রিকার্শন (Recursion)

Dec 25, 2015

প্রোগ্রামিং এর ক্ষেত্রে খুবই শক্তিশালী একটা টুল হচ্ছে রিকার্শন। রিকার্শনের মাধ্যমে আমরা অনেক বড় কোন ইটারেটিভ সমাধানকে খুব ছোট আকারে লেখতে পারি। আবার কিছু প্রবলেম আছে যেগুলো ইটারেটিভভাবে চিন্তা করা খুব কষ্টকর। সেগুলো রিকার্সিভ ভাবে চিন্তা করলে ইনটুইটিভ এবং সহজ সমাধান বের করা যায়। বিশেষ করে ডাইনামিক প্রোগ্রামিং এবং ডিভাইড অ্যান্ড কনকুয়ের প্রবমলেমগুলো সমাধান করার জন্য রিকার্শন একদম আদর্শ পন্থা। নবীনদের প্রায় সবারই প্রথম দিকে রিকার্শন বুঝতে খুব অসুবিধা হয়। এর মূল কারণ জিনিসটা ভিজুয়ালাইজ করা কঠিন। তবে ভিজুয়ালাইজ করতে পারলে এবং কিভাবে কাজ করে সে ব্যাপারে স্পষ্ট ধারনা হয়ে গেলে রিকার্শন অত্যন্ত মজার একটা বিষয়!

রিকার্শন কি?

মূলত কোন একটা ফাংশনকে রিকার্সিভ বলা হয় যখন সেই ফাংশনটা নিজেই নিজেকে কল করে! এখানেই যত কনফিউশন! একটু চিন্তা করার চেষ্টা করতে হবে এখানে। একটা ফাংশন নিজেই নিজেকে কল করছে। ব্যাপারটা কেমন হতে পারে। কল করলে কি হবে। মাথা খাটাতে হবে একটু। মূলত এই ব্যাপারটাই হচ্ছে রিকার্শন। বিভিন্ন ভাবে বিভিন্ন ধরনের রিকার্শন লেখার মাধ্যমে আমরা অনেক জটিল প্রবলেম সলভ করে ফেলতে পারি।

একটা ছোট রিকার্সিভ ফাংশন লেখা যাক।

```
void hello()
{
    printf("Hello World!\n");
    hello();
}
```

খুব সহজ সরল দেখতে একটা ফাংশন লেখলাম। যখন main ফাংশন থেকে hello ফাংশনটা কল হবে তখন কি হবে? hello এর ভেতরে আসলাম। Hello World! প্রিন্ট করলাম। তারপর আবার hello কল হবে। তাহলে আগের hello ফাংশন থেকে বের হওয়ার আগেই আমরা নতুন আরেকটা hello ফাংশনে প্রবেশ করলাম। আবার প্রিন্ট করলাম। আবার বের হওয়ার আগেই আরেকটা কল হল। এটা চলতেই থাকবে। অনেকটা ইনফিনিট লুপের মত! কখনই শেষ হবে না! আসলে ভুল

বললাম। শেষ হবে। কারণ ফাংশন কল হলে স্ট্যাকে মেমরি ব্যবহার করতে হয়। বার বার কল হতে হতে স্ট্যাক ওভারফ্লো হয়ে প্রোগ্রাম ক্র্যাশ করবে। অতএব বোঝা যাচ্ছে, রিকার্সিভ ফাংশন লেখলে এমনভাবে লেখতে হবে যেন সেটা এরকম ইনফিনিট লুপে পড়ে না যায়। সেটা যেন কোন এক সময় শেষ হয়, অর্থাৎ রিটার্ন করে।

বেস কেস (Base Case)

রিকার্সিভ ফাংশন যেন ইনফিনিট লুপে না পড়ে যায় সেজন্য ফাংশনটিকে কোন একটা বেস কেসের দিকে ধাবিত হতে হয়। অর্থাৎ ফাংশনটি প্রতিটা রিকার্সিভ কলে তার বেস কেসের দিকে এগিয়ে যাবে। বেস কেসে পৌঁছে গেলে আমরা রিটার্ন করব ফাংশন থেকে। এতে করে ফাংশনটা ইনফিনিট লুপে পড়বে না। এখন দেখা যাক সেটা আমরা কিভাবে করতে পারি। ধরা যাক আমরা 1 থেকে n পর্যন্ত নাম্বারগুলো প্রিন্ট করব। কিন্তু লুপ চালিয়ে নয়, রিকার্শনের মাধ্যমে। তার জন্য একটা ফাংশন লেখা যাক।

```
void print(int i)
{
    if(i > n) return;
    printf("%d ", i);
    print(i+1);
}

int main()
{
    print(1);
}
```

