Topic: BIT MANIPULATION

Md. Nazmus Sakib

Programmer, Writer

01992547202

engrsakib02@gmail.com

linkedIn/
codeforces/leetcode/codechef{engrsakib}

বিট ম্যানুপুলেশন বুঝার আগে আমাদের সংখ্যা পদ্ধতি সম্পর্কে জানা উচিত। কেননা সংখ্যা পদ্ধতি না বুঝলে আমরা সঠিক ভাবে বিট এর ব্যবহার করতে পারব না। আমারা দৈনন্দিন কাজে যে সংখ্যা পদ্ধতি ব্যবহার করি সেটা দশ ভিত্তিক সংখ্যা পদ্ধতি। যা ডেসিমাল নামে পরিচিত। যেখানে আমরা ০ থেকে ৯ সংখ্যা আমরা ব্যবহার করে থাকি। কিন্তু কম্পিউটার ০ ও ১ ছাড়া কিছু বুঝে না। অর্থ্যাৎ বিদ্যুতের উপস্থিতি ও অনুপস্থিতি। তাই এই কারণে একটা ডিজিটাল ডিভাইসকে সুন্দর ভাবে পরিচালনার জন্য আমাদের বিট ম্যানুপুলেশন সম্পর্কে ভালো জ্ঞান রাখা জরুরি। এতে করে আপনার সফটওয়্যার এর ম্যামরি ও রান টাইম অনেক ফাস্ট হবে। এখন আমরা বিট ম্যানুপুলেশন সম্পর্কে বিস্তারিত জ্ঞান অর্জন করব।

বিট ম্যানুপুলেশন জানার পূর্বে আমাদের নিম্নের কিছু বিষয় জানা প্রয়োজন, যা ধারাবাহিক ভাবে আলোচনা করা হলোঃ

১. প্রাইম জেনারেটর:

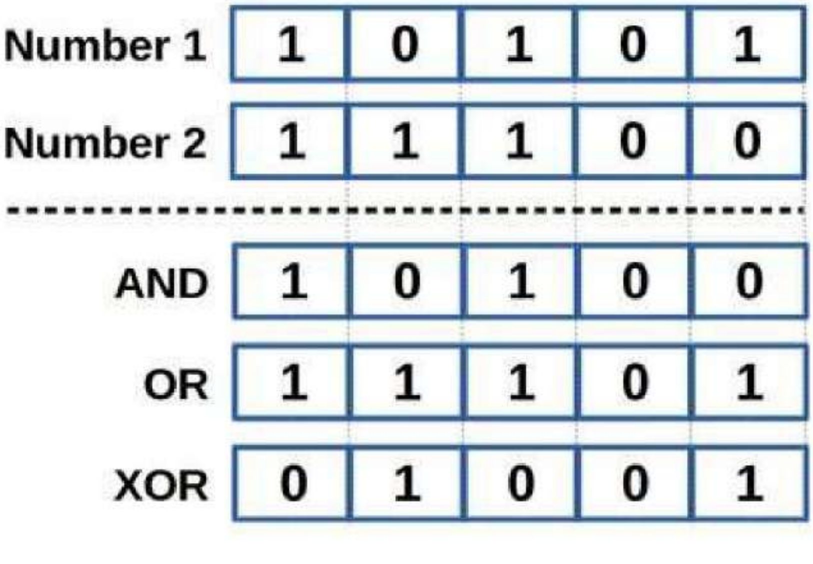
প্রাচীনকাল থেকেই গণিতবিদরা মাথা ঘামাচ্ছেন প্রাইম নাম্বার বা মৌলিক সংখ্যা নিয়ে। প্রাইম নাম্বারগুলো মধ্যে লুকিয়ে আছে বিস্ময়কর কিছু সৌন্দর্য। যেকোনো কম্পোজিট বা যৌগিক সংখ্যাকে একাধিক প্রাইমের গুণফল হিসাবে মাত্র একভাবে লেখা যায়,ঠিক যেমন সব যৌগিক পদার্থ একাধিক মৌলিক পদার্থের সমন্বয়ে তৈরি। প্রাচীনকাল থেকেই মানুষ প্রাইম নিয়ে গবেষণা করছে,চলছে এখনো। গাউস,ফার্মা,ইউলারের মত কিংবদন্তি গণিতবিদরা কাজ করেছেন প্রাইম নিয়ে। দ্রুত গতিতে প্রাইম সংখ্যা বের করার একটি পদ্ধতি আবিষ্কার করেন Eratosthenes,২০০ খ্রিস্টপূর্বের একজন গ্রীক গণিতবিদ,বিজ্ঞানি ও কবি। ২২০০ বছরেরও পুরানো সেই পদ্ধতি ব্যবহার করে আমরা আধুনিক কম্পিউটারে প্রাইম জেনারেট করি,খুব কম সময়ে বের করা যায় ১০কোটির নিচে সব প্রাইম সংখ্যা। এই অ্যালগোরিদমটি sieve of Eratosthenes নামে পরিচিত,প্রোগ্রামিং এর জগতে সুন্দরতম অ্যালগোরিদমগুলোর মধ্যে এটি একটি। sieve এর শাব্দিক অর্থ হলো ছাকনি যা অপ্রয়োজনীয় অংশ ছেটে ফেলে (A sieve, or sifter, separates wanted elements from unwanted material using a woven screen such as a mesh or net)। Eratosthenes এর ছাকনি যৌগিক সংখ্যাগুলোকে ছেটে ফেলে দেয়।

