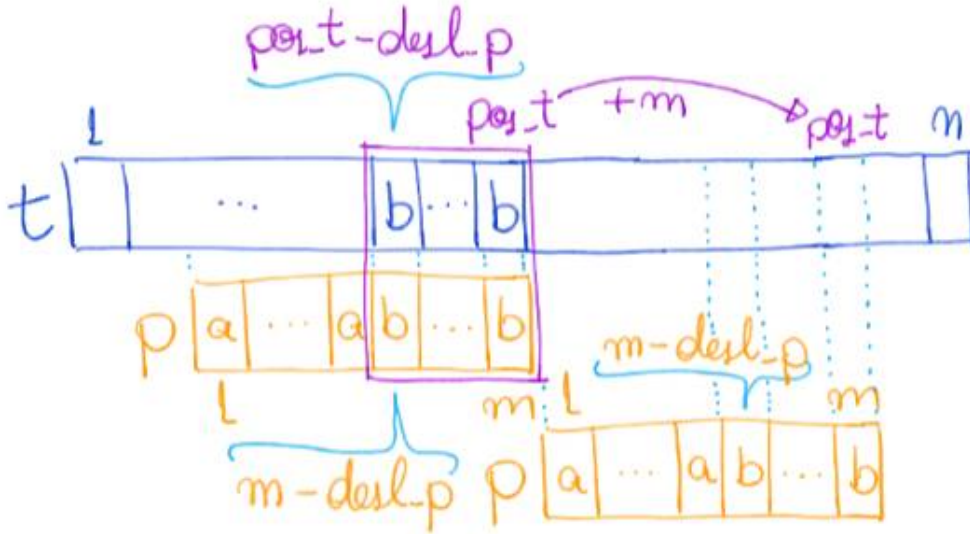
Exemplo 2:



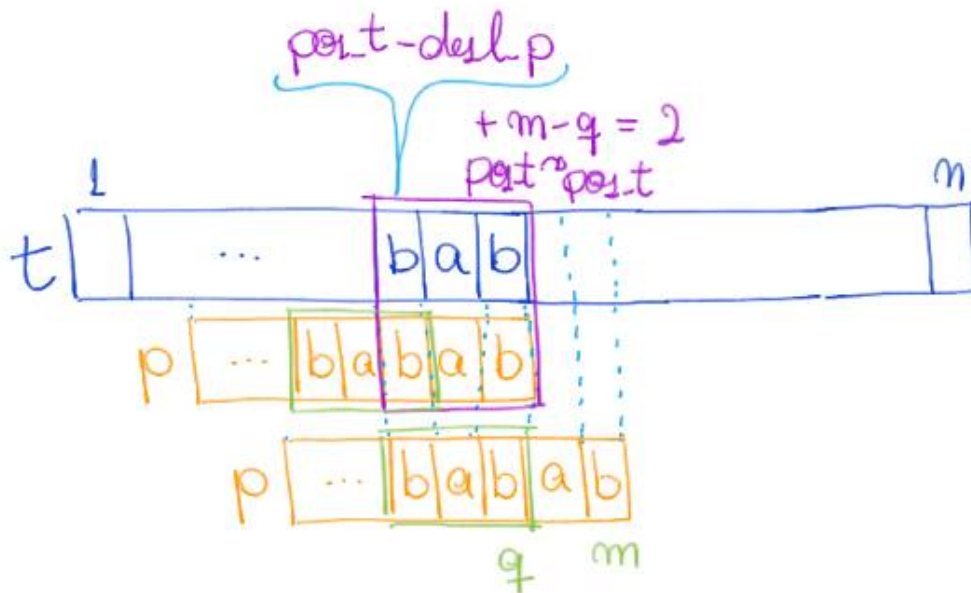
$pos\_t$  avançou 3 posições depois de detectar que  $p[1..5] = t[pos\_t-5..pos\_t] = "TACTA"$ , pois "TA" é o maior sufixo de  $p$  que também é prefixo de  $p$ , e a distância do primeiro "TA" até o final de  $p[1..m]$  é 3.

