এক্সপোনেন্ট ম্যানটিসা

তূর্য আহসান

গণিত এবং বিজ্ঞানে অনেক সময় দশমিক সংখ্যা (ডেসিমাল নাম্বার) নিয়ে বিভিন্ন ধরণের হিসাব করতে হয়। উদাহরণ হিসেবে প্লাংক কনসট্যান্ট দেখা যাক। প্ল্যাংক কনসট্যান্ট হচ্ছে (৬.৬৩ * ১০^-৩৪)। এধরণের সংখ্যার প্রথম বিট দিয়ে নন-জিরো অংশকে প্রকাশ করা হয়। এই প্রকাশকে বলা হয় ম্যানটিসা। অর্থাৎ প্লাংক কনসট্যান্ট এর ৬.৬৩ হল ম্যানটিসা। আর দশমিকের পর কতগুলো ঘর যেতে হবে সেটা এক্সপোনেন্ট দ্বারা প্রকাশ করা হয়। অর্থাৎ ১০^-৩৪ হল এক্সপোনেন্ট। প্লাংক কনসট্যান্টকে কেবল সংখ্যা দিয়ে প্রকাশ করতে হলে লিখতে হবে ০.০০ ০০০ ০০০ ০০০ ০০০ ০০০ ০০০ ০০০ ০০০ ০০০ ০০০ ৬৬৩।

আধুনিক যুগে গণিত এবং বিজ্ঞানের গাণিতিক পদ্ধতিসমূহ কমপিউটার এর মাধ্যমে করা হয়ে থাকে। ফলে কমপিউটারকে দশমিক সংখ্যা নিয়েও কাজ করতে হয়। কমপিউটার যেহেতু বাইনারি সংখ্যা নিয়ে কাজ করে, তাই প্লাংক কনসট্যান্টকে কমপিউটার দ্বারা স্টোর বা ম্যানিপুলেট করতে হলে প্ল্যাংক কনসট্যান্টকে বাইনারিতে কনভার্ট করে ব্যাবহার করতে হবে। সি প্রোগ্রামিং ল্যাঙ্গুয়েজে এই ধরণের দশমিক সংখ্যার জন্য ২ ধরণের ডেটা টাইপ আছে- ফ্লোট ও ডাবল। ডঃ উইলিয়াম মর্টন কাহানকে ফ্লোটিং পয়েন্ট এর জনক বলা হয়।

ফ্লোটিং পয়েন্ট নাম্বার প্রদর্শণের জন্য নিচের ফর্মূলা ব্যাবহার করা হয়ঃ

$$-1^S \times M \times 2^E$$

এখানে, S হচ্ছে সাইন, M হচ্ছে ম্যানটিসা এবং E হচ্ছে এক্সপোনেন্ট।

সিঙ্গেল প্রিসিশন ফ্লোটিং পয়েন্ট নাম্বারের সাইজ হচ্ছে ৩২ বিট সংখ্যার, যার-

৩১ তম বিট হচ্ছে সাইন, সাইজ হচ্ছে ১ বিট,

২৩-৩০ তম বিট হচ্ছে এক্সপোনেন্ট, সাইজ হচ্ছে ৮ বি্

০-২২ তম বিট হচ্ছে ম্যানটিসা, সাইজ হচ্ছে ২৩ বিট।

কিন্তু সিঙ্গেল প্রিসিশন ফ্লোটিং পয়েন্ট নাম্বার ব্যাবহারের অসুবিধা হচ্ছেঃ

যদি এক্সপোনেন্ট এর মান এতই বড় হয় যে এক্সপোনেন্ট ফিল্ড দ্বারা প্রকাশ করা সম্ভব না তবে ওভারফ্লো হয়।

আর যদি সংখ্যাটা এতই ক্ষুদ্র হয় যে এক্সপোনেন্ট ফিল্ড দ্বারা প্রকাশ সম্ভব না, তবে আন্ডারফ্লো হয়।

ওভারফ্লো বা আন্ডারফ্লো সমস্যা সমাধানের জন্য ডাবল-প্রিসিশন ফ্লোটিং পয়েন্ট নাম্বার ব্যাবহার করা হয় যার সাইজ হচ্ছে ৬৪ বিট, যেখানে-