আমরা জানি প্রাইম সংখ্যা বলতে সেই সংখ্যাকে বুঝায় যাকে ১ এবং সেই সংখ্যা দ্বারা অন্য সংখ্যা দ্বারা ভাগ করা যায় না। যেমনঃ ১, ৩, ৫, ৭, ১৩, ১৭, ১৯ ইত্যাদি। যেকোনো সংখ্যাকে আমরা কয়েকটি প্রাইমের গুণফল হিসাবে লিখতে পারি যাদের প্রাইম ফ্যাক্টর বলা হয়:

$$N = p_1 * p_2 * p_3 \dots * p_i$$

n যদি নিজেই প্রাইম হয় তাহলে $n=p1(=n)=1(=n)$।

উপরের আলোচনা থেকে আমরা খুব সহজে প্রাইম সংখ্যা বের করা শিখলাম। বিটওয়াইজ অপারেশন এর জন্য আমাদের প্রাইম সংখ্যা খুব দরকার হবে। এখন আমরা বিটওয়াইজ অপারেশন এর কাজ শুরু করব।



Number 1	1	0	1	0	1
Number 2	1	1	1	0	0
AND	1	0	1	0	0
OR	1	1	1	0	1
XOR	0	1	0	0	1

Bitwise Operators

বিটওয়াইজ অপারেশনে প্রধানত ৬ টা অপারশন হয়। নিম্নে তা দেওয়া হলোঃ

- bitwise AND {&}

এখানে সবগুলো ইনপুট যদি ওপেন (১) হয় তবে ইনপুট ওপেন (১) হবে।

- bitwise OR {|}

এখানে সবগুলো ইনপুট এর যেকোন একটা ওপেন (১) হলে আউটপুট ওপেন (১) হবে।

- bitwise XOR {^}

এখানে সবগুলো ইনপুট একই হলে ক্লোসড হবে (০) বাকি সময় ওপেন (১) হবে।

- left shift {<<}

এখানে লেফট শিফট আসলে প্রতিটি বিট কে সংখ্যকবার বামে সরে যাবে।

- right shift {>>}

এখানে রাইট শিফট আসলে প্রতিটি বিট কে সংখ্যকবার ডানে সরে যাবে।

- bitwise NOT {~}

এটা আসলে ইনপুট সম্পূর্ন বিপরীত হয়ে যাবে। অর্থাৎ ওপেন (১) থাকলে ক্লোসড (০) হবে। আবার ক্লোসড (০) থাকলে ওপেন হবে (১)।

