

AED2 - Aula 16

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

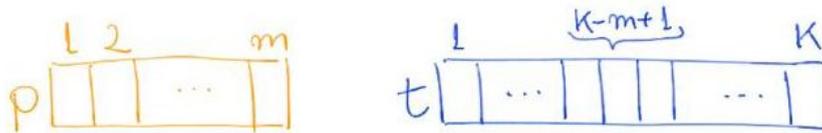
- uma sequência curta, que chamaremos de palavra,
- em uma sequência longa, que chamaremos de texto.

Este problema surge em diferentes áreas, como

- na implementação de funcionalidades em editores de texto,
- na área de biologia computacional,
- e na busca de informações na WEB.

Para definir mais formalmente o problema,

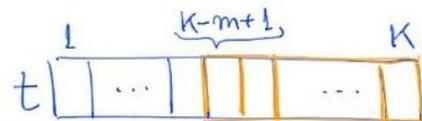
- considere uma palavra $p[1 .. m]$,
 - e uma substring $t[1 .. k]$ de um texto $t[1 .. n]$,
 - com $1 \leq k \leq n$.



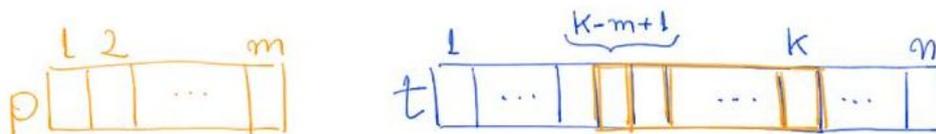
- Observe que, excepcionalmente, nossos vetores
 - começam na posição 1 e terminam na posição tamanho do vetor.

Vamos utilizar o conceito de sufixo, definido a seguir.

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



- Assim, temos que

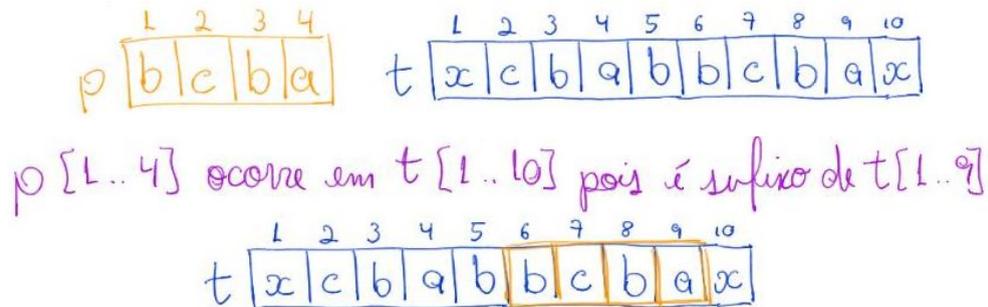


$p[1..m]$ ocorre em $t[1..n]$ se existe $k \in [m, n]$
 tal que $p[1..m]$ é sufixo de $t[1..k]$

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 $m \geq 1$.

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 da esquerda para a direita.
 - Vale notar que a outra opção é equivalente.
- Além disso, cada vez que nossos algoritmos
 - comparam a palavra com um subvetor do texto,
 - 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.

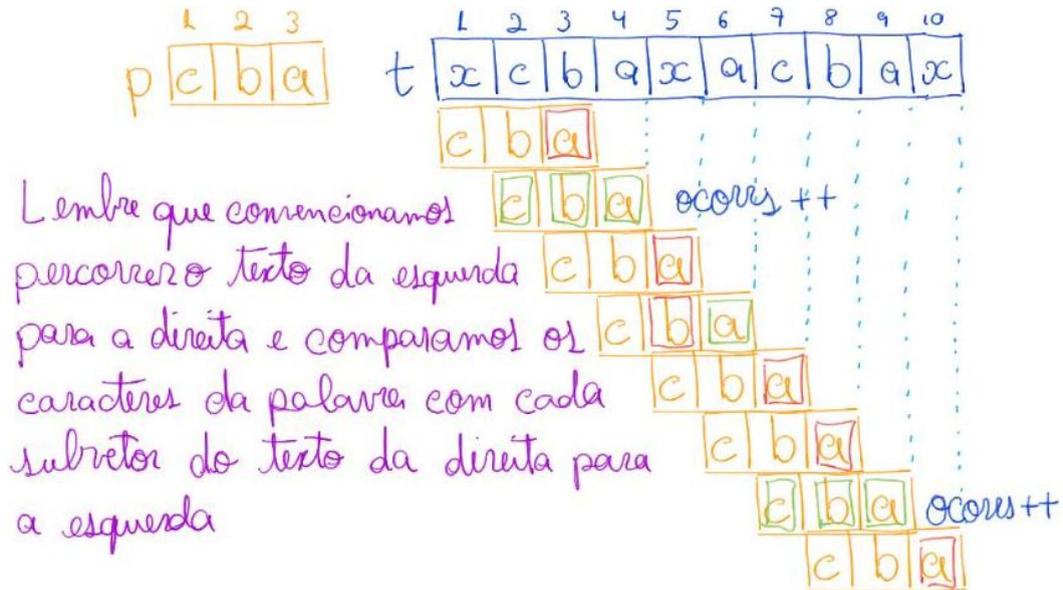
Algoritmo básico

Ideia do algoritmo:

