AED2 - Aula 18
Busca de palavras em um texto,
algoritmo de Boyer-Moore (good suffix heuristic)

Segundo algoritmo de Boyer-Moore

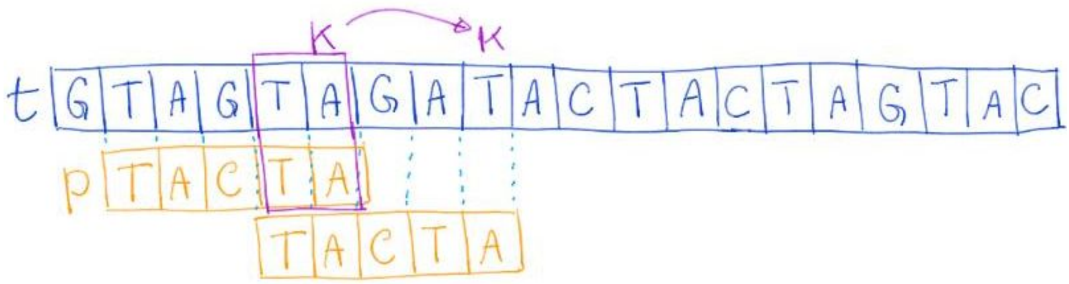
Vamos estudar a segunda heurística do algoritmo de Boyer-Moore,

- conhecida como “good suffix heuristic”.

Nos seguintes exemplos

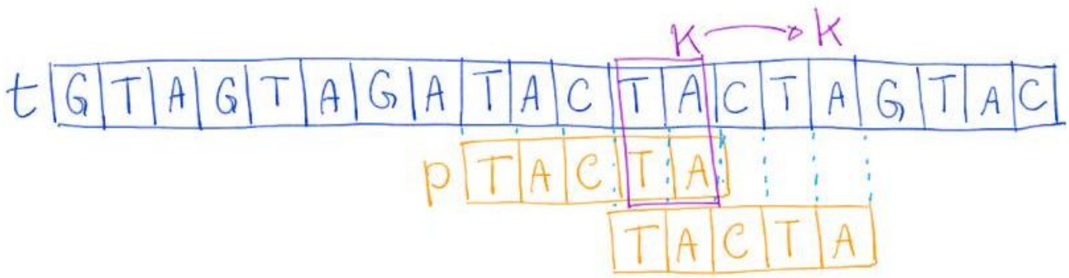
- considere que o algoritmo acabou de testar se
  - p[1 .. m] é sufixo de t[1 .. k],
- e acabou detectando que p[m - r .. m] = t[k - r .. k]

Exemplo 1:



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

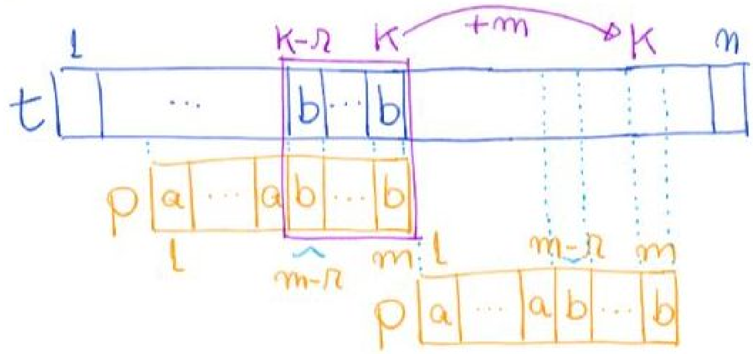
Exemplo 2:



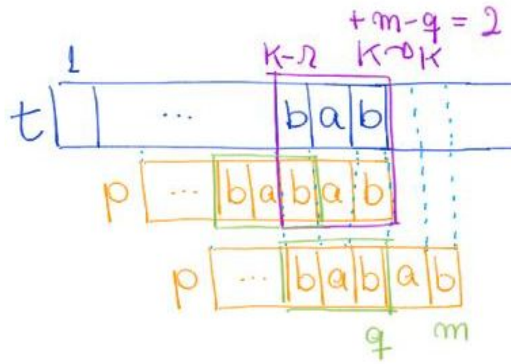
K avançou 3 posições depois de detectar que p[1..5] = t[k-5..k] = "TACTA", pois "TA" é o maior sufixo de p que também é prefixo de p, e a distância do primeiro "TA" até o final de p[1..m] é 3.