X	Y	X & Y	X | Y	~X	~Y	X ^ Y
0	0	0	0	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	1	0	0	0

বিটওয়াইজ অপারেটর সম্পর্কে আকর্ষণীয় তথ্য

১। লেফট শিফিট এবং রাইট শিফট অপারেটর নেতিবাচক সংখ্যার জন্য ব্যবহার করা উচিত নয়। এতে করে গার্বেজ ভ্যালু আসার সম্ভাবনা রয়েছে।

যেমন 1 << -1 or 1 >> -1. এসকল ক্ষেত্রে গার্বেজ ভ্যালু আসবে। এজন্য আমাদের সর্বদা ধনাত্বক সংখ্যা ব্যবহার করতে হবে। আমরা জানি ইন্টিজার সংখ্যা ৩২ বিট ব্যবহার করে ও লং লং ৬৪ বিট ব্যবহার করে থাকে।

২। দুটো সংখ্যার বিটওয়াইজ OR শুধুমাত্র তখন হবে যখন সেই দুটো সংখ্যার যোগফলে কোন ক্যারি থাকবে না।

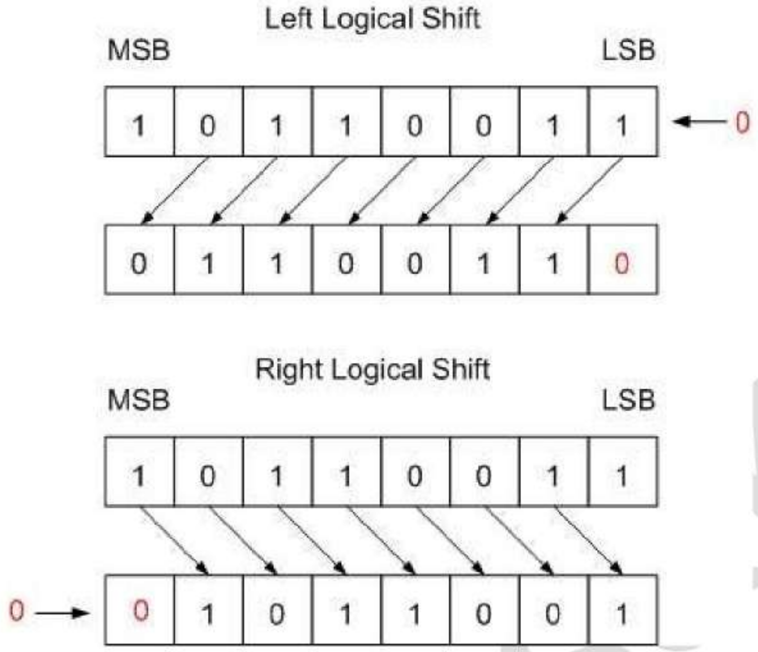
৩। লজিকাল অপারেটর এর স্থানে বিটওয়াইজ অপারেটর ব্যবহার করা উচিত নয়।

লজিকাল অপারেটর (&&, ||)

বিটওয়াইজ অপারেটর (&, |)

লজিকাল অপারেটর যেকোন অশূন্য উপাদানকে ১ হিসাবে বিবেচনা করে। কিন্তু বিটওয়াইজ অপারেটর একটা পূর্ণ সংখ্যা মান প্রদান করে থাকে।

৪। সংখ্যাকে লেফট শিফট করলে সংখ্যাটির দুই দ্বারা গুণের সমান হয়। X * 2 == left_Shift, অপরদিকে রাইট শিফট করলে সংখ্যাটির দুই দ্বারা বিভাজ্যের সমান হয়। X / 2 == right_Shift।