- Percorrer o texto $t[1 .. n]$ da esquerda para a direita
 - testando na iteração pos_t , para pos_t variando de m até n ,
 - se a palavra $p[1 .. m]$ é sufixo de $t[1 .. pos_t]$.
 - Para tanto,

- comparamos cada caractere de $p[1 .. m]$
- com os m últimos caracteres de $t[1 .. pos_t]$,
 - i.e., $t[pos_t - m + 1 .. pos_t]$.

Exemplo:



Código:

```
// Recebe vetores palavra[1..m] e texto[1..n],
// com  $m \geq 1$  e  $n \geq 0$ , e devolve o número de
// ocorrências de palavra em texto.
int basico(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;
}
```

Invariante e corretude:

- O invariante principal do laço externo é que no início da iteração pos_t
 - $ocorr$ s é o número de ocorrências
 - de $palavra[1 .. m]$ em $texto[1 .. pos_t - 1]$,
 - i.e, o número de ocorrências da palavra no prefixo já foi contado.
- O invariante principal do laço interno é que no início da iteração $desl_p$
 - $palavra[m - desl_p + 1 .. m] = texto[pos_t - desl_p + 1 .. pos_t]$,
 - i.e, o sufixo da palavra coincide com o sufixo da substring do texto.

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 $m \approx n / 2$
 - o tempo no pior caso é proporcional a n^2 .
 - Como exemplo, considere um texto de tamanho n
 - com apenas um caractere 'x' repetido
 - 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
 - com apenas um caractere 'x' repetido
 - e uma palavra terminada pelo 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 pos_t no texto
 - com passos maiores que 1 a cada iteração.

- fique emparelhado com a ocorrência mais a direita
 - do caractere $\text{texto}[\text{pos}_t + 1]$ em $\text{palavra}[1 .. m]$.
 - Para implementar essa ideia e automatizar os saltos do índice pos_t ,
 - 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 $\text{dist_ult}[]$ com uma posição
 - para cada caractere do alfabeto,
- e preencher $\text{dist_ult}[c]$ com a distância
 - da última ocorrência de 'c' até a última posição da palavra $p[1 .. m]$.
- Mais formalmente, $\text{dist_ult}[c]$ é
 - o menor valor desl em $[0, m - 1]$ tal que $p[m - \text{desl}] = c$.
- Se 'c' não está em $p[1 .. m]$,
 - então $\text{dist_ult}[c]$ deve receber o valor m .
- Note que, o incremento de pos_t é exatamente $1 + \text{dist_ult}[t[\text{pos}_t + 1]]$.

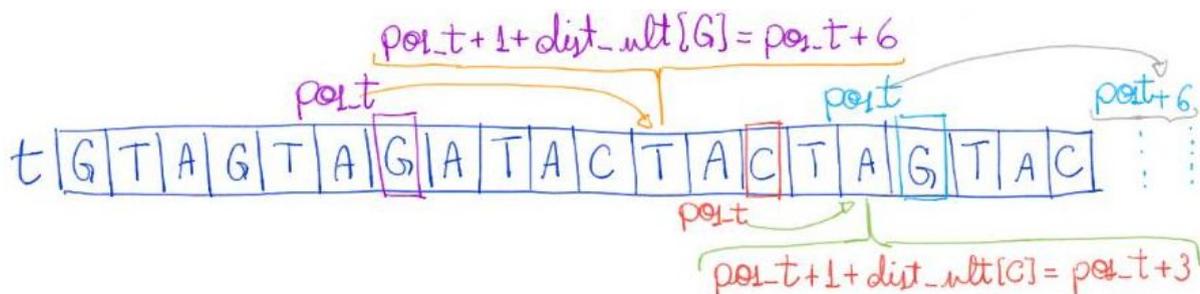
Exemplo 3:

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

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

A G T C

$\text{dist_ult}[c]$ é a distância (deslocamento) da última ocorrência do caractere 'c' até o final de $p[1..m]$



note que, os subvetores $t[1..l]$, com $\text{pos}_t < l \leq \text{pos}_t + \text{dist_ult}[t[\text{pos}_t + 1]]$, não podem ter $p[1..m]$ como sufixo, pois o caractere $t[\text{pos}_t + 1]$ alinharia com o caractere errado na palavra p .

