

Análisis y Diseño de Algoritmos II – Apunte String Matching

- String Matching:
Utilizado en diversidad de problemas que van desde encontrar un patrón de caracteres en un texto hasta buscar patrones dentro de una secuencia de ADN.
- Notación:
texto $\rightarrow T[0..n-1]$ patrón $\rightarrow P[0..m-1]$
- Se dice que existe un matching entre P y T si existe un desplazamiento s , $0 \leq s \leq n-m$, tal que $T[s..m-1] = P[0..m-1]$.
- Ejemplo:
texto T \rightarrow a b a b a b c a b c b b a b c a
patrón P \rightarrow a b c a
matching en: 4 y 12.
- $\Sigma \rightarrow$ Alfabeto
- Se dice que “w es prefijo de x” sii $x = w.y$ $y \in \Sigma^*$
- Se dice que “w es sufijo de x” sii $x = y.w$ $y \in \Sigma^*$
- Vamos a analizar:
 - Algoritmo *obvio* con costo $O((n - m + 1).m)$.(utiliza fuerza bruta).
 - Algoritmo *obvio mejorado* con costo $O(n.m)$.
 - Algoritmo de *Rabin Karp* con un costo de $O((n - m + 1).m)$.
 - Algoritmo del *Automata Finito* con un costo de $O(n)$ + armar el AF con un costo aproximado de $O(m^3.\Sigma)$.
 - Algoritmo de *Knuth Morris Pratt* con un costo aproximado de $O(n + m)$.
 - Algoritmo de *Boyer Moore* con un costo aproximado de $O((n - m + 1).m + \Sigma)$.

➤ **Algoritmo String Matcher intuitivo (obvio):**

```
bool Hay_Match(char T[N], char P[M], int s){
    int pos_patron = 0;
    int pos_texto = s;
    while (P[pos_patron] != '\0'){
        if(P[pos_patron] != T[pos_texto])
            return 0;
        pos_texto++;
        pos_patron++;
    }
    return 1;
}

void String_Matcher(char T[N], char P[M]){
    for(int s = 0; (T[s] != '\0') && (s <= N - M);s++)
        if(T[s] == P[0])//verifico si el primer caracter coincide, no es necesario.
            if(Hay_Match(T,P,s)){//puedo llamar directamente a la funcion Hay_Match.
                cout<<"Match en: "<<s<<endl;
            }
    }
```

Este algoritmo es el que uno piensa naturalmente, posee un costo de computacional de $O((n - m + 1).m)$ en el peor de los casos.

➤ **Una forma mas eficiente es la siguiente (obvio mejorado):**

```
void String_Matcher(char T[N], char P[M]){
    int s = 0;
    int p = 0;
    while(s < (N-M)){
        if(p == M){
            cout<<"Match en: "<<s-M<<endl;
            p = 0;
        }
        if(P[p] == T[s])
            p++;
        else
            p = 0;
        s++;
    }
}
```

➤ Este algoritmo posee un costo de $O(n-m)$.

➤ **Algoritmo Rabin Karp Matcher:**

Este algoritmo, utiliza la matemática teórica para comparar 2 números en modulo de un tercero. Para esto asume que cada carácter del texto es un numero entre 0 y 9.

➤ Notación:

texto $\rightarrow T[1..n]$ \rightarrow Valor decimal del texto del mismo tamaño que el patrón t_s .

patrón $\rightarrow P[1..m]$ \rightarrow Valor decimal del patrón es p .

$d \rightarrow$ cantidad de números en el alfabeto. En general $\{0, 1, \dots, d-1\}$

$q \rightarrow$ Numero primo con el que calculo el modulo para hacer las comparaciones.

➤ Hay que tener en cuenta que si $t_s \equiv p \pmod{q}$ no significa que haya un matching, hay que comparar carácter por carácter. Pero si $t_s \not\equiv p \pmod{q}$ entonces definitivamente no hay matching.

