Technical Slide

Lesson: Insiduous numbers

Video: Integer types and overflow

Integer overflow

Integer overflow

```
int a = 50000;
int b = 50000;
cout << a * b;
```

Integer overflow

```
int a = 50000;
int b = 50000;
cout << a * b;
```

-1794967296

Numbers inside a computer

- Memory — a sequence of binary digits — bits
  0101111010010011100010000111101 0. . .

Numbers inside a computer

- Memory — a sequence of binary digits — bits
  0101111010010011100010000111101 0...
- A number must come in binary
  $13 = 1101_2$

Numbers inside a computer

- Memory — a sequence of binary digits — bits
  0101111010010011100010000111010...
- A number must come in binary
  $13 = 1101_2 = 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8$

Numbers inside a computer

- Memory — a sequence of binary digits — bits
  0101111010010011100010000111101 0...
- A number must come in binary
  $13 = 1101_2 = 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8$
- Bits come in chunks of 8 — bytes
  01011110 10010011 10001000 01111010 ...

Numbers inside a computer

- Memory — a sequence of binary digits — bits
  01011110100100111000100001111010...
- A number must come in binary
  $13 = 1101_2 = 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8$
- Bits come in chunks of 8 — bytes
  01011110 10010011 10001000 01111010 ...
- `int a = 13;`
  00000000 00000000 00000000 00001101

int type

- 4 bytes, or 32 bits

int type

- 4 bytes, or 32 bits
- Only 2^{32} different values!

int type

- 4 bytes, or 32 bits
- Only 2^{32} different values!
- From -2^{31} to $2^{31}-1$
  $2^{31} = 2\,147\,483\,648$
  Slightly more than 2 billion

int type

- 4 bytes, or 32 bits
- Only 2^{32} different values!
- From -2^{31} to $2^{31} - 1$
  $2^{31} = 2\,147\,483\,648$
  Slightly more than 2 billion
- $50\,000 \cdot 50\,000 = 2.5$ billion — couldn't fit!

Python

- In C++ and Java basic integers have fixed size — 4 bytes in `int`, for example

Python

- In C++ and Java basic integers have fixed size — 4 bytes in `int`, for example
- In Python they grow as needed

Python

- In C++ and Java basic integers have fixed size — 4 bytes in `int`, for example
- In Python they grow as needed
- So no overflow there
  But larger values — more space and time

long type

- 8 bytes, or 64 bits

long type

- 8 bytes, or 64 bits
- `long long` or `int64_t` in C++, `long` in Java

long type

- 8 bytes, or 64 bits
- `long long` or `int64_t` in C++, `long` in Java
- 2^{64} different values

long type

- 8 bytes, or 64 bits
- `long long` or `int64_t` in C++, `long` in Java
- 2^{64} different values
- From -2^{63} to $2^{63} - 1$
  Slightly more than $9 \cdot 10^{18}$

long type

- 8 bytes, or 64 bits
- `long long` or `int64_t` in C++, `long` in Java
- 2^{64} different values
- From -2^{63} to $2^{63}-1$
  Slightly more than $9 \cdot 10^{18}$
- $2^{31} \cdot 2^{31} = 2^{31+31} = 2^{62} < 2^{63}$
  Product of two ints always fits!

long type

- 8 bytes, or 64 bits
- `long long` or `int64_t` in C++, `long` in Java
- 2^{64} different values
- From -2^{63} to $2^{63} - 1$
  Slightly more than $9 \cdot 10^{18}$
- $2^{31} \cdot 2^{31} = 2^{31+31} = 2^{62} < 2^{63}$
  Product of two ints always fits!
- $10^9 \cdot 10^9 = 10^{18} < 9 \cdot 10^{18}$

Technical Slide

Lesson: Insiduous numbers

Video: Dealing with overflow

How to beat overflow

How to beat overflow

```
int a = 50000;
int b = 50000;
long long c = a * b;
```

How to beat overflow

```
int a = 50000;
int b = 50000;
long long c = a * b;
```

Wrong!

