KMP অ্যালগোরিদম

Posted on September 8, 2016 by Sudip Sarker

আমার এক কিপটে বন্ধু একবার আমার সাথে বাজি ধরল, বলল, তোরা তো সবাই বলিস, আমি কোনদিন খাওয়াই না। আচ্ছা, তুই যদি ‘Lord of the Rings’ বইয়ে কতবার ‘and’ শব্দটা আছে, সেটা আমাকে বলতে পারিস, তাহলে তোকে আমি একদিন রেডিসনে বুফে খাওয়াব!

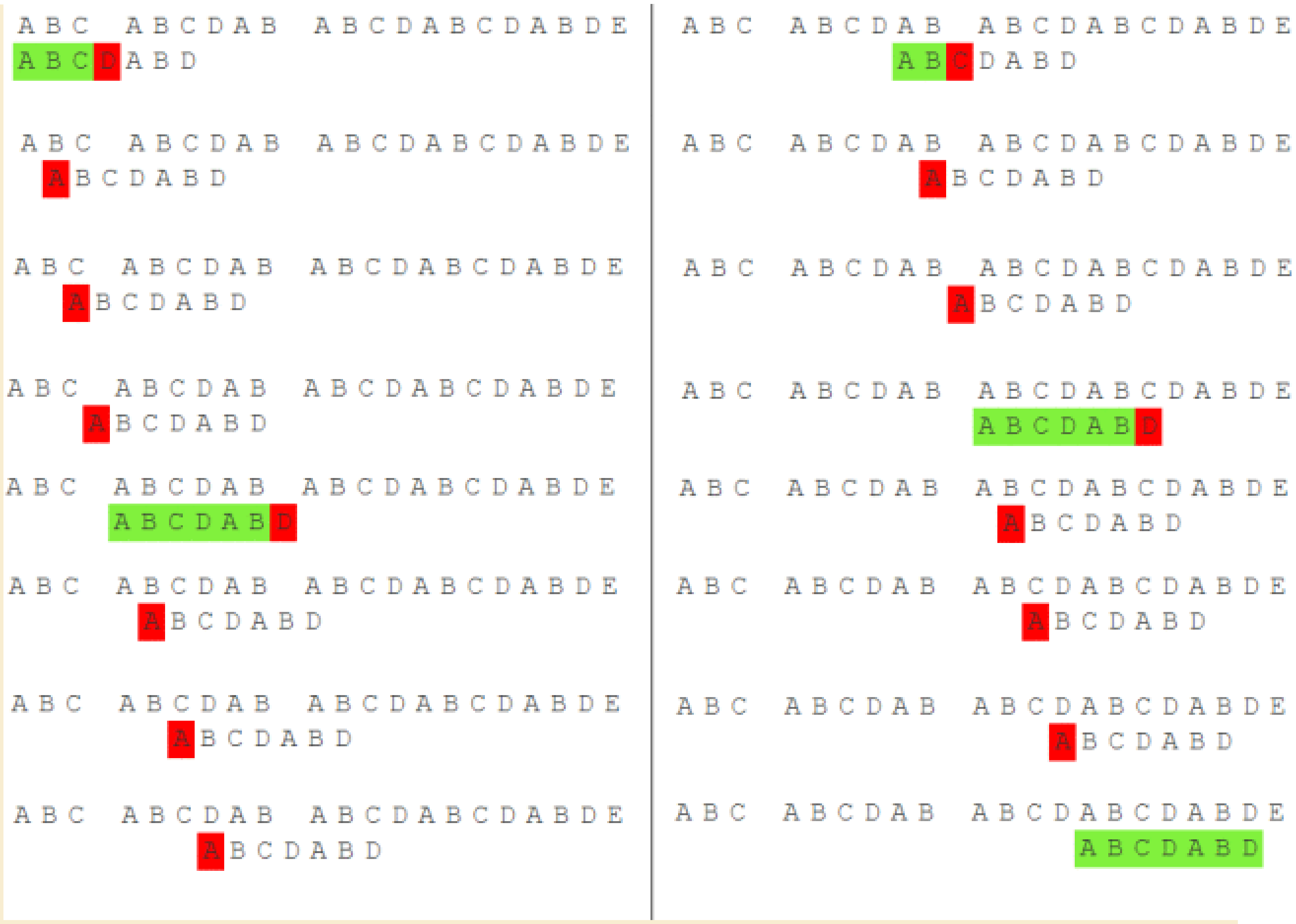
আমি তো বুফে খাওয়ার লোভে একেবারে আদাজল খেয়ে লেগে পড়লাম! কিন্তু কিছুক্ষণ পরেই বুঝতে পারলাম, হার মেনে নেয়া ছাড়া আমার আর কোন গতি নেই, কারণ এত বড় বইয়ে এত্ত ছোট্ট একটা শব্দ খুঁজে বের করা তো একগাদা খড়ে একটা ছোট্ট সুঁই খুঁজে বের করার মত (Needle in a haystack)