➤ Rabin Karp Matcher posee un costo computacional de $O((n - m + 1).m)$, aunque generalmente tiene una mejor complejidad.

```
int pow(int a, int b){
    int valor = 1;
    for(int i = 1; i <= b; i++)
        valor = valor * a;
    return valor;
}

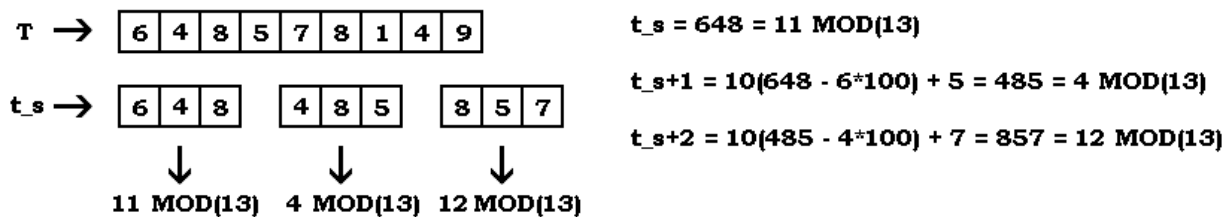
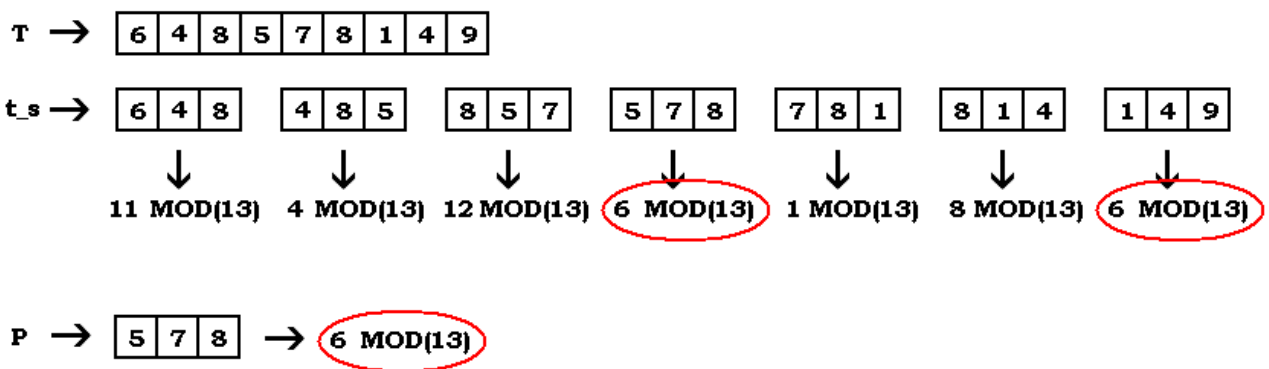
int mod(int q, int numero){
    int division = numero / q;
    return (numero - division);
}

bool Hay_Match(char T[N], char P[M], int s){
    int pos_patron = 0;
    int pos_texto = s;
    while (P[pos_patron] != '\0'){
        if(P[pos_patron] != T[pos_texto])
            return 0;
        pos_texto++;
        pos_patron++;
    }
    return 1;
}

int val(char v){
    char valor[2];
    valor[0] = v;
    valor[1] = '\0';
    return atoi(valor);
}
```

```

void Rabin_Karp_Matcher(char T[N], char P[M], int q, int d){
    int h = pow(d,m-1); //pow eleva d a la m-1
    h = mod(q,h);
    int p = 0;
    int t_s = 0;
    for(int i = 0; i < M; i++){
        p = mod(q, (d*p + val(P[i]))); //es el numero patron en mod q
        t_s = mod(q, (d*t_s + val(T[i]))); //es la primera valuacion del texto en mod q
    }
    for(int s = 0; s < N - M; s++){
        if(p == t_s)
            if(Hay_Match(T,P,s))
                cout<<"Match en: "<<s<<endl;
        t_s = mod(q, (d*(t_s - val(T[s])*h) + val(T[s+M]))); //es la siguiente valuacion
    }
} //este algoritmo utiliza la regla de Horner
    
```



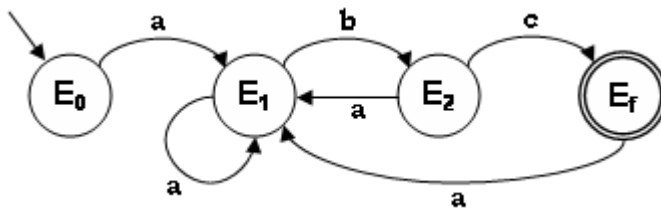
Aquí vemos claramente que $578 \equiv 149 \pmod{13}$ pero $578 \neq 149$.
 Es por eso que cuando encontramos que $t_s = p$, también se debe comparar todo el patrón para poder afirmar que hay un matching.

➤ **Algoritmo Autómata Finito Matcher:**

Este algoritmo se basa en la utilización de un autómata para encontrar un patrón. A medida que vamos leyendo caracteres, vamos avanzando de estado, y una vez que llegamos a un estado final significa que hubo matching.

$$AF = \langle Q, Q_0, \Sigma, d, F \rangle$$

Ejemplo para reconocer el patrón $P = a b c$.



Supongamos

$T \rightarrow a b a a b c a a$

$E \rightarrow E_1 E_2 E_1 E_1 E_2 E_f E_1 E_1$

Cuando llegamos a E_f significa que tenemos un matching.