How to beat overflow

```
int a = 50000;
int b = 50000;
long long c = a * b;
```

```
long long c = 50000 * 50000;
```

How to beat overflow

```
int a = 50000;
int b = 50000;
long long c = a * b;

long long c = 50000 * 50000;
```

Also wrong!

Integer constants are 32 bit almost everywhere

How to beat overflow correctly

How to beat overflow correctly

- Have at least one factor of long type

```
long long a = 50000;
long long b = 50000;
long long c = a * b;
```

How to beat overflow correctly

- Have at least one factor of long type

```
long long a = 50000;
long long b = 50000;
long long c = a * b;
```

- Cast explicitly

```
int a = 50000;
int b = 50000;
long long c = (long long)a * b;
```

Keep in mind

- Long type — twice the memory, most likely slower

Keep in mind

- Long type — twice the memory, most likely slower
- Always check products for overflow using the limits from the statement

Keep in mind

- Long type — twice the memory, most likely slower
- Always check products for overflow using the limits from the statement
- Sums could lead to overflow just as easily

```
int a = 50000;
int b = 50000;
int res = 0;
for (int i = 0; i < b; ++i) {
    res += a;
}
```

Keep in mind

- Long type — twice the memory, most likely slower
- Always check products for overflow using the limits from the statement
- Sums could lead to overflow just as easily

```
int a = 50000;
int b = 50000;
int res = 0;
for (int i = 0; i < b; ++i) {
    res += a;
}
```

Wrong!

Keep in mind

- Long type — twice the memory, most likely slower
- Always check products for overflow using the limits from the statement
- Sums could lead to overflow just as easily

```
int a = 50000;
int b = 50000;
long long res = 0;
for (int i = 0; i < b; ++i) {
    res += a;
}
```

Keep in mind

- Long type — twice the memory, most likely slower
- Always check products for overflow using the limits from the statement
- Sums could lead to overflow just as easily

```
int a = 50000;
int b = 50000;
long long res = 0;
for (int i = 0; i < b; ++i) {
    res += a;
}
```

- If even 64 bits is not enough — think again

Summary

Summary

- Always check products and sums for overflow

Summary

- Always check products and sums for overflow
- Estimate magnitude using worst-case input values

Summary

- Always check products and sums for overflow
- Estimate magnitude using worst-case input values
- Use 64 bit type when needed

Technical Slide

Lesson: Insiduous numbers

Video: Non-integers

Non-integer arithmetic

Non-integer arithmetic

- Simple arithmetics:
  $a/b \cdot b = a$

Non-integer arithmetic

- Simple arithmetics:
  $a/b \cdot b = a$
- $1/49 \cdot 49 \neq 1$
  0.99999999999999988898

Non-integer arithmetic

- Simple arithmetics:
  $a/b \cdot b = a$
- $1/49 \cdot 49 \neq 1$
  0.99999999999999988898
- Close enough to one, but not *exactly* one

Rational numbers

- As fractions $\frac{A}{B}$ where A, B are integers

Rational numbers

- As fractions $\frac{A}{B}$ where A, B are integers
- Easy to store — just a pair of integers

Rational numbers

- As fractions $\frac{A}{B}$ where A, B are integers
- Easy to store — just a pair of integers
- Could do arithmetical operations:

$$\frac{A}{B} + \frac{C}{D} = \frac{A \cdot D + C \cdot B}{B \cdot D}$$
$$\frac{A}{B} \cdot \frac{C}{D} = \frac{A \cdot C}{B \cdot D}$$

Rational numbers

- As fractions $\frac{A}{B}$ where A, B are integers
- Easy to store — just a pair of integers
- Could do arithmetical operations:

$$\frac{A}{B} + \frac{C}{D} = \frac{A \cdot D + C \cdot B}{B \cdot D}$$
$$\frac{A}{B} \cdot \frac{C}{D} = \frac{A \cdot C}{B \cdot D}$$

- *Exact* value:
  different fractions $\iff$ different pairs (A, B)
  (if irreducible)
  $\frac{1}{49} \cdot 49 = \frac{49}{49} = 1$

Rational numbers

- Not only the *magnitude* is bound — $A < 2^k$, but also the *precision* — the smallest positive number we could store is $\frac{1}{2^{k-1}}$