এখানে আমরা একটা ফাংশন লেখলাম print যার কাজ তাকে যে আর্গুমেন্ট পাঠানো হচ্ছে সেটা প্রিন্ট করা। ফাংশনটিকে আমরা 1 দিয়ে কল করব। প্রথমে ফাংশনটি দেখবে যে নাম্বারটি কি n এর থেকে বড় কিনা। কারণ আমরা n পর্যন্ত প্রিন্ট করতে চাই। যদি n এর থেকে বড় হয় তাহলে ফাংশন রিটার্ন করবে। অর্থাৎ আর কোন রিকার্সিভ কল হবে না। যদি n এর সমান বা ছোট হয়, তাহলে নাম্বারটি প্রিন্ট করে দিব। তারপর তার সাথে 1 যোগ করে আবার কল দিব। এই ব্যাপারটা লক্ষণীয়। আমরা 1 যোগ করার মাধ্যমে রিকার্শনটাকে তার বেস কেসের দিকে নিয়ে যাচ্ছি। ফলে একটা সময় n এর থেকে বড় নাম্বার দিয়ে কল হবে এবং ফাংশন রিটার্ন করা শুরু করবে। এই যে i > n কিনা চেক করছি সেটাই হচ্ছে এই ফাংশনের বেস কেস।

এক্সিকিউশন সিকোয়েন্স

উপরের প্রোগ্রামটা 1 থেকে n পর্যন্ত প্রিন্ট করছে। এখন n এর মান যদি 5 হয় তাহলে কিভাবে এক্সিকিউশন হবে সেটা দেখা যাক। নিচে সিকোয়েন্সটা দিচ্ছি।

```
print(1)
if(1 > 5) // false
output 1
    print(2)
    if(2 > 5) // false
    output 2
        print(3)
        if(3 > 5) // false
        output 3
            print(4)
            if(4 > 5) // false
            output 4
                print(5)
                if(5 > 5) // false
                output 5
                    print(6)
                    if(6 > 5) // true
                    return
                return
            return
        return
    return
return
```

ফাংশন কিভাবে কল হয় সেটা বুঝলে এটা বুঝতে খুব একটা অসুবিধা হওয়ার কথা না। ফাংশন কল হওয়ার সময় ফাংশনটা স্ট্যাকে পুশ হয়। তার কাজ শেষ হলে স্ট্যাক থেকে পপ হয়ে আগের ফাংশনের কাজে ফেরত চলে যায়। এখানে তাই দেখানো হচ্ছে। একেকবার যখন ফাংশন কল হচ্ছে তখন সেটা স্ট্যাকে পুশ হচ্ছে। তারপর যখন বেস কেসে যেয়ে রিটার্ন কল হল তখন রিটার্ন করা শুরু হল। যেহেতু print কলের পর আর কোন স্টেটমেন্ট নেই সেহেতু প্রতিটা ফাংশন সেখানেই রিটার্ন করছে। এখন এটাকে আরেকটু ইন্টারেস্টিং করা যাক। ধরা যাক বলা হল যে 1 থেকে n প্রিন্ট না করে উলটা করতে হবে। n থেকে 1 প্রিন্ট করব। একটু চিন্তা করতে হবে। চিন্তা করে বের করার চেষ্টা করতে হবে কিভাবে সেটা করা যায়। রিকার্শনের ধারনাটাই এরকম। পুরো ব্যাপারটাই ভিজুয়ালাইজেশন করতে পারার উপর নির্ভর করে।

উলটা করে প্রিন্ট করার জন্য আমরা নিচের মত করে print ফাংশনটা লেখতে পারি।

```
void print(int i)
{
    if(i > n) return;
    print(i+1);
    printf("%d ", i);
}
```

রিকার্সিভ কল করার পরে আমরা প্রিন্ট করলাম। কি হল তাতে? এখানেই রিকার্শনের মজা! রিকার্সিভ কলের আগে বা পরে লেখলাম কিনা তার উপর সমগ্র আউটপুট উলটে যেতে পারে! এটার এক্সিকিউশন সিকোয়েন্স দেখলেই পরিষ্কার হয়ে যাবে কিভাবে কি হচ্ছে।

```
print(1)
if(1 > 5) // false
    print(2)
    if(2 > 5) // false
        print(3)
        if(3 > 5) // false
            print(4)
            if(4 > 5) // false
                print(5)
                if(5 > 5) // false
                    print(6)
                    if(6 > 5) // true
                    return
                output 5
                return
            output 4
            return
        output 3
        return
    output 2
    return
output 1
return
```