Para generar el autómata podemos utilizar una matriz de transiciones, en el caso del autómata anterior la matriz sería la siguiente:

estado	alfabeto		
	a	b	c
E_0	E_1	E_0	E_0
E_1	E_1	E_2	E_0
E_2	E_1	E_0	E_f
E_f	E_1	E_0	E_0

La clase autómata dada la cadena, patrón, genera el autómata correspondiente. Entonces el código AF Matcher es el siguiente:

```

void AF_Matcher(automata AF, char T[N]){
    AF.set_estado_inicial();
    for (int pos_text = 0; pos_text < N; pos_text++){
        if(AF.estado_final())
            cout<<"Match en: "<<pos_text-M<<endl;//M tamaño del patron
        else
            AF.transicion(T[pos_text]);
    }
}
    
```

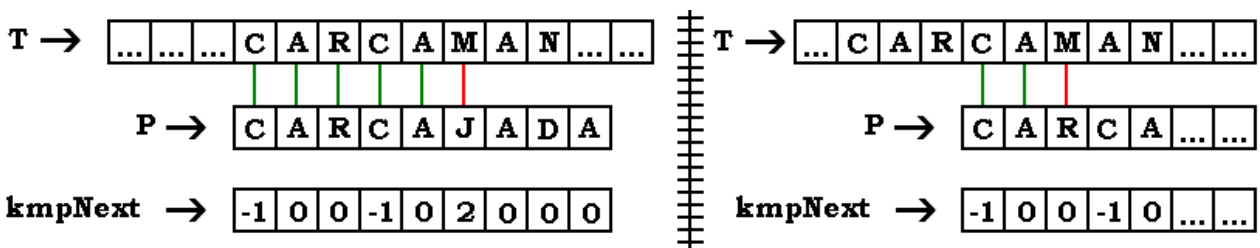
- AF Matcher posee un costo computacional de $O(n)$, esto no incluye el tiempo que demora hacer el autómata. Que lleva un costo de $O(m^3 \cdot \Sigma)$

➤ **Algoritmo Knuth Morris Pratt Matcher:**

Este algoritmo, utiliza un arreglo auxiliar `kmpNext[0..m-1]` el cual es generado con un costo de $O(m)$. Este arreglo brinda información adicional sobre el patrón a la hora de hacer las comparaciones, permite saber el corrimiento que hay que hacer en el patrón hasta la próxima comparación evitando preguntas innecesarias. Generando así un algoritmo eficiente de costo $O(n + m)$.

```
void preKMP(char P[M], int kmpNext[M]) {
    int i = 0;
    int j = -1;
    kmpNext[0] = j;
    while(i < M) {
        while(j > -1 && P[i] != P[j])
            j = kmpNext[j];
        i++;
        j++;
        if(P[i] == P[j])
            kmpNext[i] = kmpNext[j];
        else
            kmpNext[i] = j;
    }
}

void Knuth_Morris_Pratt_Matcher(char T[N], char P[M]) {
    int i = 0;
    int j = 0;
    int kmpNext[M];
    preKMP(P, kmpNext);
    while(j < N-M) {
        while(i > -1 && P[i] != T[j])
            i = kmpNext[i];
        i++;
        j++;
        if(i >= M) {
            cout<<"Match en: "<<(j - i)<<endl;
            i = kmpNext[i - 1];
        }
    }
}
```



➤ Cuando hay diferencia entre el texto y el patrón, `kmpNext`, nos dice hasta donde podemos correr el patrón, sin tener que volver a comparar los

caracteres desde el principio, en el ejemplo $M \neq J$, y `kmpNext[5]` nos dice que debemos correr el patrón hasta `P[2]` y seguir comparando desde ahí, porque nos asegura que los anteriores ya coinciden.

➤ **Algoritmo Boyer Moore Matcher:**

Este algoritmo, es utilizado cuando cuando el patrón es muy largo o se trata de buscar un texto dentro de otro texto. La novedad de este algoritmo es que las comparaciones se hacen de derecha a izquierda, así si el ultimo elemento del patrón no coincide y ademas el carácter del texto no se encuentra en el patrón, este se puede correr m posiciones sin tener que realizar ninguna otra comparación.

```
void preBM(char P[M], int bmNext[]) {
    for(int i = 0; i <= 255; i++)
        bmNext[i] = M;
    for(int i = 0; i < M; i++)
        bmNext[P[i]] = M - 1 - i ;
}

void Boyer_Moore_Matcher(char T[N], char P[M]) {
    int i = M - 1;
    int j = M - 1;
    int bmNext[255]; //255 para tener todo el codigo ascci
    preBM(P, bmNext);
    while((i < N) && (j >= 0)) {
        if(T[i] == P[j]) {
            i--;
            j--;
        }
        else {
            i += bmNext[T[i]];
            j = M - 1;
        }
        if(j < 0) {
            cout<<"Match en: "<<(i + 1)<<endl;
            i += M + 1;
            j = M - 1;
        }
    }
}
```

- `bmNext` es similar al `kmpnext` ya que nos dice hasta donde tenemos que correr el patrón. Y en caso de que el carácter leído del texto no se encuentre en el patrón, `bmNext` nos dice que avancemos en el texto, las m posiciones del patrón.