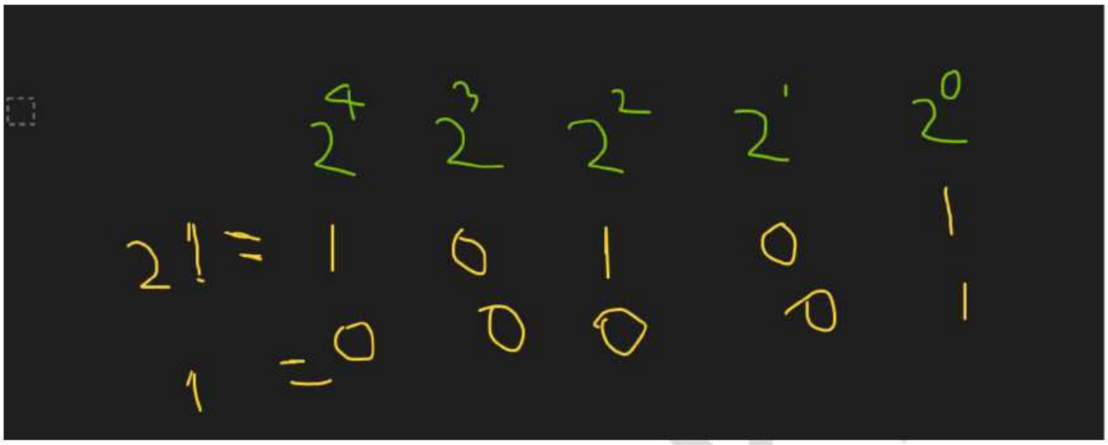
৫। & অপারেটর দ্বারা সংখ্যা জোড় না বিজোড় এটা চেক করা যায়। অর্থাৎ (X & 1) এর আউটপুট বিজোড় হলেই অশূন্য হবে অন্যথায় মান শূন্য হবে।

৬। ~ অপারেটর ব্যবহার করার সময় আমাদের বিশেষ সতর্কতা অবলম্বন করা উচিত।

একটি ছোট সংখ্যার উপর ~ অপারেটর প্রয়োগ করলে সংখ্যাটি ভিন্ন হয়ে যেতে পারি যদি আমরা সেটাকে একটা আনসাইন ভেরিয়েবলে সংরক্ষণ করি।

চেক বিট ওপেন ও ক্লোসড

আমরা এখন শিখব কোন সংখ্যার বাইনারি বিট ওপেন আছে না ক্লোসড আছে।



২১ এর বাইনারি বিট ১ ০ ১ ০ ১

০ এর বাইনারি বিট ০ ০ ০ ০ ১

এখন আমরা যদি ২১ এর K বিট ওপেন না ক্লোসড চেক করতে চাই তবে ২১ কে K পর্যন্ত লেফট শিফট করে ১ এর সাথে এন্ড করে দিব যদি আউটপুট ১ আসে তবে বিট ওপেন আসবে অন্যথায় ক্লোসড হবে।