তবে আমিও তো হাল ছাড়বার পাত্র নই! খুঁজেটুজে নেট থেকে Lord of the Rings এর একটা pdf বের করলাম, তারপর Ctrl+f চেপে সার্চ বক্সে ’and’ লিখে দিলাম! আর জাদুর মত কম্পিউটার আমাকে বলে দিল ওই বইটাতে ঠিক কতগুলো ‘and’ আছে।

আমার বুফে খাওয়া তো নিশ্চিত হয়ে গেল কিন্তু সাথেসাথেই আমার মাথায় একটা প্রশ্ন আসল, কম্পিউটার এত তাড়াতাড়ি কতগুলো and আছে খুঁজে বের করে ফেলল কিভাবে!

তোমরা তো সবাই জানোই, কম্পিউটার নিজে থেকে কোন কাজ করতে পারে না, কিন্তু তাকে কোন কাজ শিখিয়ে দিলে সে খুব দ্রুত সেই কাজটা করে ফেলতে পারে! এখানে কম্পিউটারকে যে কাজটা শিখিয়ে দেয়া হয়েছে, সেটা হচ্ছে **String Matching Algorithm**. এই স্ট্রিং ম্যাচিং প্রব্লেমে তোমাকে একটা text ও একটা pattern দেয়া থাকবে, তোমাকে বলতে হবে, প্যাটার্নটা টেক্সটের মধ্যে আছে কিনা কিংবা থাকলে কতবার আছে।

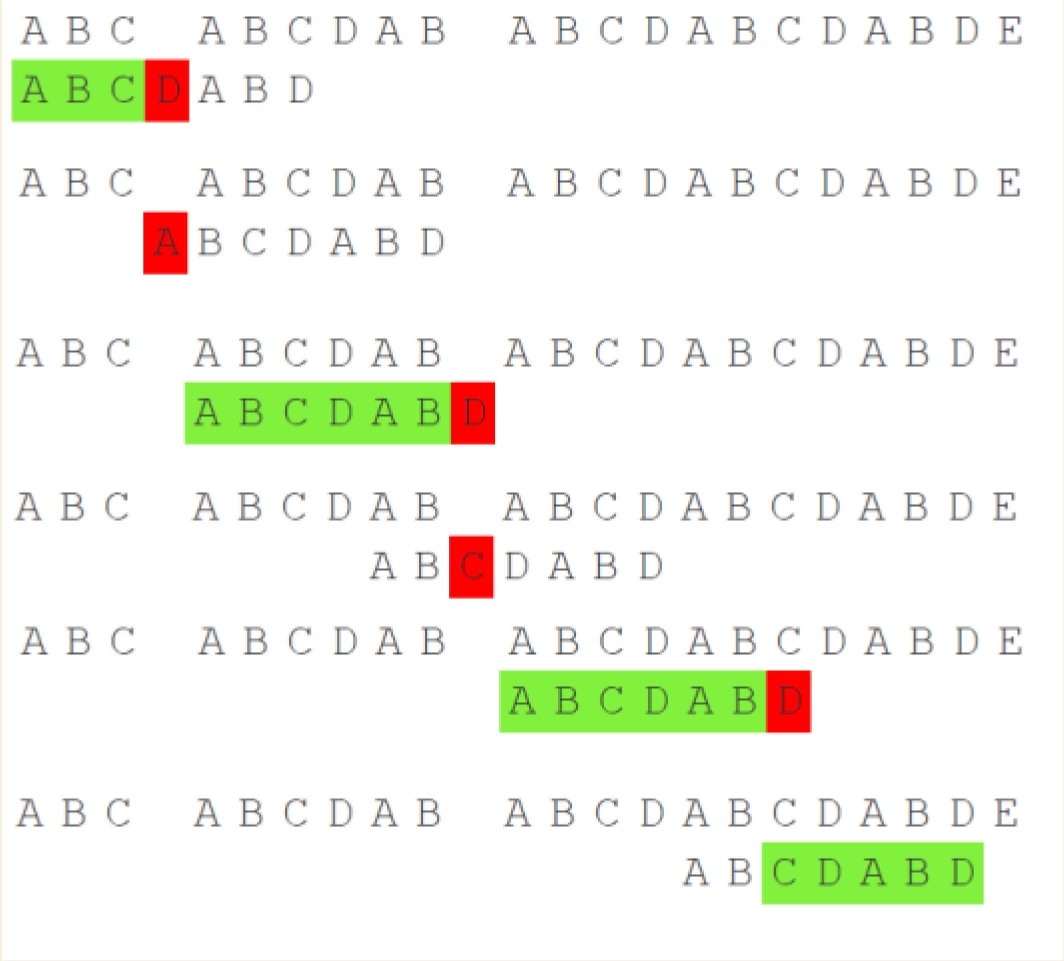
আচ্ছা, প্রথমে ভেবে দেখো তো, এই কাজটা হাতে কলমে করতে দিলে তুমি কি করতে? হ্যা, তুমি একদম শুরুর অক্ষরের সাথে প্যাটার্নের প্রথম অক্ষর মেলাতে, এরপর না মিললে ২য় অক্ষর, এরপর ৩য়। এটা হচ্ছে একদম ব্রুটফোর্স অ্যালগোরিদম!