Rational numbers

- Not only the *magnitude* is bound — $A < 2^k$, but also the *precision* — the smallest positive number we could store is $\frac{1}{2^{k-1}}$
- No way of storing arbitrarily small positive numbers — infinitely many numbers of form $\frac{1}{k}$

Rational numbers

- Not only the *magnitude* is bound — $A < 2^k$, but also the *precision* — the smallest positive number we could store is $\frac{1}{2^{k-1}}$
- No way of storing arbitrarily small positive numbers — infinitely many numbers of form $\frac{1}{k}$
- A and B have to fit in an integer type, but

$$\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{25} = \frac{34\,052\,522\,467}{8\,923\,714\,800}$$

— values grow fast even in sums!

Rational numbers

- Not only the *magnitude* is bound — $A < 2^k$, but also the *precision* — the smallest positive number we could store is $\frac{1}{2^{k-1}}$
- No way of storing arbitrarily small positive numbers — infinitely many numbers of form $\frac{1}{k}$
- A and B have to fit in an integer type, but

$$\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{25} = \frac{34\,052\,522\,467}{8\,923\,714\,800}$$

— values grow fast even in sums!
- Irrational numbers: $\sqrt{2}$, π, e

Decimal fractions

- 2.37
  0.125

Decimal fractions

- $2.37 = \frac{237}{100}$
  $0.125 = \frac{125}{1000}$
- $\dfrac{\text{whole number without the point}}{10^{\text{how many digits after the point}}}$

Decimal fractions

- $2.37 = \frac{237}{100}$
  $0.125 = \frac{125}{1000}$
- $\dfrac{\text{whole number without the point}}{10^{\text{how many digits after the point}}}$
- $\sqrt{2} = 1.41421356\ldots$
  $\frac{2}{3} = 0.66666666\ldots$

Decimal fractions

- $2.37 = \frac{237}{100}$
  $0.125 = \frac{125}{1000}$
- $\dfrac{\text{whole number without the point}}{10^{\text{how many digits after the point}}}$
- $\sqrt{2} = 1.41421356\ldots$
  $\frac{2}{3} = 0.66666666\ldots$
- $\sqrt{2} \to 1.414$
  $\frac{2}{3} \to 0.667$
  $\text{Error} \leq \frac{10^{-3}}{2}$

Binary fractions

- $1.01_2 = \frac{101_2}{4} = \frac{5}{4}$
  $0.001_2 = \frac{1_2}{8} = \frac{1}{8}$

Binary fractions

- $1.01_2 = \frac{101_2}{4} = \frac{5}{4}$
  $0.001_2 = \frac{1_2}{8} = \frac{1}{8}$
- $\dfrac{\text{whole number without the point}}{2^{\text{how many digits after the point}}}$

Binary fractions

- $1.01_2 = \frac{101_2}{4} = \frac{5}{4}$
  $0.001_2 = \frac{1_2}{8} = \frac{1}{8}$
- $\dfrac{\text{whole number without the point}}{2^{\text{how many digits after the point}}}$
- $\frac{2}{3} = 0.10101010\ldots_2$
  $\frac{2}{3} \to 0.101_2$
  Error $\leq \frac{2^{-3}}{2} = 2^{-4}$

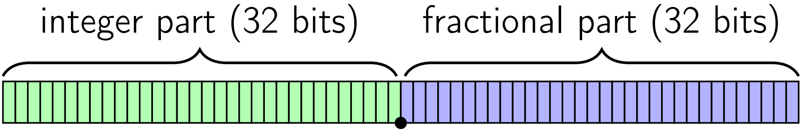
Technical Slide

Lesson: Insiduous numbers

Video: Fixed point numbers and errors

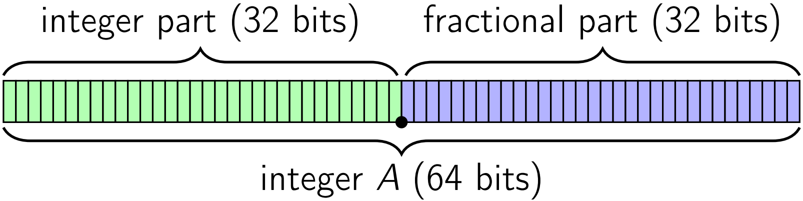
Fixed point

Idea: always keep some fixed number of digits after the point



Fixed point

Idea: always keep some fixed number of digits after the point



Actually store integer A, but think of it as $\frac{A}{2^{32}}$

Fixed point

- Maximum value:

max int	1111111...11111

$$\frac{2^{63}-1}{2^{32}} = 2^{31} - \frac{1}{2^{32}} \simeq 2000000000$$

Fixed point

- Maximum value:

max int	1111111...11111

$$\frac{2^{63}-1}{2^{32}} = 2^{31} - \frac{1}{2^{32}} \simeq 2000000000$$

- Minimum value:

min int	0000000...00000

$$\frac{-2^{63}}{2^{32}} = -2^{31} \simeq -2000000000$$

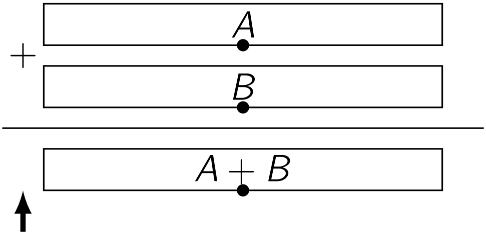
Fixed point

- We could store any real number from -2^{31} to $2^{31} - \frac{1}{2^{32}}$ with error $\leq \frac{1}{2^{33}}$

Fixed point

- We could store any real number from -2^{31} to $2^{31} - \frac{1}{2^{32}}$ with error $\leq \frac{1}{2^{33}}$
- Addition:

$$\frac{A}{2^{32}} + \frac{B}{2^{32}} = \frac{A+B}{2^{32}}$$

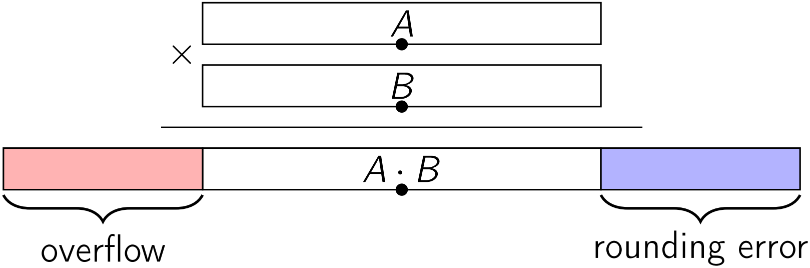


overflow is possible!

Fixed point

- Multiplication:

$$\frac{A}{2^{32}} \cdot \frac{B}{2^{32}} = \frac{A \cdot B}{2^{64}} = \frac{A \cdot B / 2^{32}}{2^{32}}$$



Absolute error

- The *absolute error* — the absolute difference between the value we have and the value we want

Absolute error

- The *absolute error* — the absolute difference between the value we have and the value we want
- Storage: some real number $a \to$ our 64-bit fixed-point representation $\widehat{a}$: $|a - \widehat{a}| \leq 2^{-33}$

Absolute error

- The *absolute error* — the absolute difference between the value we have and the value we want
- Storage: some real number $a \to$ our 64-bit fixed-point representation $\widehat{a}$: $|a - \widehat{a}| \leq 2^{-33}$
- Addition: $|(a + b) - (\widehat{a} + \widehat{b})| \leq |a - \widehat{a}| + |b - \widehat{b}|$
  $\widehat{a}$ and $\widehat{b}$ just rounded from a and b, then error $\leq 2 \cdot 2^{-33} = 2^{-32}$
  More operations — larger error

Absolute error

- Multiplication:

$$
\begin{aligned}
|a \cdot b - \widehat{a} \cdot \widehat{b}| \leq \\
& |a \cdot b - \widehat{a} \cdot b + \widehat{a} \cdot b - \widehat{a} \cdot \widehat{b}| \leq \\
& \qquad |b| \cdot |a - \widehat{a}| + |\widehat{a}| \cdot |b - \widehat{b}|
\end{aligned}
$$