৬৩ তম বিট হচ্ছে সাইন, সাইজ হচ্ছে ১ বিট,

৫২-৬২ তম বিট হচ্ছে এক্সপোনেন্ট, সাইজ হচ্ছে ১১ বিট,

০-৫১ তম বিট হচ্ছে ম্যানটিসা, সাইজ হচ্ছে ৫২ বিট।

ফ্লোটিং পয়েন্ট নাম্বার কনভার্সনঃ

ফ্লোটিং পয়েন্ট নাম্বারকে প্রদর্শনের করার কয়েকটা ধাপ আছে। ধাপগুলো হলঃ

১/ সাইন - **ম্যানটিসার সাইন, ধণাত্মক** (+) /**ঋণাত্মক** (-), **নির্ধারণ করা হয়। ১৬ বিট নাম্বারের জন্য ম্যানটিসা ১০ বিট। তার মধ্যে ১ম বিট দিয়ে সাইন প্রদর্শন করা হয়। ১ম ও ২য় বিটের মধ্যে দশমিক এর অবস্থান।**

২ *স্লাইড* - **এক্সপোনেন্ট এর মান নির্ণয় করা হয় এবং সেটা ধণাত্মক** (+) /**ঋণাত্মক** (-) **সেটা নির্ণয় করা হয়। ১৬ বিট নাম্বারের জন্য এক্সপোনেন্ট ৬ বিট দ্বারা দেখানো হয়।**

৩/ বাউন্স - **স্লাইড দ্বারা নির্ধারিত ঘর পর্যন্ত ডেসিমাল স্থান নেয়া হয়, ঋণাত্মক এক্সপোনেন্ট হলে বামদিকে এবং ধণাত্মক এক্সপোনেন্ট হলে ডানদিকে স্থাণাংক দেয়া হয়। যদি ধণাত্মক সংখ্যার বামদিকে দশমিক স্থাণাংক সরতে থাকে তবে ০ দিয়ে প্যাড করা হয়। যদি ঋণাত্মক সংখ্যার বামদিকে দশমিক স্থাণাংক সরতে থাকে তবে ১ দিয়ে প্যাড করা হয়।**

৪/ ফ্লিপ - **ম্যানটিসা ঋণাত্মক হলে টু'স কমপ্লিমেন্ট** (Two's compliment) **করা হয়।**

৫/ স্যুইম - **ডেসিমাল পয়েন্ট হতে শুরু করে প্রথমে বাম স্থানে এবং পরে ডান স্থানে ম্যানটিসার মাণ বসানো হতে থাকে।**

মেমরিতে নাম্বার রাখার প্রক্রিয়াঃ

ধরা যাক -১.৫ কে মেমরিতে রাখা হবে। বাইনারিতে রূপান্তর করলে হয় -১.১। -১.৫ কে লেখা যায় -১.৫ * ১০^০। একইভাবে -১.১ কে লেখা যায় -১.১ * ২^০। এখানে নাম্বারটিকে নরমালাইজ করে নিতে হবে। অর্থাৎ -০.০০০১৫ কে লেখা হবে -১.৫ * ১০^-৪। বাইনারির ক্ষেত্রেও নরমালাইজ করতে হবে। নরমালাইজ করে লেখা এই বাইনারি নাম্বারের তিনটা অংশ আছে। প্রথমে সাইন (+/-), এরপর ১.১ মূল নাম্বারটা এবং তারপর ২^০। প্রত্যেকটি নাম্বারকে এরকমভাবে প্রকাশ করা যায়। একটা জিনিস খেয়াল করা দরকার, ১.১ যে অংশটা আছে, যেহেতু নাম্বারটি নরমালাইজ করা হয়েছে সেহেতু সবসময় দশমিকের বামে একটা ১ থাকবে। এরজন্য নাম্বারটির যে পরিবর্তন হয় সেই অনুযায়ী ২ এর পাওয়ার পরিবর্তন করে নেওয়া হয় যেন সমগ্র নাম্বারের মানের কোন পরিবর্তন না ঘটে। আরেকটা জিনিস হল যে নাম্বারটি যেহেতু বাইনারি, অতএব সবসময় ২ এর কোন একটা পাওয়ার দিয়ে গুণ হবে, যেমন ডেসিমাল নাম্বারের ক্ষেত্রে ১০ এর পাওয়ার দিয়ে গুণ হয়। এখন তাহলে নাম্বাররের মধ্যে যে পরিবর্তন যোগ্য অংশগুলো থাকে তা হল সাইন, দশমিকের ডান পাশের অংশটুকু, এবং ২ এর পাওয়ার কত সেটি। মেমরিতে মূলত এই তিনটি তথ্যই রাখা থাকে। সাইন বিট (S), ২ এর পাওয়ার কত হবে (E) এবং . এর ডানে কি থাকবে (M)। S এর মান ০/১ হয় নাম্বারটি পজিটিভ নাকি নেগেটিভ তার উপর ভিত্তি করে। E এর মান হল নাম্বারটিকে ২ এর যে পাওয়ার দিয়ে গুণ করা হচ্ছে সেটি। তবে মেমরিতে রাখার সময় সেটির সাথে বায়াস (b) যোগ করে তারপর রাখা হয়। b এর মান ৩২ বিট এ ১২৭ এবং ৬৪ বিটে ১০২৩। এরপর M এ যা থাকে তা হল নাম্বারটির দশমিকের পর যেই বিটগুলো আছে সেগুলো।