Ideia da “good suffix heuristic”:

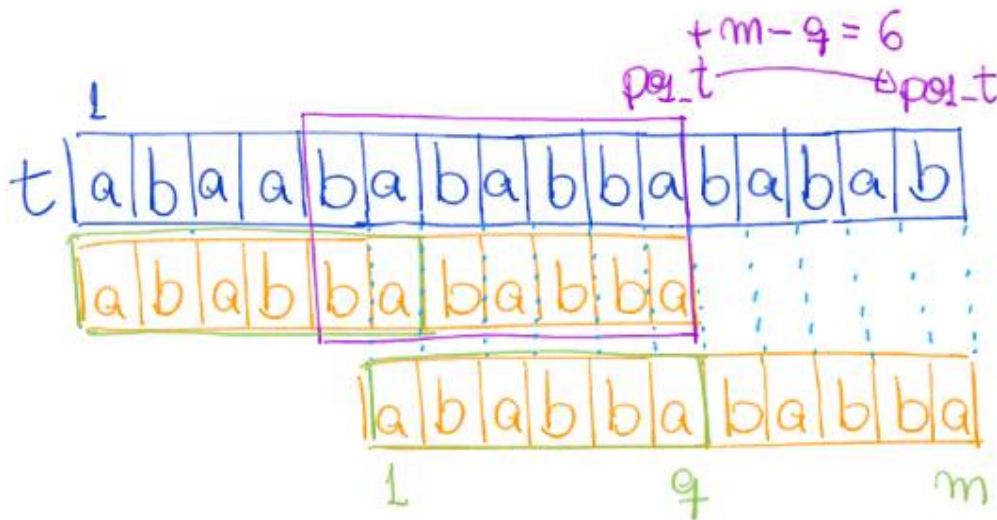
- Suponha que um sufixo de  $p[1 .. m]$  coincide com um sufixo de  $t[1 .. \text{pos}_t]$ ,
  - i.e.,  $p[m - \text{desl}_p .. m] = t[\text{pos}_t - \text{desl}_p .. \text{pos}_t]$ ,
    - para  $\text{desl}_p$  entre 1 e  $m$ .
- Agora, considere que a sequência no sufixo  $p[m - \text{desl}_p .. m]$ 
  - não se repete mais em  $p[1 .. m]$ .
- Neste caso, sabemos que podemos avançar  $\text{pos}_t$ 
  - até que a primeira posição da palavra  $p$  esteja depois do  $\text{pos}_t$  atual,
    - i.e.,  $\text{pos}_t += m$ .



- Agora, considere que a sequência no sufixo  $p[m - \text{desl}_p .. m]$ 
  - se repete em  $p[1 .. m]$  pelo menos uma vez
  - e a primeira repetição (contando da direita pra esquerda)
    - ocorre no subvetor  $p[q - \text{desl}_p .. q]$ , com  $q < m$ .
- Neste caso, sabemos que podemos avançar  $\text{pos}_t$ 
  - até que  $p[q]$  esteja alinhado com  $t[\text{pos}_t]$ ,
    - i.e.,  $\text{pos}_t += m - q$ .
- Note que, pode haver intersecção entre  $p[m - \text{desl}_p .. m]$  e  $p[q - \text{desl}_p .. q]$ ,
  - i.e., é possível ocorrer  $q \geq m - \text{desl}_p$ .



- Por fim, falta considerar um caso complementar
  - e possivelmente concomitante com os anteriores.
- Considere que a sequência  $p[m - \text{desl}_p .. m]$ 
  - não se repete integralmente em  $p[1 .. m]$ ,
  - mas um sufixo dela pode aparecer no prefixo de  $p[1 .. m]$ ,
    - i.e.,  $p[1 .. q] = p[m - q + 1 .. m]$ , com  $q \leq \text{desl}_p$ .
- Neste caso, sabemos que podemos avançar  $\text{pos}_t$ 
  - até que  $p[q]$  esteja alinhado com  $t[\text{pos}_t]$ ,
    - i.e.,  $\text{pos}_t += m - q$ .



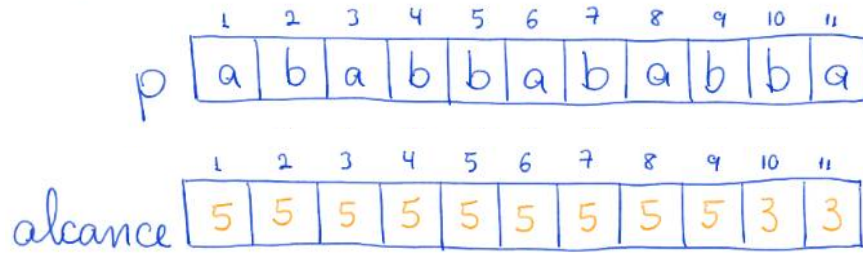
Sintetizando a ideia da “good suffix heuristic”:

- Para cada índice  $i$  entre 1 e  $m$ ,
  - corresponde um sufixo  $p[i .. m]$ .
- Queremos encontrar o maior índice  $q$ , tal que
  - $p[i .. m]$  é sufixo de  $p[1 .. q]$ , ou seja,
    - $q$  marca a primeira repetição de  $p[i .. m]$  em  $p[1 .. m]$ .
  - ou  $p[1 .. q]$  é sufixo de  $p[i .. m]$ , ou seja,
    - $q$  marca o maior prefixo de  $p[1 .. m]$ 
      - que casa com o final de  $p[i .. m]$ .
  - Se não existe tal  $q$ , então fazemos  $q = 0$ .
- Para implementar essa ideia e automatizar os saltos do índice  $\text{pos}_t$ ,
  - precisamos fazer um pré-processamento da palavra  $p[1 .. m]$ .

