**বিট-ওয়াইজ অপারেটর (BITWISE OPERATOR)**

Standard

বিটওয়াইজ অপারেটর এবং বিটওয়াইজ অপারেশনগুলো প্রোগ্রামিং এর ক্ষেত্রে অনেক গুরুত্বপূর্ণ ভূমিকা পালন করতে পারে যার মূল কারণ হল এই অপারেটরের কাজগুলো কম্পিউটার অত্যন্ত দ্রুত সম্পন্ন করতে পারে। যোগ/বিয়োগ/গুণ/ভাগ ইত্যাদির চেয়ে একটা বিটের মান পরিবর্তন করে দেওয়া কম্পিউটারের জন্য অনেক সহজ কাজ। তাই অনেকক্ষেত্রেই বিট-ওয়াইজ অপারেটর ব্যবহারের মাধ্যমে হিসাব সম্পন্ন করলে অনেক সময় বাঁচানো যায়, তথা প্রোগ্রামের এফিশিয়েন্সি অনেকাংশে বৃদ্ধি করা যায়।

কম্পিউটার যেকোন তথ্যকে নাম্বার হিসেবে মেমরিতে রাখে। আর এই নাম্বারগুলো থাকে দ্বিমিক আকারে, অর্থাৎ ২ ভিত্তিক নাম্বার বা বাইনারি নাম্বার হিসেবে। বাইনারিতে কোন নাম্বারকে প্রকাশ করার জন্য কেবল দুটি ডিজিট রয়েছে, 0 এবং 1 । প্রতিটি 0 বা 1 হল একেকটি বিট। বিট-ওয়াইজ অপারেটরগুলো সমগ্র নাম্বার নিয়ে কাজ না করে এই বিটগুলোর একেকটির উপর পৃথক পৃথকভাবে কাজ করে। যেহেতু একটি বিটে কেবল 0 এবং 1 এই দুটি ডিজিটের একটি থাকতে পারে, তাই যেকোন বিট-ওয়াইজ অপারেটর দিয়ে যাই করা হোক না কেন, শেষ পর্যন্ত একটি বিট হয় পরিবর্তিত হয়ে 0/1 হবে নতুবা যা ছিল তাই থাকবে।

সি ল্যাঙ্গুয়েজে ৬টি বিট-ওয়াইজ অপারেটর রয়েছে, যথা -

১) & (AND) ৪) ^ (XOR)

২) | (OR) ৫) << (LEFT SHIFT)

৩) ~ (NOT) ৬) >> (RIGHT SHIFT)

**AND অপারেটরঃ**

বিট-ওয়াইজ AND অপারেটরটি লজিকাল AND (&&) এর মতই কাজ করে। পার্থক্য হল, এটি প্রতিটি বিটের উপর আলাদা আলাদাভাবে কাজ করে। আমরা জানি কম্পিউটারে সত্য এবং মিথ্যা এই দুটি তথ্য যথাক্রমে 1 এবং 0 হিসেবে থাকে। 1 মানে সত্য, 0 মানে মিথ্যা। একইভাবে যখন কোন বিটে 1 থাকে সেই বিটটি সত্য বোঝায় এবং 0 থাকলে সেটি মিথ্যা বোঝায়। যখন দুটি নাম্বারের মধ্যে বিট-ওয়াইজ AND অপারেশন করা হয় তখন নাম্বার দুটির বিটগুলোর প্রতিটির মধ্যে AND অপারেশন হয় এবং সেখান থেকে নতুন একটি নাম্বার পাওয়া যায়। ঘটনাটি বোঝার জন্য একটি উদাহরণ দেখা যাক।

ধরা যাক আমাদের কাছে দুটি unsigned int আছে। হিসাবের সুবিধার জন্য আমরা unsigned int এর সাইজ ৩২ বিটের জায়গায় ধরে নিচ্ছি ৮ বিট। এখন ধরা যাক একটি নাম্বার 78 এবং আরেকটি নাম্বার 45। আমরা যদি নাম্বার দুটিকে বাইনারিতে রূপান্তর করি তাহলে তাদের বিট প্যাটার্ন পেয়ে যাব।

78 = 01001110  
45 = 00101101

এখন যদি আমরা এদের মধ্যে বিট-ওয়াইজ AND অপারেশন করি এবং সেটিকে x নামক একটি ভেরিয়েবলে রাখি তাহলে তার সিন্ট্যাক্স হবে এরকম,

x = 78 & 45;