Absolute error

- Multiplication:

$$
\begin{aligned}
|a \cdot b - \widehat{a} \cdot \widehat{b}| \leq \\
& |a \cdot b - \widehat{a} \cdot b + \widehat{a} \cdot b - \widehat{a} \cdot \widehat{b}| \leq \\
&& |b| \cdot |a - \widehat{a}| + |\widehat{a}| \cdot |b - \widehat{b}|
\end{aligned}
$$

- $a = \widehat{a} = 10^9$, $b = 1$, $\widehat{b} = 1 + 10^{-9}$
  $a \cdot b = 10^9$, $\widehat{a} \cdot \widehat{b} = 10^9 + 1$
  So the error's grown from 10^{-9} to 1!

Relative error

- The *relative error* — the absolute error divided by the magnitude of the exact value

$$\frac{|a - \widehat{a}|}{|a|}$$

Relative error

- The *relative error* — the absolute error divided by the magnitude of the exact value

$$\frac{|a - \widehat{a}|}{|a|}$$

- Multiplication:

$$\frac{|a \cdot b - \widehat{a} \cdot \widehat{b}|}{|a \cdot b|} \leq \frac{|b| \cdot |a - \widehat{a}| + |\widehat{a}| \cdot |b - \widehat{b}|}{|a \cdot b|}$$
$$\simeq \frac{|a - \widehat{a}|}{|a|} + \frac{|b - \widehat{b}|}{|b|}$$

- $a = \widehat{a} = 10^9$, $b = 1$, $\widehat{b} = 1 + 10^{-9}$
  $a \cdot b = 10^9$, $\widehat{a} \cdot \widehat{b} = 10^9 + 1$
  Relative error
  $$\frac{|10^9 - (10^9 + 1)|}{10^9} = 10^{-9}$$

- $a = \widehat{a} = 10^9$, $b = 1$, $\widehat{b} = 1 + 10^{-9}$
  $a \cdot b = 10^9$, $\widehat{a} \cdot \widehat{b} = 10^9 + 1$
  Relative error
  $$\frac{|10^9 - (10^9 + 1)|}{10^9} = 10^{-9}$$
- Addition: $a = \widehat{a} = 10^9$, $b = -10^9 + 1$, $\widehat{b} = -10^9$
  $$\frac{|(a + b) - (\widehat{a} + \widehat{b})|}{|a + b|} = \frac{1}{1} = 1,$$
  from
  $$\frac{|a - \widehat{a}|}{|a|} = 0, \quad \frac{|b - \widehat{b}|}{|b|} \simeq 10^{-9}$$

Fixed point

- Fixed point behaves well with the absolute error

Fixed point

- Fixed point behaves well with the absolute error
- But the relative error depends on the magnitude!

Fixed point

- Fixed point behaves well with the absolute error
- But the relative error depends on the magnitude!
- $a \simeq 2^{31}$ — 64 correct binary digits

Fixed point

- Fixed point behaves well with the absolute error
- But the relative error depends on the magnitude!
- $a \simeq 2^{31}$ — 64 correct binary digits
- $a \simeq 2^{-32}$ — only one correct binary digit!

Fixed point

- Fixed point behaves well with the absolute error
- But the relative error depends on the magnitude!
- $a \simeq 2^{31}$ — 64 correct binary digits
- $a \simeq 2^{-32}$ — only one correct binary digit!
- On "average" number $a \simeq 1$ first half of digits is not used
  We can do better!

Technical Slide

Lesson: Insiduous numbers

Video: Floating point numbers

Floating point

- Idea: Each number has its own most important digits

$$\ldots 000\textcolor{red}{1}\textcolor{green}{01001}\textcolor{red}{.}\textcolor{green}{010}0110\ldots$$

Floating point

- Idea: Each number has its own most important digits

$$\ldots 000101001.0100110\ldots$$

- The space is limited
  So it's natural to store some fixed number of *first* digits

$$1.01001010 \cdot 2^5$$

  and the distance between the first one and the point — to know the actual position of the point

$$01001010 \quad 0101$$

- We could store any fraction between 1.00000000 and 1.11111111 with any shift from -8 to 7