সমস্যাঃ

অসীম ধারাঃ

ফ্লোটিং পয়েন্ট নাম্বারের যে সব সমস্যা আছে তার মধ্যে একটা বড় সমস্যা হল অসীম ধারা। এক-তৃতীয়াংশ, একে সহজেই বাংলায় ১/৩ লিখে গণিতে ব্যবহার করা যায়। কিন্তু যখন এর দশমিক রূপ দেখা হবে, তখনই দেখা যায় এটা একটা অসীম ধারা, ০.৩৩৩৩৩... (অসীম পর্যন্ত)। এই ধরনের নাম্বার দেখেই বোঝা যাচ্ছে যে একে বাইনারিতে রূপান্তর করে নিখুঁতভাবে মেমরিতে রাখা সম্ভব না। কারণ মেমরির সীমা আছে, কিন্তু এই নাম্বারগুলোর সীমা নেই। আরেকটি উদাহরণ সংখ্যা- ০.১। ০.১ এর বাইনারি হল ০.০০০১১০০১১০০১১০০১১... (অসীম পর্যন্ত)। এইসব নাম্বার মেমরিতে রাখতে গেলে কিছু ডেটা হারিয়ে যায়। এই কারনে, নাম্বারের প্রিসিশন নষ্ট হয়।

একটা কোড থেকে এর বাস্তব সমস্যা দেখা যাক-

```
double value = 0.1;

printf( "%.17lf", value );

Output: 0.10000000000000001
```

দেখা যাচ্ছে, স্টোর করা হল ০.১, কিন্তু আউটপুট ০.১ এর চেয়ে বেশি এসেছে।

আরেকটি কোড দেখা যাক-

```
double value = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;

printf( "%.16lf", value );

Output: 0.9999999999999999
```

এখানে, ১০ টা ০.১ যোগ করে একটা ডাবল ভেরিয়েবলে রাখা হয়েছে। ১০ টা ০.১ এর যোগফল ১.০ হওয়ার কথা। কিন্তু, আউটপুটে ১.০ এর চেয়ে একটু কম এসেছে। অর্থাৎ, ফ্লোটিং পয়েন্ট নাম্বার নিয়ে কাজ করতে গেলে প্রিসিশনে সমস্যা থাকে। ঠিক যে নাম্বার পাওয়া যাওয়ার কথা, বা যে নাম্বার মেমরিতে থাকার কথা ঠিক সেটি থাকে না। একটু কম বা বেশি হয়।

ফ্লোটিং পয়েন্ট নাম্বারের তুলনা করাঃ

আরেকটা কোড দেখা যাক-

```
float value1 = 1.345;

float value2 = 1.123;

float total = value1 + value2; // the value should be 2.468

if( total == 2.468 )

printf( "equal" );

else

printf( "not equal");

Output: not equal
```

এখানেও প্রিসিশনে সমস্যা। দশমিকের পর মাত্র তিন ঘর হলেও মেমরিতে তো আর তিন ঘর থাকে না। পুরোটাই থাকে এবং শেষের দিকের ডিজিট যে উলটাপালটা হয় তা তো একটু আগেই দেখা গেল। এই কারণে দুইটা সমান আসছে না। অতএব তুলনা করার প্রক্রিয়া পাল্টাতে হবে।