এখন যদি আমরা x এর ভেলু আউটপুট দেই, তাহলে পাব 12 । কিভাবে হল সেটি এদের বিট প্যাটার্ন লক্ষ্য করলেই বোঝা যাবে।

78 = 0100 1110

45 = 0010 1101

& -----------

x = 0000 1100 = 12

আমরা যদি একেবারে ডানের বিট থেকে দেখা শুরু করি এবং সেটিকে ১ম বিট বলি, তাহলে ব্যাপারটি এরকম দাঁড়ায়, ১ম বিটে আছে 78 এ 0 এবং 45 এ 1 । 0 এবং 1 এর মধ্যে AND করলে ফলাফল আসবে 0 । কারণ আমরা জানি, AND অপারেটর সত্য বা 1 রিটার্ন করে যখন এটি যাদের উপর কাজ করছে তারা উভয়ই সত্য হয়, অন্যথায় মিথ্যা বা 0 রিটার্ন করবে। এভাবে ২য় বিটে 1 এবং 0 AND হয়ে 0 হল। ৩য় বিটে 1 এবং 1 আছে। তাই এদের মধ্যে AND হয়ে 1 আসবে। একইভাবে বাকি ৫টি বিটেও একই ঘটনা ঘটছে। এভাবে করে প্রতিটি বিট আলাদা আলাদাভাবে AND অপারেশনের অন্তর্ভুক্ত হচ্ছে এবং সেই অনুযায়ী একটি বিট রিটার্ন করে সবশেষে নতুন একটি সংখ্যা তৈরি করছে।

**OR অপারেটরঃ**

বিট-ওয়াইজ AND অপারেটর কাজ বুঝতে পারলে বিট-ওয়াইজ OR অপারেটরের কাজ এমনিতেই বুঝতে পারা উচিৎ। মূল কাজটি আবারও একইভাবে হবে, প্রতিটি বিট অনুযায়ী। প্রতিটি বিটের উপর আলাদা আলাদাভাবে OR অপারেশন সংঘটিত হবে এবং সেখান থেকে একটি 0 বা 1 রিটার্ন হবে। AND অপারেটরের ক্ষেত্রে দুটি বিটই যদি 1 হয়ে কেবন তখন 1 রিটার্ন করে, অন্যথায় 0 রিটার্ন করে। সেখানে OR অপারেটরের ক্ষেত্রে দুটি বিটের যেকোন একটি সত্য বা 1 হলেই 1 রিটার্ন করবে। 0 রিটার্ন করবে শুধুমাত্র তখন যখন দুটি বিটই 0 হবে।

এখন যদি আমরা আবারও আগের দুটি নাম্বার বিবেচনা করি, 78 এবং 45, তাহলে দেখা যাক এদের মধ্যে OR অপারেশন করা হলে x এর মান কি হবে।

x = 78 | 45;

78 = 0100 1110

45 = 0010 1101

| -----------

x = 0110 1111 = 111

এবার আমরা x এর মান আউটপুট দিলে পাব 111 । বিট ধরে ধরে পর্যালোচনা করলে দেখা যাবে, ১ম বিটে আছে 0 এবং 1 যাদের মধ্যে OR করা হলে পাওয়া যাবে 1 । ২য় বিটে আছে 1 এবং 0 যাদের থেকে আবারও 1 পাওয়া যাবে। ৩য় বিটে আছে 1 এবং 1 যাদের থেকেও 1 পাওয়া যাবে। এভাবে করে ৫ম বিটে গেলে দেখা যাচ্ছে সেখানে আছে 0 এবং 0 যাদের মধ্যে OR করা হলে 0 রিটার্ন করবে। এভাবে সমগ্র নাম্বারের প্রতিটি বিট OR অপারেশনের মধ্যে দিয়ে যাবে এবং শেষ পর্যন্ত 111 এর বিট প্যাটার্ন পাওয়া যাবে।

**NOT অপারেটরঃ**

NOT অপারেটর কাজ অত্যন্ত সাধারণ। এটি একটি ইউনারি অপারেটর, অর্থাৎ এটির অপার‍্যান্ড একটি বা এটি একটি নাম্বারের উপর অপারেশন সংঘটিত করে। এর কাজ হল এটি যে নাম্বারের উপর অপারেশন চালায় সেই নাম্বারের প্রতিটি বিটকে ফ্লিপ করে দেয়। অর্থাৎ যে বিটে 0 আছে তাকে 1 বানায়, আর যে বিটে 1 আছে তাকে 0 বানিয়ে দেয়। আবারও ধরা যাক দুটি নাম্বার 78 এবং 45। এদেরকে বিট-ওয়াইজ NOT করা হলে কি হয় দেখা যাক।

x = ~78;