রিকার্সিভ কল করে করে আমরা বেস কেসের দিকে আগাচ্ছি, কিন্তু প্রিন্ট করছি না। প্রিন্ট করা শুরু করলাম যখন ফাংশনগুলো রিটার্ন করবে তার আগে। ফলে আমরা একদম ভেতরের লেভেলে সবার আগে প্রিন্ট করলাম। সেখানে 5 প্রিন্ট হল। তারপর রিটার্ন করে এসে 4 প্রিন্ট হল। এভাবে 1 পর্যন্ত যাবে। এই ব্যাপারটা ভিজুয়ালাইজ করতে পারলেই রিকার্শন অনেকখানি বোঝা হয়ে যাবে। F কোন ফাংশনের ভেতর থেকে যদি G কোন ফাংশন কল হয় তাহলে যেখানে কল হচ্ছে, G থেকে রিটার্ন করে আসার পর ঠিক তার পর থেকে F এর এক্সিকিউশন আবার চলতে শুরু করবে। এটা মাথায় রাখতে হবে যে ব্যাপারটা রিকার্শনের ক্ষেত্রেও সত্যি।

সাধারন ফরম্যাট

প্রত্যেকটা রিকার্সিভ ফাংশনেরই একটা সাধারন ফরম্যাট থাকে। রিকার্সিভ ফাংশনকে দুটো অংশে ভাগ করে ফেলা যায়। ১) বেস কেস ২) রিকার্সিভ কল

```
return_type function_name()
{
    // First part: Base Case
    // ===============================
    // Check base condition and return
    // if condition is met.
    // Second part: Recursive Call
    // ===============================
    // One or more recursive calls to
    // the function itself and then
    // return.
}
```

যেকোন রিকার্সিভ ফাংশনকে এই ফরম্যাটে নিয়ে কোড করলে চিন্তা করতে সুবিধা হয়। তবে এর ব্যতিক্রম হতেই পারে। শুরুতে আমরা বেস কেস লেখবো। যেখানে কোন রিকার্সিভ কল থাকবে না। এরপর আমরা রিকার্সিভ কলের অংশটা লেখবো যেটা আস্তে আস্তে বেস কেসের দিকে এগিয়ে যাবে।

সাইকেল

একটা ব্যাপার সবসময় খেয়াল রাখতে হবে, এমনভাবে যেন আমরা রিকার্সিভ কল না লেখি যাতে করে কোন সাইকেল তৈরি হয়। সাইকেল হলে যে সমস্যা হবে তা হল ঐ সাইকেলের মধ্যে বারবার রিকার্সিভ কল হতেই থাকবে। কখনই সেখান থেকে রিটার্ন করবে না। নিচে একটা সুডোকোড দিয়ে উদাহরণ দেই।

```
void foo(int i)
{
    // Base case
    if(i == 0) return;
    if(i == n) return;
    // Recursive call
    foo(i+1);
    foo(i-1);
}
```

এখানে আমরা রিকার্সিভ কল করে i+1 এবং i-1 দুইদিকেই যাচ্ছি। দেখে মনে হতে পারে বেস কেসের দিকেই যাচ্ছে। যেহেতু বেস কেস একটা 0 আরেকটা n। কিন্তু সমস্যা হবে বেস কেসে যাওয়ার আগেই। মাঝে ধরা যাক 4 দিয়ে কল হল। এখন 4-1 বা 3 দিয়ে কল হবে। যখন 3 তে যাবে, তখন 3+1 বা 4 দিয়ে কল হবে। আবার 4 থেকে 3 তে যাবে, সেখান থেকে আবার 4 এ আসবে। এই সাইকেল শেষ হবে না। ফলে যদি এমন কোন রিকার্শন লেখি যেখানে সাইকেল তৈরি হয়, তাহলে প্রোগ্রাম ইনফিনিট লুপে পড়ে যাবে।