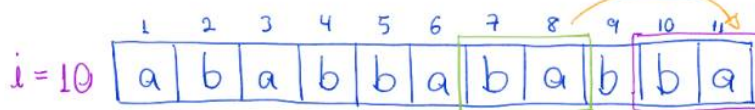
Neste pré-processamento, vamos

- alocar um vetor auxiliar  $\text{alcance}[1 .. m]$ ,
  - sendo que  $\text{alcance}[i]$  está associado ao sufixo  $p[i .. m]$ ,
- e vamos preencher  $\text{alcance}[i]$  com
  - o menor deslocamento  $(m - q)$  entre 1 e  $m$
  - que alinha/emparelha corretamente
    - o final de  $p[1 .. m]$
    - com o final de  $p[1 .. q]$ .
- Mais formalmente,  $\text{alcance}[i] = (m - q)$ 
  - sendo  $q$  o maior índice que satisfaz
    - $p[i .. m]$  é sufixo de  $p[1 .. q]$
    - ou  $p[1 .. q]$  é sufixo de  $p[i .. m]$ .
  - Se não existe tal  $q$ , então  $\text{alcance}[i]$  deve receber o valor  $m$ .

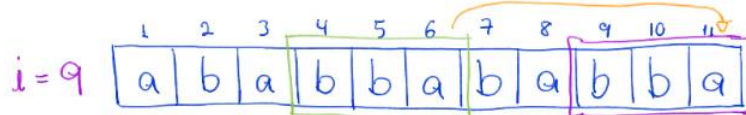
Exemplo 3:



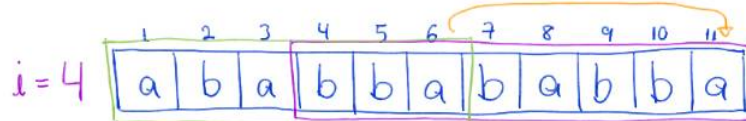
$alcance[10] = 3 = 11 - 8$



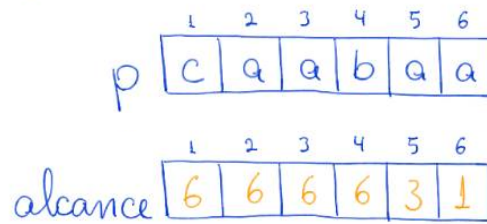
$alcance[9] = 5 = 11 - 6$



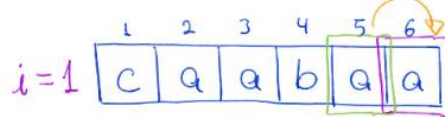
$alcance[4] = 5 = 11 - 6$



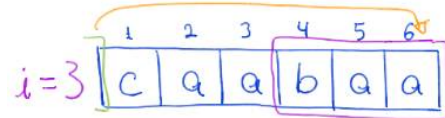
Exemplo 4:



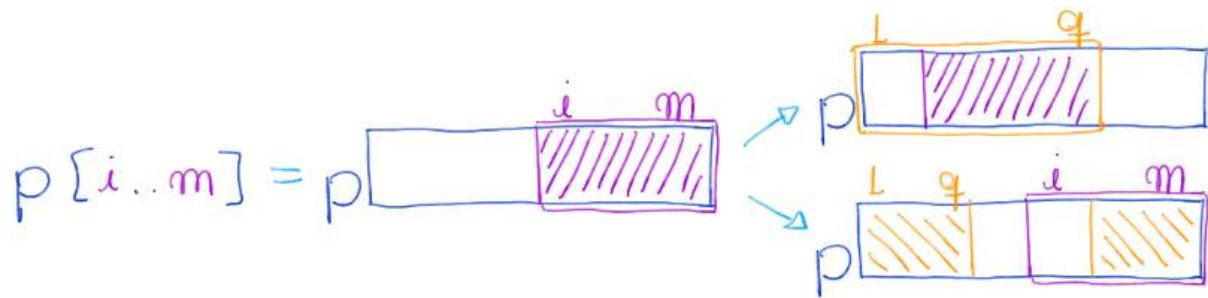
$alcance[6] = 1 = 6 - 5$



$alcance[4] = 6 = 6 - 0$



Código do pré-processamento:



```
int *preProcGoodSuff(char palavra[], int m) {
    int i, q, desl_p;
    int *alcance = malloc((m + 1) * sizeof(int));
    for (i = m; i >= 1; i--) {
        q = m - 1;
        desl_p = 0;
        // continua enquanto desl_p for menor que
        // o tamanho do sufixo palavra[i .. m]
        // e do prefixo palavra[1 .. q]
        while (desl_p < m - i + 1 && desl_p < q)
            if (palavra[m - desl_p] == palavra[q - desl_p])
                desl_p++;
            else
                q--, desl_p = 0;
        alcance[i] = m - q;
    }
    return alcance;
}
```

Eficiência de tempo:

- A fase de pré-processamento leva, no pior caso,
  - tempo proporcional ao cubo do tamanho da palavra,
    - i.e.,  $O(m^3)$ ,
  - por exemplo, quando a primeira metade da palavra
    - é composta por 'a's e a segunda metade por 'b's.
- Vale destacar que o pré-processamento pode ser melhorado
  - para levar tempo quadrático no tamanho da palavra,
    - i.e.,  $O(m^2)$ ,
  - se reaproveitarmos informações de uma iteração
    - do laço principal do pré-processamento para outra.
      - Que informações são essas?

Eficiência de espaço:

- o espaço adicional utilizado é proporcional ao tamanho da palavra,
  - i.e.,  $O(m)$ .





```

    if (desl_p >= m)
        ocorre++;
    if (desl_p == 0)
        pos_t += 1;
    else
        pos_t += alcance[m - desl_p + 1]; // por que + 1?
}
free(alcance);
return ocorre;
}

```

Invariante e corretude:

- Os invariantes principais são os mesmos do algoritmo básico.

Eficiência de tempo:

- Adicionalmente ao tempo gasto no pré-processamento, temos que
  - no pior caso ele leva tempo  $O(mn)$ , pois
    - o laço externo itera  $(n - m + 1)$  vezes
      - e o laço interno itera  $m$  vezes.
    - Um exemplo, em que ele leva tempo  $O(n^2)$ ,
      - considere o mesmo cenário do algoritmo básico,
      - o texto de tamanho  $n$  tem apenas um caractere 'x',
      - e a palavra de tamanho  $n/2$  tem o mesmo caractere 'x'.
    - No entanto,
      - o pior caso deste algoritmo é mais raro
      - e o número de comparações médio é bem menor.
  - No melhor caso,
    - o caractere texto[pos\_t] sempre casa com p[m]
      - e não aparece mais em p[1 .. m - 1],
        - ou seja, alcance[m] = m.
    - Com isso, pos\_t avança em saltos de tamanho  $m$ ,
      - e o número de comparações será da ordem de  $n / m$ .
    - Note que este valor é sublinear,
      - em relação ao tamanho do texto.

Eficiência de espaço:

- O espaço adicional é o mesmo daquele utilizado no pré-processamento,
  - i.e.,  $O(m)$ .

## Terceiro algoritmo de Boyer-Moore

Corresponde à combinação dos dois algoritmos anteriores,

- i.e., escolhendo o maior incremento para `pos_t` a cada iteração.

Código:

```
// Recebe vetores palavra[1..m] e texto[1..n],
// com m >= 1 e n >= 0, e devolve o número de
// ocorrências de palavra em texto.
int BoyerMoore(char palavra[], int m, char texto[], int n) {
    int *alcance, *dist_ult, pos_t, desl_p, ocorrencias, incr1, incr2;
    dist_ult = preProcBadCharac(palavra, m);
    alcance = preProcGoodSuff(palavra, m);
    ocorrencias = 0;
    pos_t = m;
    while (pos_t <= n) {
        desl_p = 0;
        while (desl_p < m
            && palavra[m - desl_p] == texto[pos_t - desl_p])
            desl_p++;
        if (desl_p >= m)
            ocorrencias++;
        if (pos_t == n)
            incr1 = 1;
        else
            incr1 = 1 + dist_ult[(int)texto[pos_t + 1]];
        if (desl_p == 0)
            incr2 = 1;
        else
            incr2 = alcance[m - desl_p + 1];
        pos_t += maximo(incr1, incr2);
    }
    free(dist_ult);
    free(alcance);
    return ocorrencias;
}
```