ছবিটা দেখে নিশ্চয়ই বুঝতে পারছো, ব্রুটফোর্স এলগো খুব একটা এফিশিয়েন্ট না, অনেক বড় সাইজের টেক্সট ও প্যাটার্নের ক্ষেত্রে এটা অনেক ধীরে কাজ করবে। এটার টাইম কমপ্লেক্সিটি হচ্ছে O(n*m) যেখানে n = text length, m = pattern length.

Donald Knuth, Vaughan Pratt, James H. Morris 1977 সালে স্ট্রিং ম্যাচিংয়ের জন্যে Knuth-Pratt-Morris বা সংক্ষেপে KMP অ্যালগোরিদমটি পাবলিশ করেন যা ব্রুটফোর্সের থেকে অনেক দ্রুত কাজ করে।

KMP অ্যালগোরিদম ব্রুটফোর্সের বেসিক যে সমস্যা, সেটা দূর করে টাইম অপ্টিমাইজ করার চেষ্টা করে। ব্রুটফোর্সে আমরা যে প্রতিটা অক্ষর থেকে আলাদা আলাদাভাবে প্যাটার্ন ম্যাচিংয়ের চেষ্টা করি, KMP সেখানে কিছু অক্ষর স্কিপ করার চেষ্টা করে। এটাই KMP র বেসিক আইডিয়া।



```
A B C   A B C D A B   A B C D A B C D A B D E
A B C D A B D

A B C   A B C D A B   A B C D A B C D A B D E
      A B C D A B D

A B C   A B C D A B   A B C D A B C D A B D E
        A B C D A B D

A B C   A B C D A B   A B C D A B C D A B D E
                A B C D A B D

A B C   A B C D A B   A B C D A B C D A B D E
                      A B C D A B D

A B C   A B C D A B   A B C D A B C D A B D E
                              A B C D A B D
```

দেখো, KMP অ্যালগোরিদমের মাধ্যমে সেইম টেক্সট এ সেইম প্যাটার্ন খুঁজে বের করতে কত কম স্টেপ লাগল! ৫ নম্বর স্টেপে দেখো, ABCDABD এর শুধু D মেলেনি, এখন যেহেতু শুরুর দিকে থাকা “AB” শেষেও আছে, সেক্ষেত্রে ঐটা কিন্তু আমরা এক অর্থে মিলিয়েই ফেলেছি। তাই আমরা পরবর্তী স্টেপে প্যাটার্নে AB স্কিপ করে C থেকে সার্চ করতে পারি ৬ নম্বর স্টেপে আর টেক্সটেও আমরা ঠিক এর পরের ক্যারেক্টার থেকেই প্যাটার্ন ম্যাচিং শুরু করতে পারছি আবার পেছনে না গিয়ে। কি, বুঝতে কষ্ট

হচ্ছে?! সমস্যা নেই, আমরা এটা নিয়ে আরও বিস্তারিত আলোচনা করব।

এই স্কিপ করার কাজটা তাহলে কিভাবে করা যায়?! সেটার জন্যে আমাদের প্রথমে কিছু জিনিস সম্পর্কে জানতে হবেঃ

