AED2 - Aula 17
Busca de palavras em um texto,
algoritmo de Boyer-Moore (bad character heuristic)

Definição do problema

Considere o problema de encontrar todas as ocorrências de

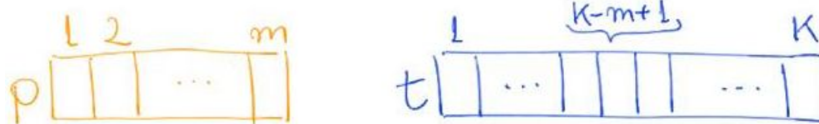
- uma sequência curta, que chamaremos de palavra,
- em uma sequência longa, que chamaremos de texto.

Este problema surge em diferentes áreas,

- como na implementação de funcionalidades em editores de texto,
- na área de biologia computacional.

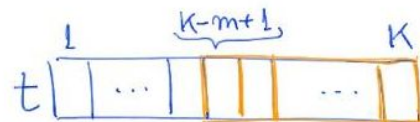
Para definir mais formalmente o problema,

- considere uma palavra p[1 .. m],
- e uma substring t[1 .. k] de um texto t[1 .. n],
    - com 1 <= k <= n.

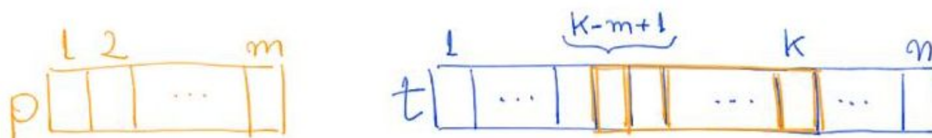


- Vamos utilizar o conceito de sufixo, definido a seguir.

$p[1..m]$ é sufixo de $t[1..K]$ se
$p[m] = t[K], \ldots, p[1] = t[K-m+1]$,
i.e., $p[i] = t[K-m+i]$, $i \in [1, m]$



- Assim, temos que

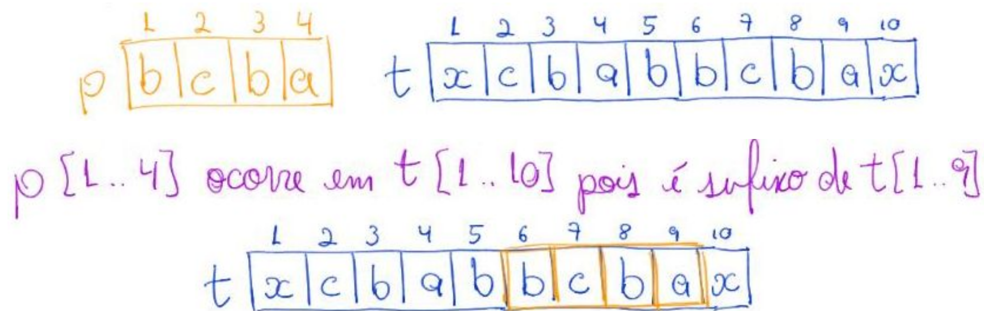


$p[1..m]$ ocorre em $t[1..n]$ se existe $K \in [m, n]$
tal que $p[1..m]$ é sufixo de $t[1..K]$

- Note que, uma palavra nunca será sufixo
    - de um texto que seja menor do que ela.
    - Por isso, k começa em m.
- Também decorre dessa observação que, se m > n
    - então o número de ocorrências de p em t é zero.

- Além disso, nossa definição não faz sentido se a palavra for vazia.
    - Por isso, supomos m >= 1.
- Observe também que, excepcionalmente,
    - vamos considerar que nossos vetores começam na posição 1.

Exemplo:



Embora estejamos interessados em localizar

- as ocorrências de uma palavra p[1 .. m] em um texto t[1 .. n],
- para simplificar, vamos tratar do problema de
    - determinar o número de ocorrências de p[1 .. m] em t[1 .. n].

Antes de apresentar nosso primeiro algoritmo, vamos definir algumas convenções:

- Nossos algoritmos vão varrer o texto t da esquerda para a direita.
    - Vale notar que a outra opção é equivalente.
- Além disso, cada vez que nossos algoritmos
    - comparam a palavra p com um subvetor de t,
        - vamos varrê-los da direita para a esquerda.
    - Em geral, as duas alternativas são equivalentes,
        - mas um dos algoritmos que veremos exige que
            - a comparação seja feita no sentido contrário
                - ao da varredura do texto.

Algoritmo básico

Ideia do algoritmo:

- Percorrer o vetor t[1 .. n] da esquerda para a direita
    - testando na iteração k, para k variando de m até n,
        - se p[1 .. m] é sufixo de t[1 .. k].
    - Para tanto,
        - comparamos cada caractere de p[1 .. m]
        - com os m últimos caracteres de t[1 .. k],
            - i.e., t[m - k + 1 .. k].