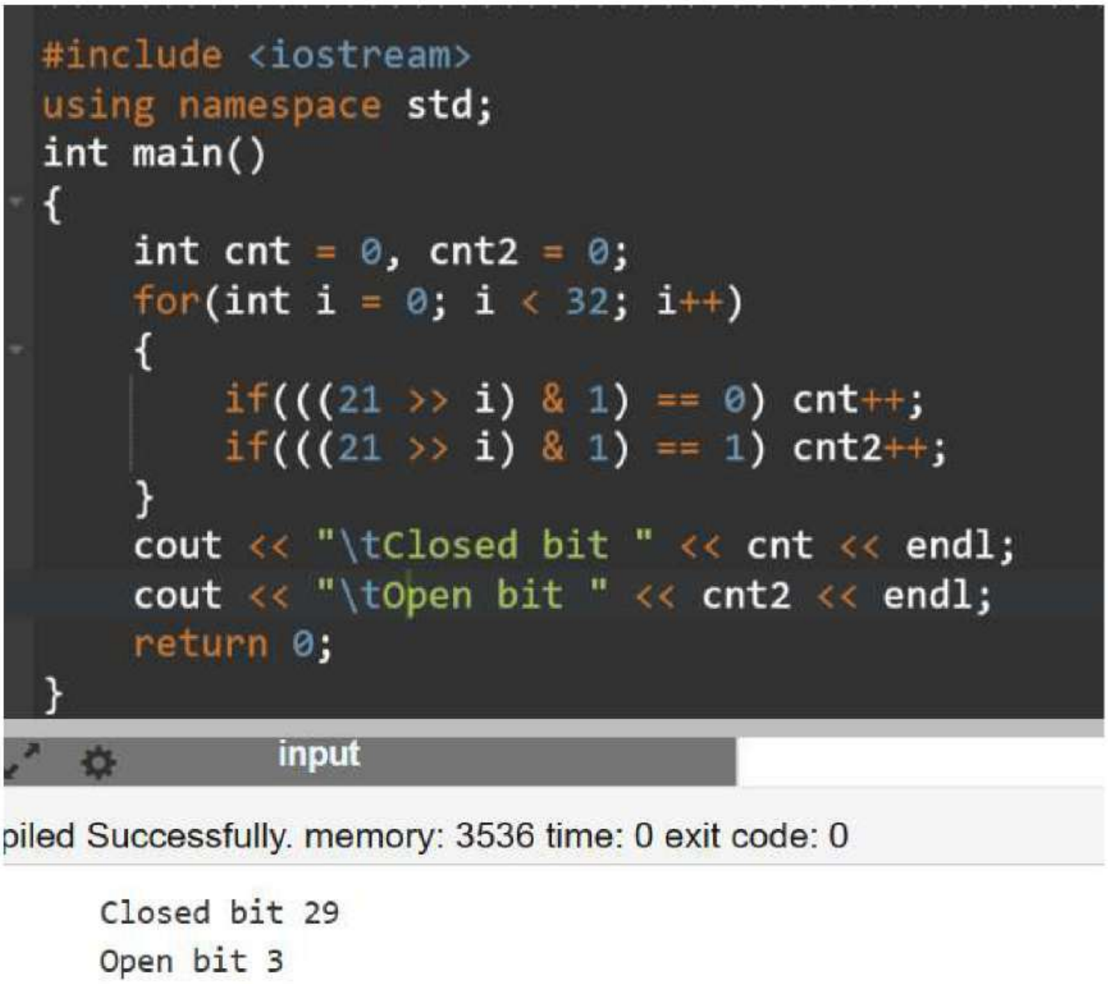
যেমনঃ

```cpp
#include <iostream>
using namespace std;
int main()
{
    cout << "\t K = 4 : " << ((21 >> 4) & 1) << endl;
    cout << "\t K = 3 : " << ((21 >> 3) & 1) << endl;
    return 0;
}
```

```
iled Successfully. memory: 3584 time: 0 exit code: 0

        K = 4 : 1
        K = 3 : 0
```

উপরে আমরা ২১ এর K = 4 ও K = 3 বিট ওপেন না ক্লোসড এটা চেক করে দেখব। এটার জন্য আমরা দেখতে পাচ্ছি ৪ তম বিটের জন্য আউটপুট ১ এসেছে। তার মানে এটা ওপেন আছে। কিন্তু ৩তম বিট এর জন্য আউটপুট 0 এসেছে তারমানে এটা ক্লোসড আছে।

আমরা যেহেতু বিট ওপেন না ক্লোসড আছে এটা বের করতে পেরেছি তাহলে আমরা খুব সহজেই কয়টা ওপেন বিট আছে আবার কয়টা ক্লোজড বিট আছে সেটা কাউন্ট করতে পারি। যেমনঃ



```
#include <iostream>
using namespace std;
int main()
{
    int cnt = 0, cnt2 = 0;
    for(int i = 0; i < 32; i++)
    {
        if(((21 >> i) & 1) == 0) cnt++;
        if(((21 >> i) & 1) == 1) cnt2++;
    }
    cout << "\tClosed bit " << cnt << endl;
    cout << "\tOpen bit " << cnt2 << endl;
    return 0;
}
```