Ideia da “good suffix heuristic”:

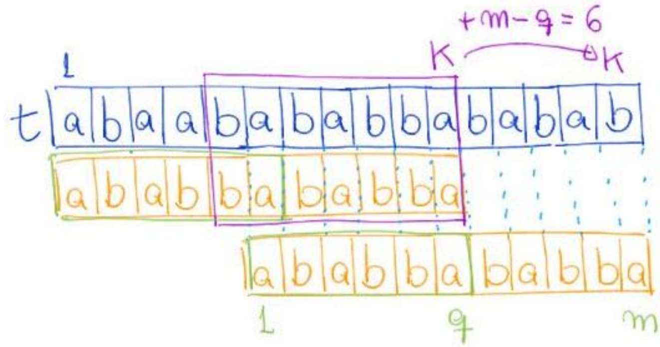
- Suponha que um sufixo de p[1 .. m] coincide com um sufixo de t[1 .. k],
    - i.e., p[m - r .. m] = t[k - r .. k], para r entre 1 e m.
- Agora, considere que a sequência no sufixo p[m - r .. m]
    - não se repete mais em p[1 .. m].
- Neste caso, sabemos que podemos avançar k
    - até que a primeira posição de p esteja depois do k atual,
        - i.e., k += m.



- Agora, considere que a sequência no sufixo p[m - r .. 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 - r .. q], com q < m.
- Neste caso, sabemos que podemos avançar k
    - até que p[q] esteja alinhado com t[k],
        - i.e., k += m - q.
- Note que, pode haver intersecção entre p[m - r .. m] e p[q - r .. q],
    - i.e., é possível ocorrer q >= m - r.



- Por fim, falta considerar um caso complementar
    - e possivelmente concomitante com os anteriores.
- Considere que a sequência p[m - r .. 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 <= r.
- Neste caso, sabemos que podemos avançar k
    - até que p[q] esteja alinhado com t[k],
        - i.e., k += m - q.