একাধিক রিকার্সিভ কল

সাইকেল বোঝানোর জন্য আমরা যে কোডটা লেখলাম সেখানে দুইটা রিকার্সিভ কল আছে। বেশিরভাগ ক্ষেত্রেই যখন আমরা রিকার্সিভ সমাধান লেখবো দেখা যাবে একাধিক রিকার্সিভ কল আছে। এখন আমাদের বুঝতে হবে একাধিক রিকার্সিভ কল হলে কি হয়।

আবারও পুরো ব্যাপারটা ভিজুয়ালাইজেশনের উপর নির্ভর করবে। একাধিক রিকার্সিভ কল হলে একটা ট্রি স্ট্রাকচার তৈরি হয়। ব্যাপারটা একটু এঁকে দেখানোর চেষ্টা করি। নিচের ফাংশনটি খেয়াল করি।

```
void foo(int n)
{
    // Base case
    if(n < 1) return;
    // Recursive case
    foo(n-1);
    foo(n-2);
}
```

এখানে আমরা দুটো রিকার্সিভ কল করছি। একবার n-1 দিয়ে, আরেকবার n-2 দিয়ে। এখানে ফাংশন কলটা একটা ট্রি এর মত করে আগাবে। যদি আমরা 5 দিয়ে ফাংশনটা কল করি তাহলে এরকম হবে।

```
                         foo(5)
                  /                 \
                 /                   \
           foo(4)                     foo(3)
         /       \                  /        \
    foo(3)       foo(2)        foo(2)         foo(1)
    /    \       /    \        /    \         /    \
foo(2) foo(1) foo(1) foo(0) foo(1) foo(0) foo(0) foo(-1)
```

অর্থাৎ প্রতিটা ফাংশন কল থেকে আরও দুইটা করে ফাংশন কল হচ্ছে। ফলে যত লেভেল বাড়ছে আমরা আগের লেভেলের ডাবল ফাংশন কল পাচ্ছি। 1টা, 2টা, 4টা, 8টা এভাবে ফাংশন কল হচ্ছে। যেটা বোঝাতে চাচ্ছি তা হল যে রিকার্সিভ কলে একাধিক কল থাকলে ট্রি আকারে ফাংশন কল হয়। ব্যাপারটা ভিজুয়ালাইজ করতে পারতে হবে। যদি দুটোর জায়গায় তিনটা ফাংশন কল হত তাহলে কিরকম ট্রি হতো সেটা নিজেরা এঁকে দেখতে হবে। আরেকটি বিষয় হচ্ছে যে যেহেতু ট্রি আকারে বাড়ে, সেহেতু এটার গ্রোথ এক্সপোনেনশিয়াল। যেমন উপরের উদাহরণে যদি h তম লেভেল যাই আমরা তাহলে সেই লেভেল মোট ফাংশন কল হবে 2^h টা। অর্থাৎ অনেক বেশি মেমরি এবং টাইম কমপ্লেক্সিটি লাগবে! তবে এখানে বলে রাখি, বেশিরভাগ ক্ষেত্রেই এ ধরনের রিকার্শন অপটিমাইজ করে কাজে লাগাতে হয় যার অন্যতম উদাহরণ হচ্ছে ডাইনামিক প্রোগ্রামিং। সেদিকে এখন আর না যাই।

সাধারন ব্যবহার

রিকার্শন সাধারনত কখন ব্যবহার করা হয়? মূলত যেসব প্রবলেমকে ছোট সাবপ্রবলেম ভাঙ্গা যায় এবং সাবপ্রবলেমের সমাধান দিয়ে মূল প্রবলেমটা সলভ করা যায় সেসব ক্ষেত্রে রিকার্শন ব্যবহার করা হয়। যেমন প্রব্লেমটা যদি এমন হয় যে আমরা n এর জন্য সমাধান বের করতে পারি যদি n-1 এর জন্য সমাধান জানি, তাহলে আমরা রিকার্শন ব্যবহার করব। যদি এমন হয় যে আমরা n এর জন্য সমাধান বের করতে পারি যদি n-1 এবং n-2 এর জন্য সমাধান জানি, তাহলে আমরা রিকার্শন ব্যবহার করব। এতে করে কাজটা অনেক সহজ হয়ে যায়। উদাহরণ দিয়ে দেখাই।