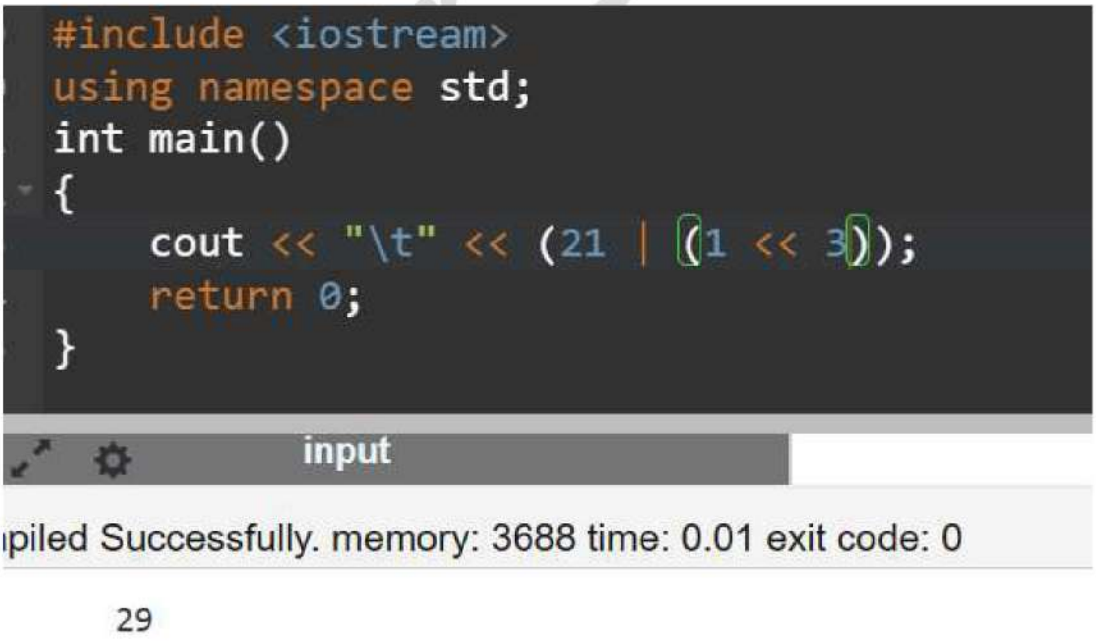
```
piled Successfully. memory: 3536 time: 0 exit code: 0

	Closed bit 29
	Open bit 3
```

আমরা জানি ইন্টিজারে ৩২ বিট এজন্য আমরা ২১ এর ৩২ বিট পর্যন্ত লুপ চালিয়ে দিয়ে কাউন্টার রেখে দেখতে পারতেছি কয়টা বিট ওপেন আছে আর কয়টা ক্লোসড আছে।

ক্লোসড বিটকে ওপেন করা

এখন আমরা দেখব কিভাবে একটা ক্লোসড বিটকে ওপেন করব। আমরা ২১ বিটস এর কোন একটা ক্লোসড বিটকে ওপেন করে দেখব

যে K তম বিটকে ওপেন করতে হবে আমরা ১ এর বিটকে K পর্যন্ত লেফট শিফট করব 1 << K । এরপর সেটা উক্ত সংখ্যার সাথে OR (|) পারফর্ম করব।