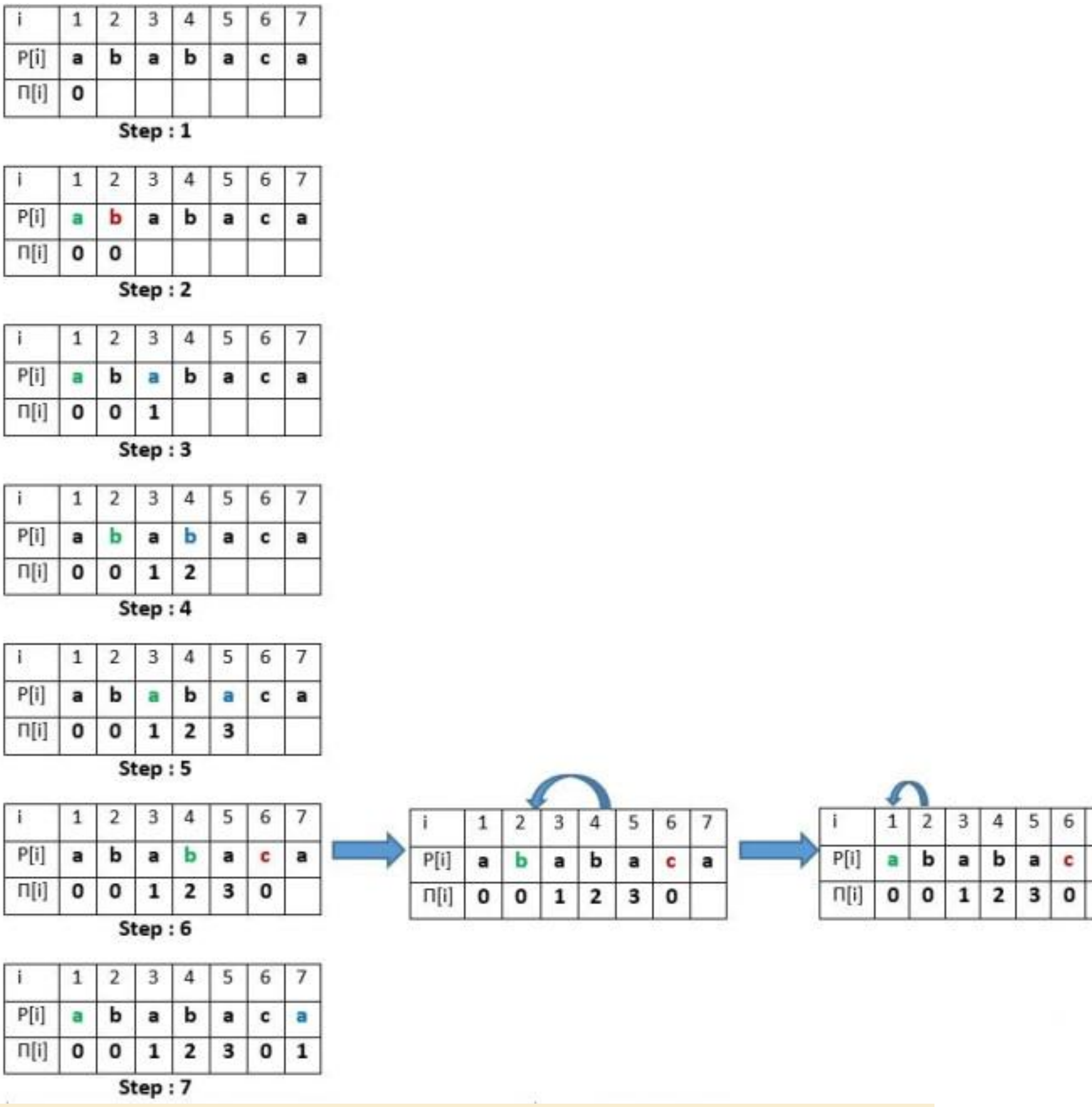
Proper prefix : কোন একটা শব্দের শেষের এক বা একাধিক অক্ষর বাদ দিয়ে যেই substring গুলো পাব, সেগুলোই proper prefix. যেমনঃ band এর proper prefix হচ্ছে b,ba,ban.

Proper suffix : এটা proper prefix এর ঠিক উল্টো! band এর proper suffix হচ্ছে d,nd,and.