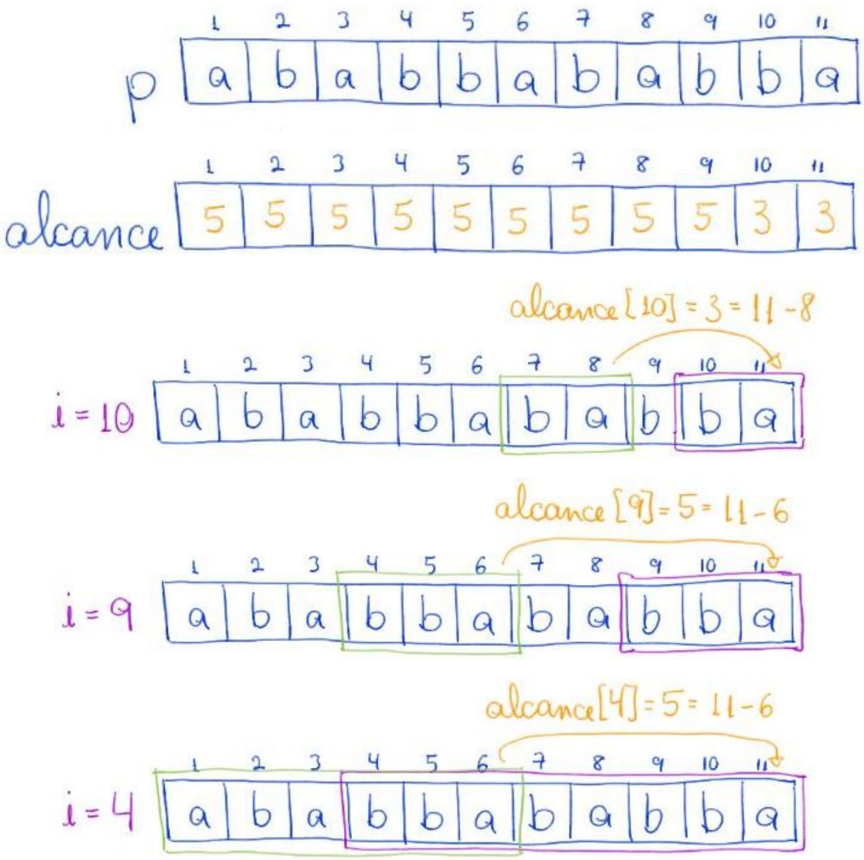


Sintetizando a ideia da "good suffix heuristic":
- para cada índice i entre 1 e m,
    - corresponde um sufixo p[i .. m].
    - Queremos encontrar o maior índice q, tal que
        - p[i .. m] é sufixo de p[1 .. q], ou seja,
            - q marca a primeira repetição de p[i .. m] em p[1 .. m].
        - 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 k,
    - precisamos fazer um pré-processamento da palavra p[1 .. m].

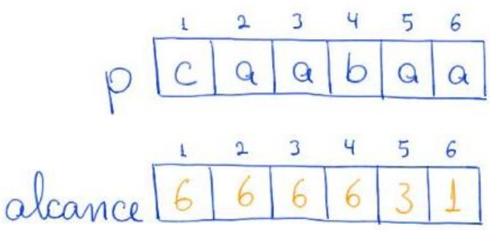
Neste pré-processamento, vamos
- alocar um vetor auxiliar alcance[1 .. m],

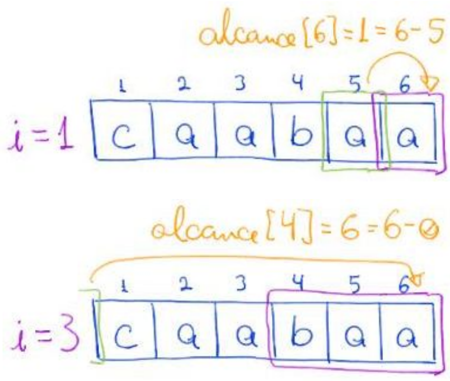
    - sendo que alcance[i] está associado ao sufixo p[i .. m],
- e vamos preencher alcance[i] com
    - o menor deslocamento (m - q) entre 1 e m
    - que alinha/emparelha corretamente
        - o final de p[1 .. m]
        - com o final de p[1 .. q].
- Mais formalmente, alcance[i] = (m - q)
    - sendo 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 alcance[i] deve receber o valor m.

Exemplo 3:



Exemplo 4:





Código do pré-processamento:

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

Eficiência de tempo:

- A fase de pré-processamento leva, no pior caso,
    - tempo proporcional ao quadrado do tamanho da palavra,
        - i.e., O(m^2).
- Vale destacar que o pré-processamento pode ser melhorado
    - para levar tempo linear no tamanho da palavra,
        - i.e., O(m).
    - O que podemos reaproveitar
        - de uma iteração do laço principal do pré-processamento
            - para outra, a fim de melhorar sua eficiência?

Eficiência de espaço:

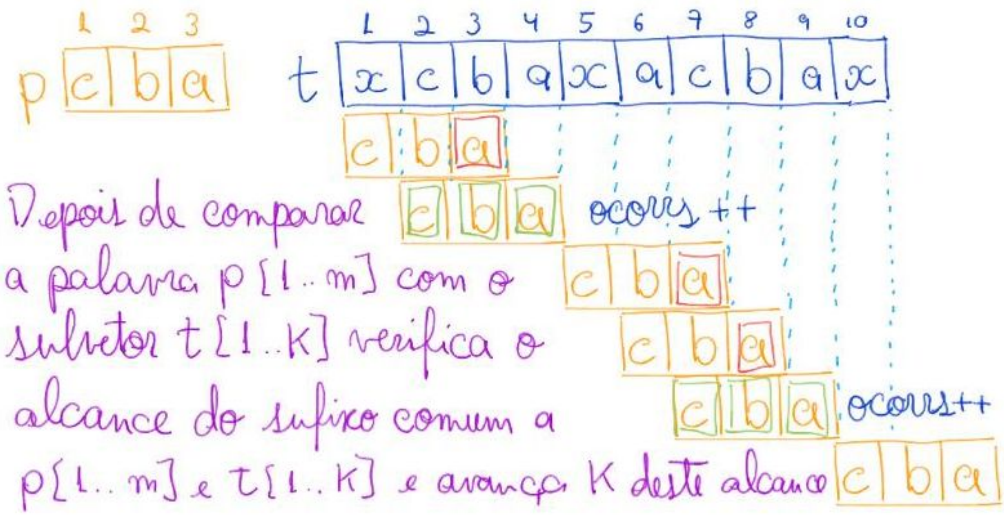
- o espaço adicional utilizado é proporcional ao tamanho da palavra,
    - i.e., O(m).

Ideia do algoritmo:

- Assim como o algoritmo anterior,
    - vamos percorrer o vetor t[1 .. n] da esquerda para a direita
    - testando em cada iteração,
        - se p[1 .. m] é sufixo de t[1 .. k].
- No entanto, antes de incrementar k para avançar no texto t,
    - vamos utilizar a “good suffix heuristic”
        - em busca de um maior incremento para k.

Exemplo 5:

- Neste exemplo vamos buscar p em t,
    - indo da esquerda para a direita,
- e saltando, a cada iteração,
    - de acordo com o deslocamento alcance[i]
        - do sufixo p[i .. m] terminado em t[k],
            - que acabamos de detectar.



Código do algoritmo:

```
// Recebe vetores p[1..m], com 1 <= m <= MAX e
// um texto t[1..n] e devolve o número de
// ocorrências de p em t.
int BoyerMoore2(char p[], int m, char t[], int n)
{
   int *alcance, k, r, ocorrs;
   // pré-processamento da palavra p
   alcance = preProcGoodSuff(p, m);
   // busca da palavra p no texto t
   ocorrs = 0;
```

```
    k = m;
    while (k <= n)
    {
        r = 0;
        // p[1..m] casa com t[k-m+1..k]?
        while (r < m && p[m - r] == t[k - r])
            r++;
        if (r >= m)
            ocorrs++;
        if (r == 0)
            k += 1;
        else
            k += alcance[m - r + 1];
    }
    free(alcance);
    return ocorrs;
}
```

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 t de tamanho n tem apenas um caractere ‘x’,
            - e a palavra p 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 t[k] sempre casa com p[m]
            - e não aparece mais em p[1 .. m - 1],
                - ou seja, alcance[m] = m.
        - Com isso, k 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.

Terceiro algoritmo de Boyer-Moore

Corresponde à combinação dos dois algoritmos anteriores,

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

Código:

```
// Recebe vetores p[1..m], com 1 <= m <= MAX e
// um texto t[1..n] e devolve o número de
// ocorrências de p em t.
int BoyerMoore(char p[], int m, char t[], int n)
{
   int *alcance, *ult, k, r, ocorrs, desloc1, desloc2;
   // pré-processamento da palavra p para "bad character heuristic"
   ult = preProcBadCharac(p, m);
   // pré-processamento da palavra p para "good suffix heuristic"
   alcance = preProcGoodSuff(p, m);
   // busca da palavra p no texto t
   ocorrs = 0;
   k = m;
   while (k <= n)
   {
       r = 0;
       // p[1..m] casa com t[k-m+1..k]?
       while (r < m && p[m - r] == t[k - r])
           r++;
       if (r >= m)
           ocorrs++;
       if (k == n)
           desloc1 = 1;
       else
           desloc1 = ult[t[k + 1]] + 1;
       if (r == 0)
           desloc2 = 1;
       else
           desloc2 = alcance[m - r + 1];
       k += maximo(desloc1, desloc2);
   }
   free(ult);
   free(alcance);
   return ocorrs;
}
```