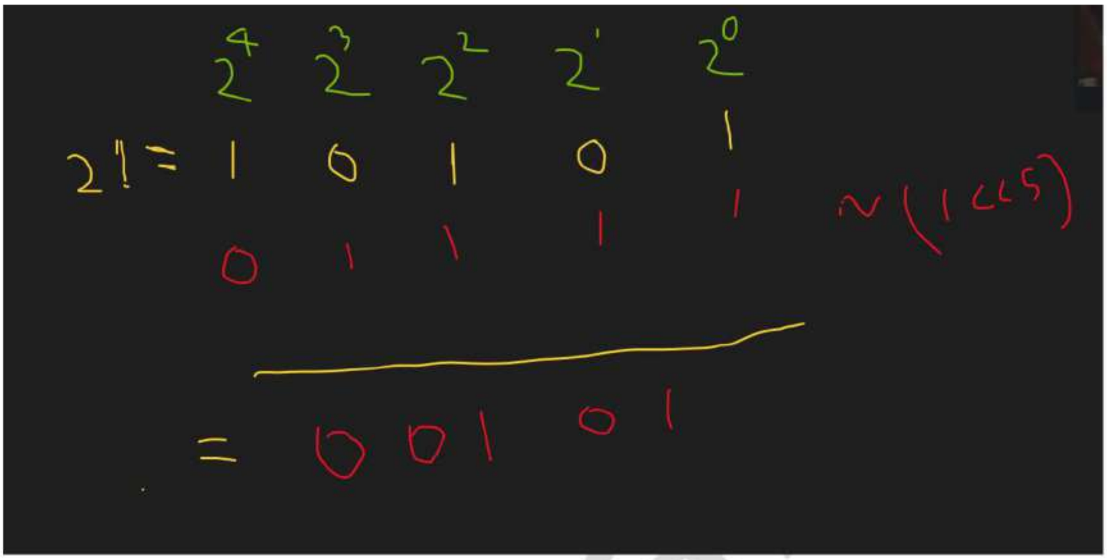


```cpp
#include <iostream>
using namespace std;
int main()
{
    cout << "\t" << (21 | (1 << 3));
    return 0;
}
```

```
piled Successfully. memory: 3688 time: 0.01 exit code: 0

    29
```

ওপেন বিটকে ক্লোসড করা

এখন দেখব যে কিভাবে একটা ওপেন বিটকে ক্লোসড করা যায়। আমরা ২১ এর ৪ তম বিটকে ক্লোসড করে দেখব।



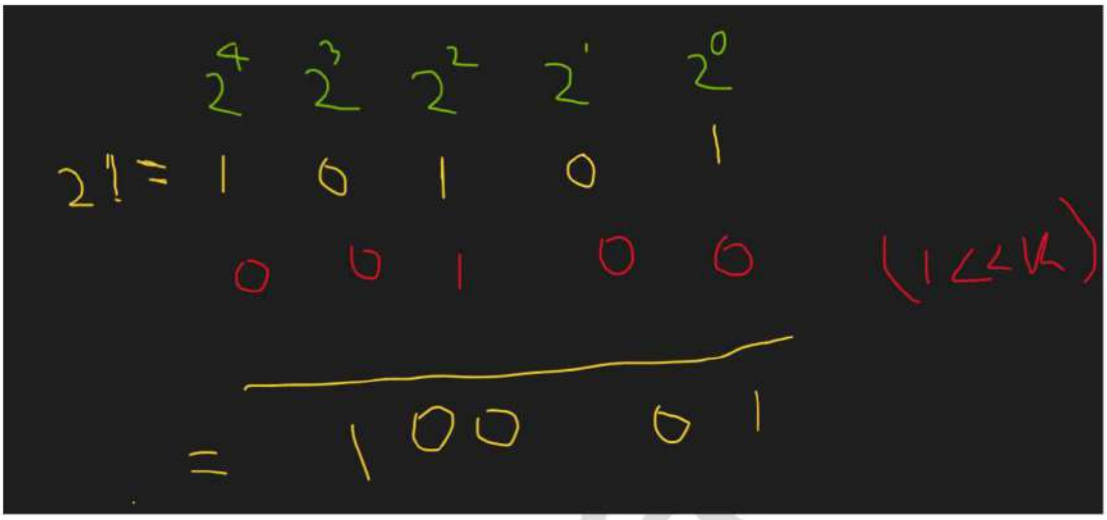
যে K তম বিটকে ক্লোসড করব সেই বিট পর্যন্ত ১ কে লেফট শিফট করে নট (~) অপারেশন ~(1 <<k) চালিয়ে দিবো। এরপর সেটার সাথে ২১ কে এন্ড (&) 1 অপারেশন চালিয়ে দিলে K তম বিট ক্লোসড হয়ে যাবে।

```
#include <iostream>
using namespace std;
int main()
{
    cout << "\t" << (21 & ~(1 << 4));
    return 0;
}
```