দুইক্ষেত্রেই empty string both proper prefix and proper suffix, কারণ সবগুলো অক্ষর বাদ দিলে empty string ই থাকবে।

প্রথমেই আমাদের একটা prefix / π table তৈরি করতে হবে। এই টেবিলের সাইজ হবে প্যাটার্নের সাইজের সমান আর অ্যারের প্রত্যেকটা ঘরে থাকবে শুরু থেকে ওই ইনডেক্স পর্যন্ত সাবপ্যাটার্নের জন্যে যে longest proper prefix ও longest proper suffix ম্যাচ করে, তার দৈর্ঘ্য। এর জন্যে আমরা একটা ফাংশন বানাব, যার নাম দিতে পারি kmp_preprocess

একটা উদাহরণ দিলে বুঝতে সহজ হবে। ধরা যাক, আমাদের প্যাটার্ন হচ্ছে abab. এখন আমরা যদি ৩য় ইনডেক্স বিবেচনা করি (১-বেইজড), সেক্ষেত্রে আমাদের সাবপ্যাটার্ন হচ্ছে aba. এই সাবপ্যাটার্নের proper prefix হচ্ছে a,ab এবং proper suffix হচ্ছে a,ba. এখানে proper prefix “a” = proper suffix “a”, তাই ৩য় ইনডেক্সের মান হবে ১। এভাবে পুরো প্রিফিক্স টেবিল তৈরি করতে হবে।



i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
Π[i]	0						

Step : 1

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
Π[i]	0	0					

Step : 2

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
Π[i]	0	0	1				

Step : 3

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
Π[i]	0	0	1	2			

Step : 4

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
Π[i]	0	0	1	2	3		

Step : 5

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
Π[i]	0	0	1	2	3	0	

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
Π[i]	0	0	1	2	3	0	

i	1	2	3	4	5	6
P[i]	a	b	a	b	a	c
Π[i]	0	0	1	2	3	0

Step : 6

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
Π[i]	0	0	1	2	3	0	1

Step : 7

Reference : Tanvir’s Blog

উপরের ছবিটি দেখলেই ফাংশনটি কিভাবে কাজ করে বুঝতে পারার কথা। আমরা ২টা ইনডেক্স পয়েন্টার i,j রাখব। এখানে সবুজ কালারের ইনডেক্স পয়েন্টার হচ্ছে j আর নীল অথবা লাল কালারের ইনডেক্স পয়েন্টার হচ্ছে i. আসল কথা হচ্ছে, আমরা এই দুই ইনডেক্স পয়েন্টের ক্যারেক্টারদ্বয়কে কম্পেয়ার করব। **Step – 1 এ π[1] সবসময়ই ০, কারণ এক্ষেত্রে proper prefix ও proper suffix হচ্ছে empty string. আমরা এরপর ২য় ইন্ডেক্স থেকে π টেবিল তৈরির কাজ শুরু করব। Step – 2 তে ab এর কোন proper prefix, proper suffix ম্যাচ করে না, সো π[2]=0. Step – 3 তে aba এর proper prefix “a” = proper suffix**

“a”, সো $\pi[3]=1$. Step – 4 এ proper prefix “b” = proper suffix “b”. কিন্তু খেয়াল করে দেখো, এর থেকেও বড় proper prefix ও proper suffix ম্যাচ করে, “ab”. তাই $\pi[4]=2$ হবে।

Step – 6 এ $\pi[6]$ এর মান একটু অন্যরকমভাবে বের করা হয়েছে! যেহেতু, $j = 4$ ($j>1$), সেহেতু প্রথমে ৪নম্বর ইন্ডেক্সে থাকা ক্যারেক্টারের সাথে প্যাটার্নের ৬ নম্বর ইন্ডেক্সের ক্যারেক্টার কম্পেয়ার করা হয়েছে। ক্যারেক্টার দুটো মিললে আমরা আগের স্টেপগুলোর মতই কাজ করতাম। যেহেতু মেলেনি, সেক্ষেত্রে আমাদের $\pi[4-1]$ অর্থাৎ $\pi[3]$ এ যেই মানটা আছে সেই মানের এক ঘর পরের ক্যারেক্টারের সাথে আমরা আবার ৬ নম্বর ইনডেক্সের ক্যারেক্টারকে কম্পেয়ার করব! এখানে $\pi[3]=1$ অর্থাৎ আমরা এরপর ২ নম্বর ক্যারেক্টারের সাথে কম্পেয়ার করব। এভাবে যতক্ষণ পর্যন্ত ক্যারেক্টার ম্যাচ না করে, ততক্ষণ পর্যন্ত কম্পেয়ার করতে থাকব। ম্যাচ করলে আগের মতই π টেবিলে ($j+1$) রেখে দিব। সবশেষে ১ম ক্যারেক্টারের সাথেও ম্যাচ না করলে আমরা ওই ইনডেক্সে ০ রেখে দিব। খেয়াল করে দেখো, $\pi[6]=0$

স্টেপগুলো বুঝতে পারলে কোড লিখতে কোন সমস্যা হওয়ার কথা না!