- We could store any fraction between 1.00000000 and 1.11111111 with any shift from -8 to 7
- Maximum value

$$1.11111111 \cdot 2^{7}$$

$$11111111.1 = 255.5$$

- We could store any fraction between 1.00000000 and 1.11111111 with any shift from -8 to 7
- Maximum value

$$1.11111111 \cdot 2^{7}$$

$$11111111.1 = 255.5$$

- Minimum positive value

$$1.00000000 \cdot 2^{-8} = \frac{1}{256}$$

- We could store any fraction between 1.00000000 and 1.11111111 with any shift from -8 to 7
- Maximum value

$$1.11111111 \cdot 2^{7}$$

$$11111111.1 = 255.5$$

- Minimum positive value

$$1.00000000 \cdot 2^{-8} = \frac{1}{256}$$

- For any number, we round to first 9 digits, so the relative error $\leq 2^{-9}$

Floating point addition

$$1.01110101 \cdot 2^{3} + 1.10010110 \cdot 2^{-1}$$

Floating point addition

$$1.01110101 \cdot 2^{3} + 1.10010110 \cdot 2^{-1}$$

$$\begin{array}{rr} & 1011.10101 \\ + & 0.110010110 \\ \hline & 1100.011100110 \end{array}$$

$$1.10001110 \cdot 2^{3}$$

Tail of the smaller gets rounded!

Floating point multiplication

$$\begin{aligned} 1.01110101 \cdot 2^{3} \times 1.10010110 \cdot 2^{-1} &= \\ 1.01110101 \times 1.10010110 \cdot 2^{3+(-1)} &= \\ 1.100100111110001110 \cdot 2^{2} & \end{aligned}$$

Floating point multiplication

$$1.01110101 \cdot 2^{3} \times 1.10010110 \cdot 2^{-1} =$$

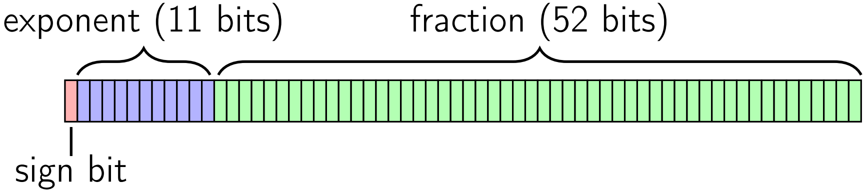
$$1.01110101 \times 1.10010110 \cdot 2^{3+(-1)} =$$

$$1.100100111110001110 \cdot 2^{2}$$

$$1.10010100 \cdot 2^{2}$$

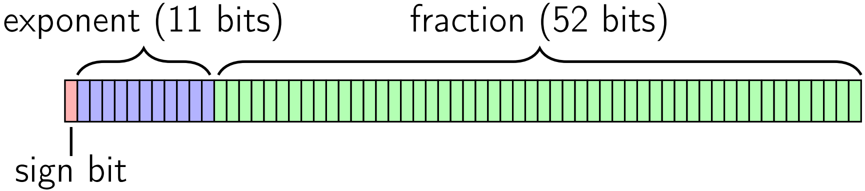
The product has more digits, need to round!