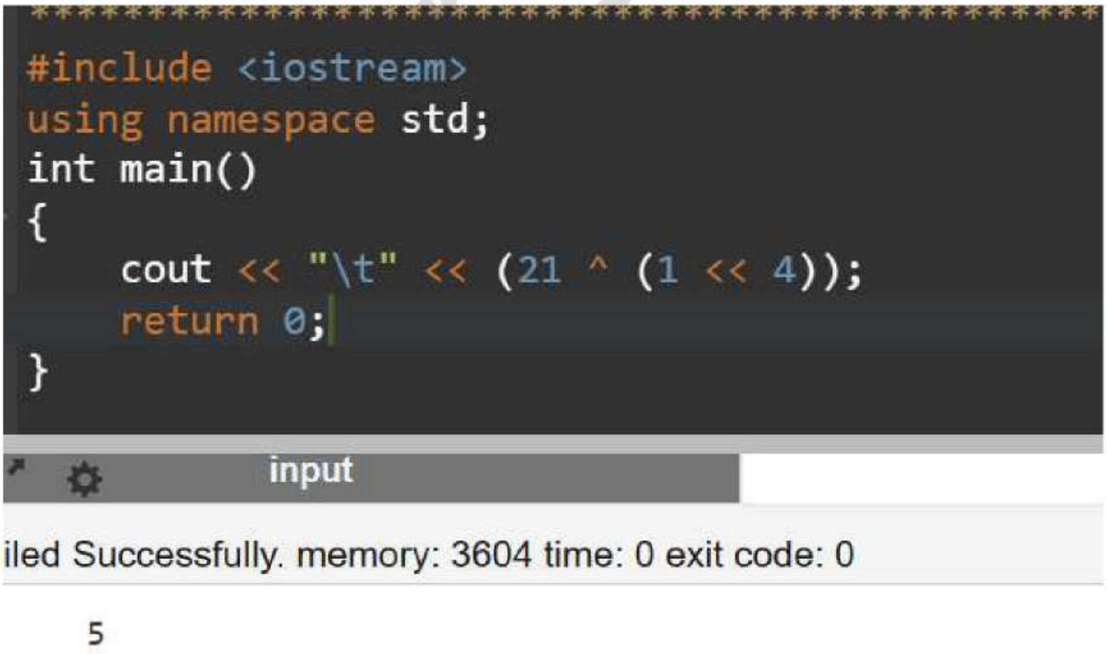
```
piled Successfully. memory: 3496 time: 0.01 exit code: 0

    5
```

বিটকে ট্রগল করা (ওপেন থাকলে ক্লোসড, ক্লোসড থাকলে ওপেন)

যে K বিটকে ট্রগল করতে চাইবে ১ কে সে পর্যন্ত লেফট শিফট (1 << K) করে জর (^) পারফর্ম করলে আমাদের ভ্যালু ট্রগল হয়ে যাবে।



আমরা এখানে ২১ এর ৪ তম বিটকে ট্রগল করব এটা করার ফলে আমাদের আউটপুট ৫ আসবে। যেমনঃ



```
#include <iostream>
using namespace std;
int main()
{
    cout << "\t" << (21 ^ (1 << 4));
    return 0;
}
```

```
iled Successfully. memory: 3604 time: 0 exit code: 0

	5
```