Código do pré-processamento:

```
int *preProcBadCharac(char palavra[], int m)
{
    int i;
    int *dist_ult = malloc(256 * sizeof(int)); // o alfabeto é
0..255
    for (i = 0; i < 256; i++)
        dist_ult[i] = m;
    for (i = 1; i <= m; i++) // ordem do laço importa?
        dist_ult[(int)palavra[i]] = m - i;
    return dist_ult;
}
```

Invariante e corretude:

- O invariante principal do segundo laço é que no início da i -ésima iteração
 - `dist_ult[]` tem a distância da ocorrência de cada caractere
 - em `palavra[1 .. i - 1]` até o fim da `palavra[1 .. m]`.

Eficiência de tempo:

- Seja `bitsDigito` o número de bits usados em cada caractere.
- A fase de pré-processamento leva tempo proporcional
 - ao tamanho do alfabeto,
 - que é $O(2^{\text{bitsDigito}})$,
 - mais o tamanho da palavra m ,
- i.e., $O(2^{\text{bitsDigito}} + m)$.

Eficiência de espaço:

- O espaço adicional utilizado é proporcional ao tamanho do alfabeto,
 - i.e., $O(2^{\text{bitsDigito}})$, por conta do vetor auxiliar `dist_ult`.
- 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?
 - Ambos podem cair para $O(m)$. Por que?

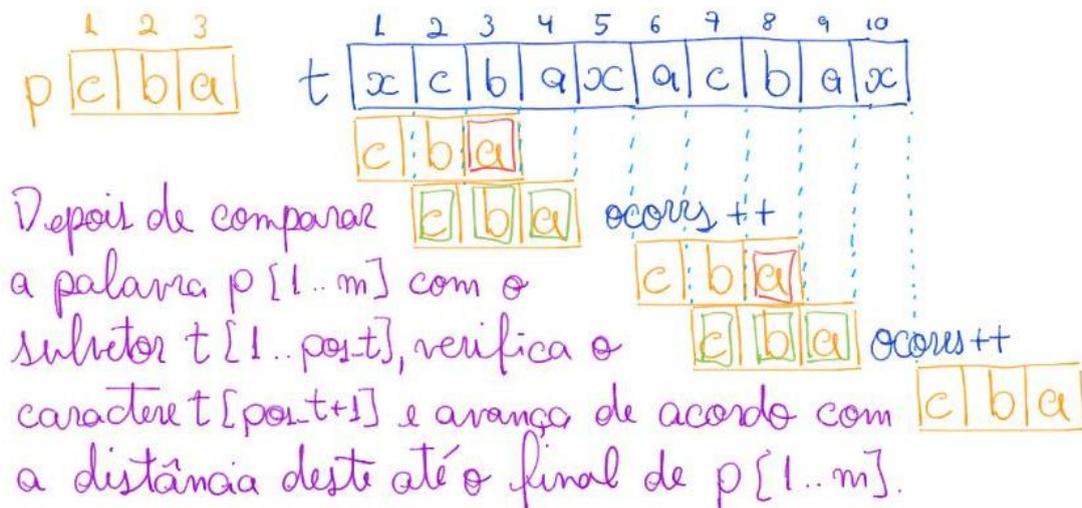
Ideia do algoritmo:

- Assim como no algoritmo básico,

- vamos percorrer o vetor texto[1 .. n] da esquerda para a direita
- testando em cada iteração pos_t,
 - se palavra[1 .. m] é sufixo de texto[1 .. pos_t].
- No entanto, antes de incrementar pos_t para avançar no texto,
 - vamos utilizar a “bad character heuristic”
 - em busca de um maior incremento para pos_t.

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 dist_ult[]
 - do caractere em t[pos_t + 1].



Código do algoritmo:

```
// 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 BoyerMoore1(char palavra[], int m, char texto[], int n)
{
    int *dist_ult;
    int pos_t, desl_p, ocorrencias;
    // pré-processamento da palavra p
    dist_ult = preProcBadCharac(palavra, m);
    // busca da palavra p no texto t
    ocorrencias = 0;
    pos_t = m;
```

```

while (pos_t <= n)
{
    desl_p = 0;
    // palavra[1..m] casa com p[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)
        ocorre++;
    if (pos_t == n) // para evitar acessar posição inválida de
memória
        pos_t += 1;
    else
        pos_t += 1 + dist_ult[(int)texto[pos_t + 1]];
}
free(dist_ult);
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
 - 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 difere de palavra[m]
 - e o caractere texto[pos_t + 1]
 - sempre está ausente de p[1 .. m].
 - Com isso, pos_t avança em saltos de tamanho $m + 1$,

- e o número de comparações é da ordem de n / m ,
 - i.e., $O(n / m)$;
- 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.