Double type



$$(-1)^s (1.f_0 f_1 \dots f_{51})_2 \cdot 2^e$$

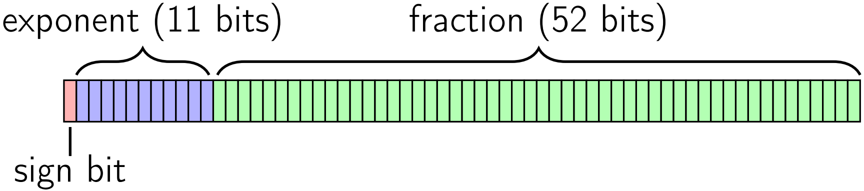
Double type



$$(-1)^s(1.f_0f_1\ldots f_{51})_2 \cdot 2^e$$

- Maximum value $2^{2^{10}} \simeq 10^{309}$

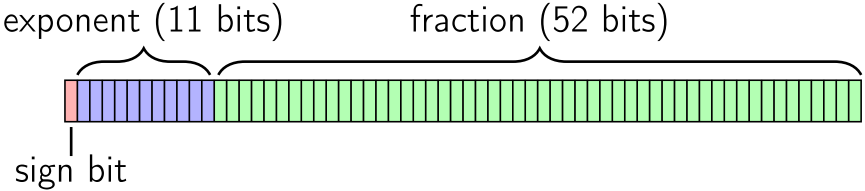
Double type



$$(-1)^s(1.f_0 f_1 \dots f_{51})_2 \cdot 2^e$$

- Maximum value $2^{2^{10}} \simeq 10^{309}$
- Minimum positive value $2^{-2^{10}} \simeq 10^{-309}$

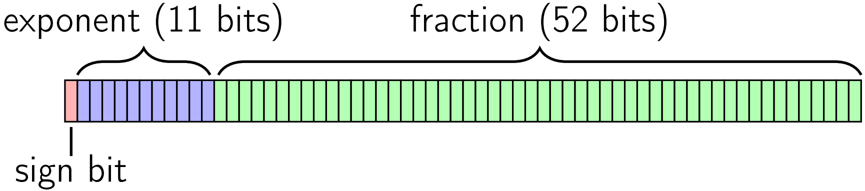
Double type



$$(-1)^s(1.f_0f_1\ldots f_{51})_2\cdot 2^e$$

- Maximum value $2^{2^{10}} \simeq 10^{309}$
- Minimum positive value $2^{-2^{10}} \simeq 10^{-309}$
- Huge range of magnitude for 11 bits
  Would be 2 kilobytes for fixed point

Double type



$$(-1)^s (1.f_0 f_1 \ldots f_{51})_2 \cdot 2^e$$

- Maximum value $2^{2^{10}} \simeq 10^{309}$
- Minimum positive value $2^{-2^{10}} \simeq 10^{-309}$
- Huge range of magnitude for 11 bits
  Would be 2 kilobytes for fixed point
- Each number with 53 accurate binary digits,
  about 16 decimal digits

Technical Slide

Lesson: Insiduous numbers

Video: Where and how to use doubles

Double vs 64 bit integer

Double vs 64 bit integer

- Less actual digits: 53 vs 64 — no exponents in integers

Double vs 64 bit integer

- Less actual digits: 53 vs 64 — no exponents in integers
- More time on computations — doubles could be 1.5-2 times slower

Double vs 64 bit integer

- Less actual digits: 53 vs 64 — no exponents in integers
- More time on computations — doubles could be 1.5-2 times slower
- No overflow in doubles
  Fractional part is always first digits
  Possible overflow in exponent, but only at 10^{300}

Double vs 64 bit integer

- Less actual digits: 53 vs 64 — no exponents in integers
- More time on computations — doubles could be 1.5-2 times slower
- No overflow in doubles
  Fractional part is always first digits
  Possible overflow in exponent, but only at 10^{300}
- Errors everywhere
  `1.0 / 49`, `sqrt(2)`
  `>>> 0.1 + 0.2`
  `0.30000000000000004`
  and they grow!

Integers wherever possible

Integers wherever possible

- Rational numbers $\frac{9}{13}$

Integers wherever possible

- Rational numbers $\frac{9}{13}$
- Decimal fractions \$2.49 = 249¢

Integers wherever possible

- Rational numbers $\frac{9}{13}$
- Decimal fractions \$2.49 = 249¢
- Roots:
  `while i < sqrt(n)` $\rightarrow$ `while i * i < n`

Integers wherever possible

- Rational numbers $\frac{9}{13}$
- Decimal fractions $\$2.49 = 249¢$
- Roots:
  `while i < sqrt(n)` $\rightarrow$ `while i * i < n`
- Comparing lengths:
  $|(x, y)| = \sqrt{x^2 + y^2}$
  $\sqrt{a} < \sqrt{b} \iff a < b$

Doubles are needed

- Most common case: floating point in output "Output ... with absolute or relative error no more than 10^{-6}."