78 = 0100 1110

x = 1011 0001 = 177

x = ~45;

45 = 0010 1101

x = 1101 0010 = 210

এখানে আসলে আর বলার কিছু নেই। খুবই সহজ একটা ব্যাপার। যেখানে 0 আছে সেখানে 1 হবে আর যেখানে 1 আছে সেখানে 0 হবে।

**XOR অপারেটরঃ**

বিট-ওয়াইজ অপারেটরগুলোর মধ্যে একটি বিশেষ অপারেটর হল ^ (XOR) অপারেটর যাকে বলা হয় এক্সক্লুসিভ OR অপারেটর। এটি OR অপারেটরের মতই কাজ করে। তবে একটু বিশেষত্ব রয়েছে এতে। সেকারণেই নামের আগে ‘এক্সক্লুসিভ’ শব্দটা আছে! OR অপারেশনে যদি দুটি বিটের যেকোন একটি 1 হয় তাহলে 1 রিটার্ন করে। অতএব এটি 0 রিটার্ন করে কেবলমাত্র তখন যখন দুটি বিটই 0 হয়। সেখানে XOR অপারেটর 1 বা সত্য রিটার্ন করে শুধু মাত্র তখন যখন যেকোন একটি বিট, শুধুমাত্র একটি বিট 1 হয়। অর্থাৎ OR এ যেমন দুটি বিট 1 হলেও 1 রিটার্ন করত, এখন তেমনটি হবে না। XOR 1 রিটার্ন করে যখন দুটি বিটের একটি 1 এবং অপরটি 0 হয়। যদি দুটি বিটই 1 বা দুটি বিটই 0 হয় তাহলে 0 রিটার্ন করবে।

আবারও আমরা দুটি নাম্বার বিবেচনা করি, 78 এবং 45 ।

x = 78 ^ 45;

78 = 0100 1110

45 = 0010 1101

^ -----------

x = 0110 0011 = 99

১ম বিটে আছে 0 এবং 1 যাদের XOR করলে 1 পাওয়া যাবে। ২য় বিটেও একই। ৩য় বিটে দুটি বিটই 1 হওয়ায় XOR এর নিয়ম অনুযায়ী এদের থেকে 0 রিটার্ন হবে। ৪র্থ বিটেও একইভাবে 0 হবে। ৫ম বিটের দুটি বিটই 0 থাকায় সেটিও 0 হবে। এভাবে করে আগাতে থাকলে সমগ্র বিট প্যাটার্ন পাওয়া যাবে যেটি হবে 99 এর বিট প্যাটার্ন। ফলে x এর মান আউটপুট দিলে আমরা 99 পাব।

**LEFT SHIFT এবং RIGHT SHIFT অপারেটরঃ**

SHIFT অপারেটরগুলো কোন নাম্বারের সমগ্র বিট প্যাটার্নকে ডানে বা বামে শিফট করে দেয়। অর্থাৎ যতঘর বলা হবে এরা নাম্বারের বিটগুলোর প্রতিটিকে স্ব স্ব অবস্থান থেকে ততঘর ডানে বা বামে সরিয়ে দিবে। LEFT SHIFT বামে এবং RIGHT SHIFT অপারেটর ডানে সরাবে। SHIFT অপারেটরের সিন্ট্যাক্সে অপারেটরের বামে থাকে যে নাম্বারটিকে শিফট করা হবে সেটি এবং ডানে থাকে যতঘর শিফট করা হবে সেই নাম্বার। ধরা যাক আমরা 78 কে 1 ঘর বামে শিফট করতে চাই। তাহলে ব্যাপারটি কিরকম হবে দেখা যাক।

x = 78 << 1;