Exemplo:

Lembre que convencionamos percorrer o texto da esquerda para a direita e comparamos os caracteres da palavra com cada subvetor do texto da direita para a esquerda

Código:

```
// Recebe vetores p[1..m] e t[1..n],
// com m >= 1 e n >= 0, e devolve
// o número de ocorrências de p em t.
int basico(char p[], int m, char t[], int n)
{
   int k, r, ocorrs;
   ocorrs = 0;
   for (k = m; k <= n; k++)
   {
       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++;
   }
   return ocorrs;
}
```

Invariante e corretude:

- o invariante principal do laço externo é que
  - no início da iteração k
    - ocorrs é o número de ocorrências de p[1 .. m] em t[1 .. k - 1].
- o invariante principal do laço interno é que
  - no início da iteração r temos
    - p[m - r + 1 .. m] = t[k - r + 1 .. k].

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 pior caso.
  - Note que, quando m ~= n/2
    - o tempo no pior caso é proporcional a n^2.
  - Como exemplo, considere um texto t de tamanho n
    - com apenas um caractere ‘x’
    - e uma palavra p de tamanho n/2
      - composta inteiramente pelo mesmo caractere ‘x’
- O melhor caso do algoritmo ocorre
  - se a palavra terminar com um caractere não presente no texto.
  - Como exemplo, considere um texto t
    - com apenas um caractere ‘x’
    - e uma palavra p terminada por caractere ‘y’.
  - Neste caso o número de operações é O(n - m + 1).
- Vale notar que, se m é muito menor que n,
  - por exemplo, m = O(lg n),
    - a eficiência do algoritmo é próxima de linear.

Eficiência de espaço:

- o espaço adicional utilizado é constante.

Agora vamos estudar o algoritmo de Boyer-Moore

- que utiliza duas heurísticas
  - para melhorar a eficiência do algoritmo básico.
- Em particular, essas heurísticas utilizam critérios não triviais
  - que nos permitirão avançar o índice k no texto t
    - com passos maiores que 1 a cada iteração.

Primeiro algoritmo de Boyer-Moore

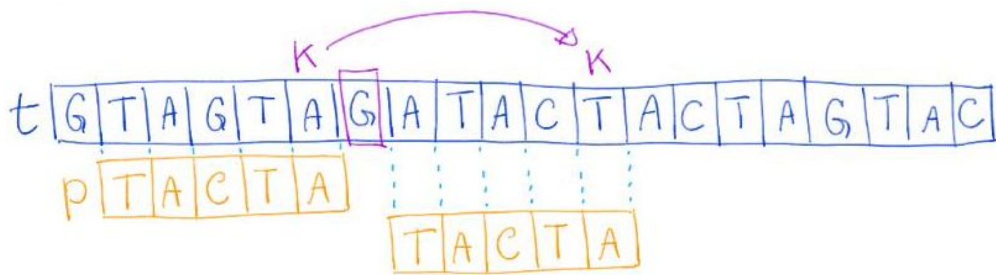
Vamos estudar a primeira heurística do algoritmo de Boyer-Moore,

- conhecida como “bad character heuristic”.

Nos seguintes exemplos

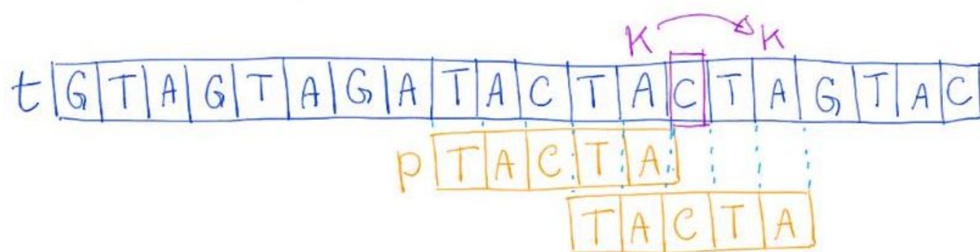
- considere que o algoritmo acabou de testar se
  - p[1 .. m] é sufixo de t[1 .. k]
- e, antes de incrementar k,
  - vai avaliar o caractere t[k + 1]

Exemplo 1:



K avançou 6 posições (1+o tamanho m da palavra p) depois de detectar o caractere 'G' na posição K+1 pois 'G' não aparece em p[1..m]

Exemplo 2:



K avançou 3 posições depois de detectar o caractere 'C' na posição K+1, pois 'C' não aparece nas duas últimas posições de p[1..m], i.e., p[m-1..m].

Ideia da “bad character heuristic”:

- calcular um incremento para k
  - de modo que t[k+1] fique emparelhado
    - com a última ocorrência do caractere t[k+1] em p[1 .. m].
- Para implementar essa ideia e automatizar os saltos do índice k, precisamos
  - conhecer o alfabeto sobre o qual estamos trabalhando,
    - i.e., o conjunto de valores que cada caractere pode assumir,
  - e fazer um pré-processamento da palavra p[1 .. m].

Neste pré-processamento, vamos:

- alocar um vetor auxiliar ult[ ] com uma posição
  - para cada caractere do alfabeto,
- e preencher ult[c] com a distância
  - da última ocorrência de ‘c’ até o final da palavra p[1 .. m].
- Mais formalmente, ult[c] é
  - o menor valor t em [0, m - 1] tal que p[m - t] = c.