এপসাইলনঃ

নতুন যে প্রক্রিয়ায় তুলনা করা হবে সেখানে সরাসরি দুইটা নাম্বার সমান কিনা তুলনা করা হয় না। যেটি করা হয় তা হল দুইটা নাম্বার কি একটা আরেকটার যথেষ্ট কাছাকাছি কিনা তা দেখা হয়। আর এই যে যথেষ্ট কাছাকাছি কিনা তা দেখার জন্য একটা অতি ক্ষুদ্র নাম্বারের সাহায্য নেওয়া হয়। যদি নাম্বার দুইটার ব্যবধান ঐ ক্ষুদ্র নাম্বারটির চেয়ে ছোট হয় তাহলে ধরে নেওয়া হয় তারা যথেষ্ট কাছাকাছি, এবং তাদেরকে তখন সমান বলে গণ্য করা হয়। ব্যাপারটা সহজ করে বোঝানোর জন্য ধরা যাক দুটি নাম্বার আছে আমাদের কাছে ১০০০০ এবং ১০০০১। এখন এদের ব্যবধান যদি একটি অতি ক্ষুদ্র নাম্বার (এক্ষেত্রে ধরা যাক ২) এর চেয়ে ছোট হয় তাহলে ধরে নিতে হবে নাম্বার দুটি সমান কারণ নাম্বার দুটির তুলনায় ২ অনেক ছোট। বাস্তবে কিন্তু নাম্বার দুটি সমান না, কিন্তু ধরে নেয়া হবে এরা যথেষ্ট কাছাকাছি, তাই এরা সমান।

ছোট নাম্বারের সাপেক্ষে তুলনাঃ যেসব ছোট নাম্বার নেয়া হয় সেগুলো হতে পারে 1e-8, 1e-9 ইত্যাদী যাদের মানে যথাক্রমে ১ * ১০^(-৮), ১ * ১০^(-৯) ইত্যাদী। নাম্বারটি কত ছোট হবে তা প্রয়োজনের উপর নির্ভর করে। এই নাম্বারকে এপসাইলন বলা হয়। উপরে যে কোডটি দেখানো হয়েছে সেটিই এবার এপসাইলন (EPS) এর সাহায্যে তুলনা করে দেখা যাক।

```
float EPS = 1e-5;

float value1 = 1.345;

float value2 = 1.123;

float total = value1 + value2; // the value should be 2.468

if( fabs( total - 2.468 ) <= EPS )

printf( "equal" );

else

printf( "not equal" );

Output: equal
```

এখানে শুরুতেই EPS ডিক্লেয়ার করা হল যার মান দেয়া হয়েছে 1e-5 । অর্থাৎ যদি তুলনা করার নাম্বার দুটির ব্যবধান এর সমান বা ছোট হয় তবে ধরে নেয়া হবে তারা সমান বলার মত যথেষ্ট কাছাকাছি। এরপর fabs( ) ফাংশনটি দিয়ে ফ্লোটিং পয়েন্ট নাম্বারের পরম মান রিটার্ন করা হল। এরপর কন্ডিশন এর মাধ্যমে চেক করা হল দেয়া হল যে সেটি EPS এর সমান বা ছোট কী না। যদি তা হয় তাহলে নাম্বার দুটি সমান। না হলে এরা সমান না। অতএব প্রিসিশন ক্ষুদ্র হেরফেরটুকু হিসাবে আনার জন্য অতি ক্ষুদ্র একটা এপসাইলন মান নেওয়া হয়।

একই যুক্তি ব্যবহার করে আরও বেশ কিছু উদাহরণ এর কোড দেখা যাক-

```
double a, b;

double EPS = 1e-9;

if( a + EPS < b ) // to compare a < b

if( a > b + EPS ) // to compare a > b

if( a > EPS ) // to compare a > 0

if( a < -EPS ) // to compare a < 0
```