ফ্যাক্টরিয়াল

n ফ্যাক্টরিয়াল বা n! বলতে আমরা বুঝি 1 থেকে n পর্যন্ত নাম্বারগুলোর গুণফল।

$n! = n * (n-1) * (n-2) * \ldots * 1$

একটু ভালোমত যদি আমরা লক্ষ্য করি তাহলে এখানে দুটো অংশ খেয়াল করতে পারি।

n! = n * (n-1) * (n-2) * … * 1

দুটো অংশ আলাদা করে দিলাম। প্রথম অংশ n, পরের অংশ (n-1)*(n-2) … 1। এই পরের অংশটি কিন্তু জটিল কিছু না। এটা হচ্ছে (n-1)! । অর্থাৎ n! = n*(n-1)! । অতএব আমরা যদি n-1 এর ফ্যাক্টরিয়াল বের করতে পারি, তার সাথে n গুণ করে দিলেই n! পেয়ে যাচ্ছি। এই হচ্ছে সাবপ্রবলেম যেটা দিয়ে আমরা মূল প্রবলেমের সমাধান করতে পারি। অতএব আমরা রিকার্সিভ কল দিয়ে n-1 এর ফ্যাক্টরিয়াল বের করব এবং তার সাথে n গুণ করে দিয়ে রিটার্ন করব।

তাহলে বেস কেস কি হবে? বেস কেস হবে 1 এর জন্য। 1 এর জন্য সমাধান আমরা জানি। 1! = 1। সাধারনত বেস কেস এমন কিছুই হয় যেটা আমাদের জানা আছে, সরাসরি কোডে লিখে দিতে পারি কোন হিসাব ছাড়াই। তাহলে ফ্যাক্টরিয়ালের কোড কিরকম হবে?

```
int fact(int n)
{
    // Base case
    if(n == 1) return 1;
    // Recursive part
    return n*fact(n-1);
}
```

আরেকটা উদাহরণ দেখা যাক।

ফিবোনাচ্চি নাম্বার

আমরা জানি ফিবোনাচ্চি সিকোয়েন্সের প্রথম দুটো নাম্বার হচ্ছে 1 এবং 1। এর পরের নাম্বারগুলো হচ্ছে সিকোয়েন্সের আগের দুটো নাম্বারের যোগফল। তাহলে যদি আমাদেরকে n তম নাম্বার বের করতে বলে কি করতে পারি? দেখা যাচ্ছে যে n তম নাম্বার হবে n-1 এবং n-2 তম নাম্বার দুটোর যোগফল। অতএব দুটো সাবপ্রবলেম সলভ করতে পারলে তাদেরকে যোগ করে দিলেই আমাদের n এর জন্য সমাধান হয়ে যাচ্ছে। আর বেস কেস হচ্ছে প্রথম এবং দ্বিতীয় নাম্বার দুটি যে দুটি আমরা আগে থেকেই জানি। তাহলে কোড হবে এরকম।

```
int fib(int n)
{
    // Base case
    if(n == 1) return 1;
    if(n == 2) return 1;
```

```
    // Recursive case
    return fib(n-1) + fib(n-2);
}
```

এগুলো রিকার্শনের একদম সহজ সরল কিছু প্রায়োগিক দিক। ডাইনামিক প্রোগ্রামিং, গ্রাফ থিওরি, ডিভাইড অ্যান্ড কনকুয়ের ইত্যাদি বিভিন্ন ধরনের প্রবলেম সলভ করতে আরেকটু জটিল রিকার্শন ব্যবহার করতে হয়। তবে সবক্ষেত্রেই বেসিক কনসেপ্ট একই। বেস কেস থাকবে, রিকার্সিভ কেস থাকবে। ভিজুয়ালাইজ করতে পারলেই পুরো ব্যাপারটা অনেক সহজ হয়ে যায়।

প্যালিন্ড্রোম চেক

রিকার্শনের ব্যাপারে কথা বলতে গেলে প্যালিন্ড্রোম চেকের এই উদাহরণটা না দিলে আমার কেন যেন ভালো লাগে না। প্রথম যখন এই কোডটা দেখি তখন মন ভরে গিয়েছিল। রিকার্শনের উপর সম্মান বেড়ে গিয়েছিল অনেক গুণ। রিকার্শন যে কতটা এলিগেন্ট তা বুঝতে পেরেছিলাম তখনই। তাই সবাইকে কোডটা দেখাতে ইচ্ছা করে। :P