- Se ‘c’ não está em p[1 .. m],
  - então ult[c] deve receber o valor m.
- Note que o incremento de k é exatamente ult[t[k + 1]] + 1.

Exemplo 3:

- Neste exemplo, vamos considerar o alfabeto das bases nitrogenadas.

alfabeto = {A, G, T, C} p |T|A|C|T|A| ult |0|5|1|2|
A G T C

ult[c] é a distância (deslocamento) da última ocorrência do caractere 'c' até o final de p[1..m]

K+ult[G]+1 = K+6

K K K+6

t |G|T|A|G|T|A|G|A|T|A|C|T|A|C|T|A|G|T|A|C|

K

K+ult[C]+1 = K+3

note que, os subvetores t[1..l], com $K < l \leq K + ult[t[K+1]]$, não podem ter p[1..m] como sufixo, já que o caractere t[K+1] estaria na posição errada em p.

Código do pré-processamento:

```
int *preProcBadCharac(char p[], int m)
{
   int i;
   int *ult = malloc(256 * sizeof(int)); // o alfabeto é 0..255
   for (i = 0; i < 256; i++)
       ult[i] = m;
   for (i = 1; i <= m; i++)
       ult[p[i]] = m - i;
   return ult;
}
```

Eficiência de tempo:

- Seja bitsdigit o número de bits usados em cada caractere.
- A fase de pré-processamento leva tempo proporcional
  - ao tamanho do alfabeto,
    - que é O(2^bitsdigit),
  - mais o tamanho da palavra m.

Eficiência de espaço:

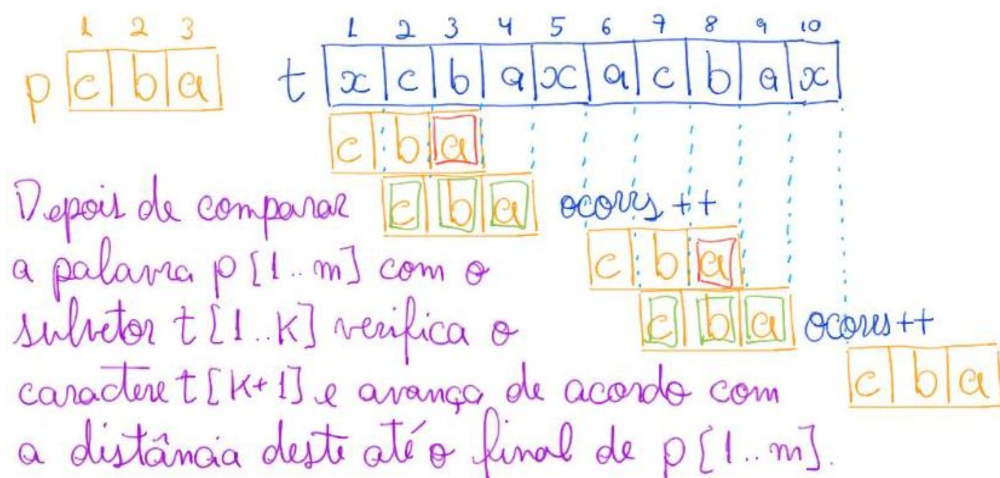
- o espaço adicional utilizado é proporcional ao tamanho do alfabeto,
  - i.e., O(2^bitsdigit).
- Será que podemos fazer melhor?
  - Particularmente, no caso em que o tamanho do alfabeto é
    - muito maior que o conjunto de caracteres distintos em p[1 .. m]?
  - Considere usar uma tabela de espalhamento (hash table).
    - Como isso pode impactar o espaço adicional
      - e o tempo do pré-processamento?

Ideia do algoritmo:

- Assim como o algoritmo básico,
  - 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 “bad character heuristic”
    - em busca de um maior incremento para k.

Exemplo 4:

- Neste exemplo vamos buscar p em t,
  - indo da esquerda para a direita,
- e saltando, a cada iteração,
  - de acordo com o deslocamento ult[ ]
    - do caractere em t[k + 1].



Código do algoritmo:

```
// Recebe vetores p[1..m] e t[1..n] de chars,
// com m >= 1 e n >= 0, e devolve o número
// de ocorrências de p em t.
```

```c
int BoyerMoore1(char p[], int m, char t[], int n)
{
    int *ult;
    int i, k, r, ocorrs;
    // pré-processamento da palavra p
    ult = preProcBadCharac(p, m);
    // busca da palavra p no texto t
    ocorrs = 0;
    k = m;
    while (k <= n)
    {
        r = 0;
        // p[1..m] casa com p[k-m+1..k]?
        while (r < m && p[m - r] == t[k - r])
            r++;
        if (r >= m)
            ocorrs++;
        if (k == n)
            k += 1;
        else
            k += ult[t[k + 1]] + 1;
    }
    free(ult);
    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
    - 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 difere de p[m]
            - e o caractere t[k + 1] sempre está ausente de p[1 .. m].
        - Com isso, k avança em saltos de tamanho m + 1,

            - 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.