78 = 0100 1110

x = 1001 1100 = 156 // একঘর বামে শিফট করার পর

লক্ষ্য করলেই বোঝা যাবে, যেটি এখানে হয়েছে তা হল, প্রতিটি বিট একঘর করে বামে সরে গিয়েছে। যখন LEFT SHIFT অপারেটরের মাধ্যমে বিটগুলোকে বামে সরানো হয় তখন একেবারে ডানে যে ফাঁকা বিট থেকে যায় সেটিতে 0 বসে যায়। আর একেবারে বামের যে বিটগুলো থাকে যেগুলোকে শিফট করার পর আর জায়গা থাকে না বসানোর সেগুলো হারিয়ে যায়। যেমন এক্ষেত্রে একেবারে বামের 0 হারিয়ে গিয়েছে। যদি আমরা 78 কে ২ ঘর বামে শিফট করতাম তাহলে ব্যাপারটি এমন হত,

x = 78 << 2;

78 = 0100 1110

x = 0011 1000 = 56 // দুইঘর বামে শিফট করার পর

এখানে পরিষ্কার বোঝা যাচ্ছে ২ ঘর বামে শিফট করার ফলে বামের দুটি বিট হারিয়ে গিয়েছে, যার মধ্যে একটি বিট 1 ছিল।

একইভাবে RIGHT SHIFT অপারেশন করা হলে যতঘর বলা থাকে নাম্বারটির বিটগুলো ততঘর ডানে সরে যায়। LEFT SHIFT এ যেমন বামের বিটগুলো হারিয়ে যায়, RIGHT SHIFT এ ডানের বিটগুলো হারিয়ে যায় এবং বামে 0 যোগ হতে থাকে।

x = 45 >> 2;

45 = 0010 1101

x = 0000 1011 => 11 // দুইঘর ডানে শিফট করার পর

এই হল বিট-ওয়াইজ অপারেটর সম্পর্কিত ধারণা, কোনটি কিভাবে কাজ করে। তবে এখানেই শেষ নয়। এই অপারেটরগুলোকে নানা ভাবে নানা কৌশলে কাজে লাগিয়ে অনেক সহজেই বিভিন্ন হিসাব সম্পন্ন করা যায়। এতে প্রোগ্রামের রানটাইমও অনেকখানি কমে যায় যেটি কিছু কিছু ক্ষেত্রে এসিএম প্রোগ্রামিং এ TLE এবং AC এর মধ্যে পার্থক্য করে দিতে পারে।

আগের পর্বঃ [বিট-ওয়াইজ অপারেটর](http://binaryrongo.wordpress.com/2013/07/27/bitwise-operator/)

এর আগে বিট-ওয়াইজ অপারেটরগুলো নিয়ে লিখেছিলাম। এবার এগুলোর কিছু ব্যবহার নিয়ে লিখছি যেগুলোর মাধ্যমে প্রোগ্রাম অনেক এফিসিয়েন্ট করা যায়। তবে এগুলো বোঝার জন্য বিট-ওয়াইজ অপারেটরগুলো কিভাবে কাজ করে তা জানা আবশ্যক। জানা না থাকলে [আগের পর্বটি](http://binaryrongo.wordpress.com/2013/07/27/bitwise-operator/) পড়তে অনুরোধ করছি।

এই লেখায় যে বিট প্যাটার্নগুলো দেখানো হবে সবক্ষেত্রে সবার ডানের বিটটি হল Least Significant Bit (LSB), অর্থাৎ ডান দিকের বিট থেকে নাম্বারিং শুরু হবে। 1 হবে 0000 0001, 2 হবে 0000 0010, 3 হবে 0000 0011 এরকম। আর বেশিরভাগ ক্ষেত্রেই আমরা unsigned ডাটা টাইপ নিয়ে কাজ করব। কারণ signed টাইপের ডাটার মধ্যে সবার বামের বিটটিতে (যদি ডানের বিট LSB হয়) চিহ্ন নির্দেশ করার জন্য একটি বিট বরাদ্দ থাকে।

**জোড়-বিজোড় নির্ণয়**

কোন একটি সংখ্যা জোড় নাকি বিজোড় সেটি নির্ণয় করার জন্য আমরা সাধারণত তাকে 2 দিয়ে mod করি, বা 2 দিয়ে ভাগ করলে ভাগশেষ কত হয় সেটি বের করি। যদি সেটি 0 হয় তাহলে বলি নাম্বারটি জোড়, না হলে বিজোড়। কিন্তু কম্পিউটারের জন্য এই কাজটি বেশ সময়সাপেক্ষ। আমরা বিট-ওয়াইজ অপারেটরের মাধ্যমে সহজেই অনেক দ্রুত কাজটি করতে পারি।