আমরা সাধারনত লুপ চালিয়ে প্যালিন্ড্রোম চেক করি। সহজেই করা যায়। কয়েক লাইনের কোড। স্ট্রিং দুইপাশ থেকে চেক করে আসলেই হয়ে যায়। রিকার্শনের মাধ্যমেও একই কাজ করা যায়।

```
int isPali(char *s, int l, int r)
{
    return ((l >= r) || (s[l] == s[r] && isPali(s, l+1, r-1)));
}
int main()
{
    char str[100];
    scanf("%s", str);
    if(isPali(str, 0, strlen(str)-1))
        printf("Palindrome\n");
    else
        printf("Not palindrome\n");
}
```

মাত্র এক লাইনের একটা রিকার্সিভ ফাংশন। বোঝার দায়িত্ব পাঠকের হাতে তুলে দিলাম। :P

শেষ কিছু কথা

রিকার্শন খুবই ইন্টারেস্টিং একটা বিষয়। এটা বোঝা যেমন সহজ, আবার তেমনই কঠিন। ছোট রিকার্শন দেখলেই বোঝা যায়। বড় জটিল রিকার্শনগুলো অনেক চিন্তা করলেও মাথা তালগোল পাকিয়ে যায়। পুরোটাই নির্ভর করে অভিজ্ঞতা আর প্র্যাক্টিসের উপর। অনেক প্র্যাক্টিস প্রয়োজন। বিভিন্ন ধরনের রিকার্শন ভিজুয়ালাইজ করতে পারতে হবে। এই ব্যাপারটার উপর বার বার জোর দিচ্ছি, "**ভিজুয়ালাইজেশন**"। মাথার ভেতর পুরো জিনিসটার ছবি এঁকে ফেলতে পারতে হবে। খুব জরুরি এটা।

রিকার্শনের মূল সমস্যা হয় একটা প্রবলেমকে রিকার্সিভভাবে চিন্তা করতে পারাতে। একবার রিকার্সিভ মডেলে ফেলে দিতে পারলে কোড করা খুব সহজ। কয়েক লাইন কোড দিয়ে বিশাল সমস্যা সমাধান করা যায়। আরেকটা সমস্যা হচ্ছে ডিবাগিং। রিকার্সিভ ফাংশন ডিবাগ করা খুব কষ্টকর। বিশাল বিশাল ট্রি স্ট্রাকচার তৈরি হয়। এগুলোর কোথায় যেয়ে কি সমস্যা হচ্ছে, রিকার্সিভ কল এর আগে পরে কখন কি লেখছি সেগুলোর কোনটার কারণে সমস্যা হচ্ছে এগুলো ধরতে পারাটা অনেক দক্ষতার ব্যাপার। যেহেতু এগুলোর কোন ধরা বাঁধা নিয়ম নেই তাই অভিজ্ঞতা এবং প্র্যাকটিসই একমাত্র উপায় বিষয়টাকে আয়ত্তে আনার।

আরেকটা বিষয় বলে রাখি, রিকার্সিভ সমাধান লেখার সময় চেষ্টা করতে হবে ফাংশনের ভেতর থেকে কোন গ্লোবাল ভ্যারিয়েবল বা অ্যারে চেঞ্জ না করার। এতে বিভিন্ন সমস্যা হতে পারে যেটা প্রাথমিক দৃষ্টিতে ধরা পড়বে না। মনে হবে কাজ করার কথা, কিন্তু করবে না। রিকার্সিভ সমাধান লেখলে যেসব ভ্যারিয়েবল পরিবর্তন হবে সেগুলো ফাংশনের ভেতরে রাখতে হবে। গ্লোবাল ভ্যারিয়েবল গুলো শুধুমাত্র পড়ার (read) জন্য। সেখানে লেখা (write) যাবে না। এটা মাথায় রেখে কোড করলে অনেক সমস্যা শুরুতেই এড়ানো যায়।

রিসোর্স

রিকার্শন ভালো মত আয়ত্তে আনতে অনেক প্র্যাকটিস করার কোন বিকল্প নেই। নবীনদের উচিত আরো ভালো মত বিষয়টা নিয়ে জানা। এখানে কিছু ভালো আর্টিকেলের লিংক দিচ্ছি।

- visualgo এর সিমুলেশন
- ফাহিম ভাই এর লেখা
- জোবায়ের ভাই এর লেখা
- তানভীর ভাই এর লেখা
- টপকোডার টিউটোরিয়াল
- টাওয়ার অফ হ্যানয় প্রবলেম

প্র্যাকটিস প্রবলেম

জোবায়ের ভাই এর এই আর্টিকেলে অনেকগুলো প্র্যাকটিস প্রবলেম দেওয়া আছে। এগুলো সমাধানের চেষ্টা করতে হবে।