প্রথমে একবার নিজে চেষ্টা করে দেখো।

নিচের কোডটা ০-বেইজড ইনডেক্সে লেখা হয়েছে।

```
void kmp_preprocess(string pattern)
{
    f[0]=0;
    int j = 0;
    for(int i = 1; i < m; i++)
    {
        if(pattern[i]==pattern[j])
            f[i]=j+1,j++;
        else
        {
```

```
            while(j!=0)
            {
                j = f[j-1];
                if(pattern[i]==pattern[j])
                {
                    f[i] = j+1;
                    j++;
                    break;
                }
            }
        }
    }
}
```

এবার এই π টেবিলের সাহায্যে আমরা কোন প্যাটার্ন কোন টেক্সটে আছে কি নেই, সেটা চেক করব। প্যাটার্ন চেক করার এই কাজটি উপরের ফাংশনটার মতই কাজ করে, টেক্সটে একটা ইনডেক্স পয়েন্টার i এবং প্যাটার্নে একটা ইনডেক্স পয়েন্টার j এর মাধ্যমে টেক্সটের সাথে প্যাটার্নের ক্যারেক্টার কম্পেয়ার করব। পুরো প্যাটার্ন ম্যাচ করলে ফাংশনটি true রিটার্ন করবে, নাহলে false.

নিচের ছবিটা ভালমত বুঝলে আশা করি, কোড লিখে ফেলতে পারবে।

Reference : Tanvir’s Blog

এখানে টেক্সটের ক্যারেক্টারের সাথে প্যাটার্নের ক্যারেক্টার কম্পেয়ার করতে করতে এগোচ্ছি। যখনই দুইটি ক্যারেক্টার ম্যাচ না করছে, তখন আমরা π টেবিলের ওই ইনডেক্সে যে মানটা আছে, সেটা ব্যবহার করে ক্যারেক্টার স্কিপ করছি। কেন স্কিপ করছি সেটার কারণ কিন্তু উপরেই বলেছিলাম, যেহেতু ওই

সাবপ্যাটার্নের proper prefix এবং proper suffix মিলে যাচ্ছে, সেহেতু আমরা ওই prefix এর জন্যে টেক্সটে আবার চেক করবনা। ওই অংশটুকু স্কিপ করে সামনে এগিয়ে যাব।

```
bool kmp(string text, string pattern)

{

    int j = 0;

    for(int i = 0; i < n; i++)

    {

        if(text[i]==pattern[j])

        {

            if(j==m-1)

            {

                return true;

            }

            j++;

        }

        else

        {

            while(j!=0)

            {

                j = f[j-1];

                if(text[i]==pattern[j])

                {

                    j++;
```

```
                    break;
                }
            }
        }
    }
    return false;
}
```

এই এলগো দিয়ে আরও অনেক কাজ করা যায়, যেমন কোন প্যাটার্ন টেক্সটের ভেতরে কতবার আছে সেটা বের করা, যেই প্রব্লেমটার কথা আমরা প্রথমেই বলেছিলাম।

KMP অ্যালগোরিদমের টাইম কমপ্লেক্সিটি $O(n+m)$ যা $O(n*m)$ এর থেকে অনেক কম! এক্ষেত্রে মেমোরি কমপ্লেক্সিটি $O(m)$

KMP অ্যালগোরিদমটা প্রথম প্রথম বুঝতে একটু সময় লাগতে পারে, তাই হতাশ না হয়ে কয়েকবার পড়ো, আর কোড কপি পেস্ট না করে স্টেপগুলো বুঝে বুঝে নিজে কোড লেখার চেষ্টা করো। মনে রাখতে হবে,

‘একবার না পারিলে, দেখো শতবার’

আর হ্যা, পেটভরে বুফে খাওয়াটা কিন্তু আসলেই মজার ছিল আর তার থেকেও বেশি মজার ছিল

আমার কিপটে বন্ধুর করুণ মুখটা!

এরকমভাবে বন্ধুদের জব্দ করতে নিচের প্রব্লেমগুলো সলভ করে অ্যালগোরিদমটা ঝালাই করে নাও!

10679 – I Love Strings!! (UVA)