যেকোন নাম্বার কম্পিউটারে বাইনারিতে থাকে। কোন একটি নাম্বারকে যদি আমরা বাইনারিতে রূপান্তর করি তাহলে লক্ষ করলে দেখা যাবে, নাম্বারটি যদি জোড় হয় তাহলে তার একেবারে ডানের বিটটি 0 হয়। আর যদি বিজোড় হয় তাহলে সেটি 1 হয়। এটি যেকোন নাম্বারের ক্ষেত্রেই প্রযোজ্য। যেমন,

59 = 0011 1011

46 = 0010 1110

113 = 0111 0001

28 = 0001 1100

এখন আমরা যদি একটি নাম্বারকে 1 দিয়ে AND করি তাহলে নাম্বারটির একেবারে ডানের বিটে যদি 1 থাকে তাহলে AND হয়ে 1 আসবে, আর যদি ডানের বিটে 0 থাকে তাহলে AND হয়ে 0 আসবে। এখান থেকে আমরা সহজেই বলে দিতে পারি নাম্বারটি জোড় নাকি বিজোড়।

59 = 0011 1011           46 = 0010 1110

1 = 0000 0001            1 = 0000 0001

& ----------- & -----------

= 0000 0001              = 0000 0000

কোড লিখলে হবে এরকমঃ

|  |  |
| --- | --- |
| 1  2  3  4 | int x;  scanf( "%d", &x );  if( x & 1 ) puts( "Odd." );  else puts( "Even." ); |

**2^n দ্বারা গুণ/ভাগ**

কোন একটা নাম্বারকে যদি আমরা 2^n দিয়ে গুণ বা ভাগ করতে চাই তাহলে সহজ উপায় হল সরাসরি গুণ বা ভাগ করে দেওয়া। কিন্তু এখন আমাদের উদ্দেশ্যই হল কাজটি দ্রুত এবং এফিসিয়েন্টভাবে করা। গুণ-ভাগ অত্যন্ত সময় সাপেক্ষ প্রক্রিয়া। তাই আমরা বিট-ওয়াইজ অপারেটর ব্যবহার করে এটি করব।

ধরা যাক আমাদের কাছে একটা নাম্বার  x আছে। এখন যদি এই নাম্বারটিকে 2^n দিয়ে গুণ করতে বলা হয়, অথবা 2 দিয়ে n সংখ্যকবার গুণ করতে বলা হয় তাহলে আমরা যেটা করতাম তা হল, 2^n দেওয়া থাকলে সরাসরি গুণ করতাম, আর যদি n দেওয়া থাকে তাহলে একটা লুপ চালিয়ে n সংখ্যকবার 2 দিয়ে গুণ করতাম। একইভাবে ভাগের কাজটিও করতাম। কিন্তু এই একই কাজ করা যায় LEFT SHIFT এবং RIGHT SHIFT অপারেটরের মাধ্যমে।

আমরা যদি 2^n দিয়ে x কে গুণ করতে চাই তাহলে লিখব, x << n আর যদি ভাগ করতে চাই তাহলে x >> n । এতোটাই সহজ! ধরা যাক x = 5 এবং 2^n = 8 বা n = 3 । তাহলে,

x = 5 = 0000 0101

x << 3 = 0010 1000 = 40 = 5 \* 8 = 5 \* 2^3

আবার ধরা যাক, x = 96 এবং 2^n = 8 বা n = 3 । তাহলে,

x = 96 = 0110 0000

x >> 3 = 0000 1100 = 12 = 96 / 8 = 96 / 2^3

**2^n কিনা?**

একটি নাম্বার কি 2 এর পাওয়ার কিনা সেটি যদি বের করতে হয় তাহলে আমরা কিভাবে করব? সহজ উপায় হল একটা লুপ চালানো। 2 দিয়ে বারবার ভাগ করতে থাকব। যদি কখনও ভাগশেষ 0 বাদে অন্যকিছু আসে তার মানে সেটি 2 এর পাওয়ার না। অন্যথায় যদি ভাগ করতে করতে আমরা একসময় 2 পাই তাহলে বুঝব নাম্বারটি 2 এর পাওয়ার।