-0.0

সাইন-ম্যান্টিসা-এক্সপোনেন্ট এই পদ্ধতিতে নাম্বার রাখার ফলে আরেকটা সমস্যা হল- ঋণাত্মক শূন্য। মেমরিতে শূন্যকে রাখার জন্য ম্যান্টিসা এবং এক্সপোনেন্টের সবগুলো বিট ০ করে দেওয়া হয়। এরপর অবশিষ্ট যে সাইন বিটটি থাকে সেটি নিয়ে অসুবিধার উদ্ভব হয়। সেটি ০ হলে নাম্বারটি +০.০ আর সেটি ১ হলে নাম্বারটি হয় -০.০! কোড এর মাধ্যমে উদাহরণ দেখা যাক-

```
double x = -0.0;

double y = +0.0;

cout << x <<”, “<<y << endl;

Output: -0, 0
```

লং ডাবল

যেহেতু মেমরিতে বিট সংখ্যা সীমাবদ্ধ থাকে তাই হিসাব সবসময় একদম নিখুঁত হয় না। স্বাভাবিকভাবেই বিট সংখ্যা বাড়লে প্রিসিশন ভালো হবে, ফলে হিসাব আরো বেশি নিখুঁত হবে। একারণে long double ব্যবহার করে প্রিসিশন বাড়ানো যেতে পারে। তবে এটা প্ল্যাটফর্ম এর উপর নির্ভর করে। যেমন g++ এ long double ডাটা টাইপটির জন্য মেমরিতে ৮০ বিট মেমরি এলোকেট হয়। অন্যদিকে MSVC++ এ long double ডাটা টাইপ থাকলেও সেটি double এ ম্যাপ করা। অর্থাৎ long double ব্যবহার করলে সেটা ৬৪ বিটের double ডাটা টাইপ হিসাবে কাজ করে।

একই এরিথমেটিক অপারেশনের ভিন্ন মাণ

অনেক সময় এরকম হতেই পারে যে একই এরিথমেটিক অপারেশন একবার একরকম ফল দিল, আরেকবার আরেকরকম দিচ্ছে। এর কারণ হল কম্পাইলার অনেকসময় প্রিসিশন ঠিক রাখার চেষ্টা করতে গিয়ে ইন্টারনালি double এর জায়গায় long double নিয়ে কাজ করতে পারে। সবশেষে যে আউটপুট দেওয়ার কথা সেটিকে double এ টাইপকাস্ট করে নিবে। কম্পাইলার যদি একসময় ঠিক করে যে সে long double নিয়ে কাজ করবে, আবার অন্য কোনসময় ঠিক করে যে double নিয়ে কাজ করবে তাহলে একই েরিথমেটিক অপারেশনের আউটপুট ভিন্ন হবে। তাই সবক্ষেত্রে long double নিয়ে কাজ করা নিরাপদ।

আরেকটি সমস্যা হতে পারে যে অনেকক্ষেত্রে কোড এর এক্সিকিউশন অপ্টিমাইজ করতে যেয়ে কম্পাইলার ইন্সট্রাকশনের সিকোয়েন্স পাল্টাতে পারে। ভিন্ন ভিন্ন ক্ষেত্রে কম্পাইলার একই কোড ভিন্ন ভিন্ন ভাবে এক্সিকিউট করতে পারে। যেমন একটি স্টেটমেন্ট যদি এমন হয় x+y-z সেটিকে কম্পাইলার (x+y)-z অথবা x+(y-z) এর যেকোনভাবে এক্সিকিউট করতে পারে।

নিচের কোডটি দেখা যাক-

```
double x = 1.0;

double y = 1e30;

double z = 1e30;

double out1 = (x+y)-z;

double out2 = x+(y-z);

cout << out1 << “, ” << out2 << endl;

Output: 0, 1
```

অতএব একই কোড একাধিক জায়গায় থাকলেও বলা সম্ভব না যে সেটি একই আউটপুট দিবে। এই সমস্যা দূর করার জন্য একটা উপায় হল যেই কোডটি একাধিক জায়গায় ব্যবহার করা হচ্ছে সেটিকে একটি ফাংশন হিসেবে লিখে সেই ফাংশনটি দুই জায়গায় কল করা। তাহলে একই আউটপুট আসবে।

রেফারেন্সঃ

i. http://floating-point-gui.de/formats/fp/
ii. http://hackaday.com/2015/10/22/an-improvement-to-floating-point-numbers/
iii. https://www.doc.ic.ac.uk/~eedwards/compsys/float/
iv. https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_2/Fundamentals_of_data_representation/Floating_point_numbers
v. http://www.cs.uwm.edu/~cs151/Bacon/Lecture/HTML/ch03s10.html