আরেকভাবেও করা যায়, আমরা একটি ভেরিয়েবলে 2 নিয়ে লুপ চালাবো এবং বার বার 2 দিয়ে গুণ করতে থাকব। যতক্ষণ পর্যন্ত নাম্বারটি প্রদত্ত নাম্বার থেকে ছোট থাকবে ততক্ষণ পর্যন্ত গুণ করতে থাকব। যদি কোন একসময় প্রদত্ত নাম্বারের সমান হয়ে যায় তাহলেই বুঝব নাম্বারটি 2 পাওয়ার। আর যদি সমান না হয়, প্রদত্ত নাম্বারের চেয়ে বড় হয়ে যায় তাহলে লুপ break করে দিব এবং নাম্বারটি যে 2 এর পাওয়ার না সেটি পেয়ে যাব। এরকম আরও অনেকভাবে করা যায়। তবে বিট-ওয়াইজ অপারেটরের সাহায্যে আমরা কোন লুপ না চালিয়ে একটা স্টেটমেন্ট দিয়েই এটি বের করতে পারি।

যেকোন নাম্বার যদি 2 এর পাওয়ার হয় তাহলে তার বিট প্যাটার্ন হবে, 1 এবং তার ডানে, 2 এর পাওয়ার যত ততগুলো 0। কিছু নাম্বার দেখা যাক,

 4 = 2^2 = 0000 0100

8 = 2^3 = 0000 1000

16 = 2^4 = 0001 0000

32 = 2^5 = 0010 0000

এই নাম্বারগুলো, অর্থাৎ 2 এর পাওয়ার যেগুলো তাদের থেকে যদি 1 বিয়োগ করি তাহলে যে নাম্বারটি পাওয়া যাবে তার বিট প্যাটার্ন হবে যত পাওয়ারের নাম্বার থেকে 1 বিয়োগ করা হয়েছে, ততগুলো 1। যেমন,

 3 = 2^2 - 1 = 0000 0011

7 = 2^3 - 1 = 0000 0111

15 = 2^4 - 1 = 0000 1111

31 = 2^5 - 1 = 0001 1111

এই প্যাটার্ন দেখে পরিষ্কার হওয়া উচিত যে আমরা যদি x এর সাথে x-1 এর AND করি, তাহলে x যদি 2 এর কোন পাওয়ার হয় সেক্ষেত্রে আমরা 0 পাব। অন্যথায় 0 ব্যতীত অন্য কোন নাম্বার পাব।

8 = 2^3     = 0000 1000

7 = 2^3 - 1 = 0000 0111

-----------------------

8 & ( 8-1 ) = 0000 0000

10 = 0000 1010

9 = 0000 1001

& ------------

= 0000 1000 ( 0 নয় )

অতএব কোড লিখলে দাঁড়াবেঃ

|  |  |
| --- | --- |
| 1  2  3  4 | unsigned int x;  scanf( "%u", &x );  if( x & (x-1) ) puts( "Not a power of 2." );  else puts( "Power of 2." ); |

**2^n – 1 কিনা?**

আগের অংশটুকু বুঝলে এটি খুবই সহজ। আগেরবার আমরা x এবং x-1 এর মধ্যে AND করেছি। এবার x এবং x+1 এর মধ্যে AND করব। তাহলেই যদি ফলাফল 0 আসে, নাম্বারটি 2^n – 1 টাইপ, অন্যথায় হবে না। কিভাবে হয় তার ব্যাখ্যা উপরের মতই। বাকিটুকু নিজে থেকে চিন্তা করার দায়িত্ব রইল। এবার কোড দেখা যাকঃ

|  |  |
| --- | --- |
| 1  2  3  4 | unsigned int x;  scanf( "%u", &x );  if( x & (x+1) ) puts( "Not 2^n - 1 type." );  else puts( "2^n - 1 type." ); |

**2^n দ্বারা বিভাজ্য কিনা?**

একটি নাম্বার যদি দেওয়া থাকে তাহলে সেটি কি 2 এর কোন পাওয়ার দ্বারা নিঃশেষে বিভাজ্য কিনা তা আমরা কিভাবে বের করব? সহজ উত্তর হল আবারও সেই mod (%) ব্যবহার করা। কিন্তু আবারও একই কথা, এই অপারেটরটির কাজ অনেক সময়সাপেক্ষ। এই জন্য আমরা বিট-ওয়াইজ অপারেটরের মাধ্যমে কাজটি করব।

কোন একটি নাম্বার x দেওয়া আছে। আরেকটি নাম্বার দেওয়া আছে d যাকে দিয়ে আমরা x কে ভাগ করে দেখবো ভাগশেষ 0 হয় কিনা। এখন d কে যদি আমরা 2^n আকারে লিখতে পারি, অর্থাৎ d যদি 2 এর কোন পাওয়ার হয় তাহলে, x % d = x & ( d – 1 ) হবে। এখানে d কে অবশ্যই 2^n টাইপের একটি নাম্বার হতে হবে, অন্যথায় সমীকরণটি ঠিক হবে না।

নাম্বারগুলোর বিট প্যাটার্ন দিয়ে পরীক্ষা করে দেখা যাক। x = 74, d = 8, x % d = 6 ।

x    = 74 = 1010 1110

d - 1 = 7 = 0000 0111

& --------------------

6  = 0000 0110

আবার, x = 96, d = 8, x % d = 0 ।

x = 96 = 0110 0000

d - 1 = 7 = 0000 0111

& --------------------

0 = 0000 0000

অতএব কোডে আমরা লিখতে পারিঃ

|  |  |
| --- | --- |
| 1  2  3  4 | unsigned int x, d = 8;  scanf( "%u", &x );  if( x & ( d - 1 ) ) puts( "x is not divisible by d." )  else puts( "x is divisible by d." ); |

**2^n দ্বারা mod**

আগের অংশটুকু থেকেই আসলে এটা বলা হয়ে গেছে। d কে যদি আমরা 2^n আকারে লিখতে পারি, অর্থাৎ d যদি 2 এর কোন পাওয়ার হয় তাহলে, x % d = x & ( d – 1 ) হবে। এখানে d কে অবশ্যই 2^n টাইপের একটি নাম্বার হতে হবে, অন্যথায় সমীকরণটি ঠিক হবে না। আগের অংশের উদাহরণ থেকেই এটা পরিষ্কার।

**দুটি ভেরিয়েবলের মান পাল্টাপাল্টি (৩য় ভেরিয়েবল ছাড়া)**

সাধারণত দুটি ভেরিয়বলের মান পাল্টাপাল্টি (swap) করার জন্য আমরা ৩য় একটি ভেরিয়েবলের সাহায্য নিয়ে থাকি। ১ম ভেরিয়েবলকে ৩য়টিতে রাখি, তারপর ২য়টিকে ১মটিতে রাখি, তারপর ৩য়টি থেকে ২য়টিতে রাখি। কিন্তু যদি আমরা ৩য় ভেরিয়েবলটির সাহায্য ছাড়াই কাজটি করতে চাই? তাহলেও একটা সহজ উপায় আছে, যোগ-বিয়োগ অথবা গুণ-ভাগের মাধ্যমে। দেখা যাক পদ্ধতি দুটি।

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | unsigned int x = 21;  unsigned int y = 13;    // যোগ-বিয়োগের মাধ্যমে  x = x + y; // x = 21 + 13 = 34  y = x - y; // y = 34 - 13 = 21  x = x - y; // x = 34 - 21 = 13    // গুণ-ভাগের মাধ্যমে  x = x \* y; // x = 21 \* 13 = 651  y = x / y; // y = 351 / 13 = 21  x = x / y; // x = 351 / 21 = 13 |

কিন্তু আবারও কথা একই, সময় বেশি লাগবে! সময় কমানোর জন্য এখন আমরা একই কাজ বিট-ওয়াইজ অপারেটর XOR এর মাধ্যমে করব।

|  |  |
| --- | --- |
| 1  2  3 | x = x ^ y;  y = x ^ y;  x = x ^ y; |

কিভাবে হল সেটার জন্য বিট প্যাটার্ন দেখা যাক।

x = 21 = 0001 0101

y = 13 = 0000 1101

x = x ^ y = 0001 1000

y = x ^ y = 0001 0101 = 21

x = x ^ y = 0000 1101 = 13

এভাবে বিট-ওয়াইজ অপারেটরসমূহকে নানাভাবে কাজে লাগিয়ে বিভিন্ন গাণিতিক সমস্যার সমাধান করা যায়। যদিও কাজগুলো সাধারণ অপারেটরগুলো দিয়েও করা যায়, বিট-ওয়াইজ অপারেটর ব্যবহারের মূল উদ্দেশ্য হল প্রোগ্রামের গতি বৃদ্ধি করা। অনেকসময় এই সাধারণ কিছু পরিবর্তন প্রোগ্রামিং কনটেস্টে TLE এবং AC এর মধ্যে পার্থক্য সৃষ্টি করে দিতে পারে!