Doubles are needed

- Most common case: floating point in output "Output ... with absolute or relative error no more than 10^{-6}."
- Print answer with some fixed large number of digits after the point

```
cout << fixed << setprecision(20) << ans;
System.out.format("%.20f", ans);
print("%.20f" % ans)
```

Integers as long as possible

Integers as long as possible

- $\frac{11}{7} + \frac{1}{2} + \frac{5}{14} = \frac{34}{14}$
  five floating point operations vs **one**

Integers as long as possible

- $\frac{11}{7} + \frac{1}{2} + \frac{5}{14} = \frac{34}{14}$
  five floating point operations vs **one**
- $1/2/3/5 = 1/(2 \cdot 3 \cdot 5)$
  three floating point operations vs **one**

Integers as long as possible

- $\frac{11}{7} + \frac{1}{2} + \frac{5}{14} = \frac{34}{14}$
  five floating point operations vs **one**
- $1/2/3/5 = 1/(2 \cdot 3 \cdot 5)$
  three floating point operations vs **one**
- $\sqrt{5} \cdot 2 \cdot 3 = \sqrt{2^2 \cdot 3^2 \cdot 5}$
  three floating point operations vs **one**

Integers as long as possible

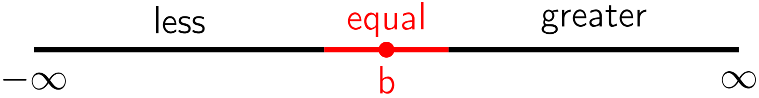
- $\frac{11}{7} + \frac{1}{2} + \frac{5}{14} = \frac{34}{14}$
  five floating point operations vs **one**
- $1/2/3/5 = 1/(2 \cdot 3 \cdot 5)$
  three floating point operations vs **one**
- $\sqrt{5} \cdot 2 \cdot 3 = \sqrt{2^2 \cdot 3^2 \cdot 5}$
  three floating point operations vs **one**
- But watch for integer overflow!

How to live with errors

How to live with errors

- Consider values of small difference equal

statement	integers	doubles
a is equal to b	`a == b`	`abs(a - b) < eps`
a is *strictly* less than b	`a < b`	`a < b - eps`
a is less than or equal to b	`a <= b`	`a < b + eps`



How to live with errors

- If strictness is not important, usual `a < b` is still better (e.g. while sorting)

How to live with errors

- If strictness is not important, usual `a < b` is still better (e.g. while sorting)
- Truncate carefully with `floor` and `ceil`: instead of `floor(a)`, better `floor(a + eps)` or else if a should be 1, but has an error of 10^{-9}, `floor(0.999999999)` is zero

How to choose eps

- It could be proven, that certain value of eps is enough
  Knowing how errors grow while
  - storing
  - summing
  - multiplying
  - ...

  numbers, you could bound the difference from the exact value and use it as eps

How to choose eps

- It could be proven, that certain value of eps is enough
  Knowing how errors grow while
  - storing
  - summing
  - multiplying
  - ...

  numbers, you could bound the difference from the exact value and use it as eps
- But this is rarely done on contests

How to choose eps

- It could be proven, that certain value of eps is enough
  Knowing how errors grow while
  - storing
  - summing
  - multiplying
  - ...

  numbers, you could bound the difference from the exact value and use it as eps
- But this is rarely done on contests
- Usually it's enough to take some feasible value like 1e-8 or 1e-9

How to choose eps

- What if your `eps` doesn't work?

How to choose eps

- What if your `eps` doesn't work?
- By usual means of debugging, find where errors appear firstly

How to choose eps

- What if your `eps` doesn't work?
- By usual means of debugging, find where errors appear firstly
- If indeed `eps` is guilty then either

How to choose eps

- What if your eps doesn't work?
- By usual means of debugging, find where errors appear firstly
- If indeed eps is guilty then either
- eps is too big, and unequal values are treated as equal

  then you should decrease eps — take the next power of 10, e.g. $10^{-8} \to 10^{-9}$

How to choose eps

- What if your eps doesn't work?
- By usual means of debugging, find where errors appear firstly
- If indeed eps is guilty then either
- eps is too big, and unequal values are treated as equal
  then you should decrease eps — take the next power of 10, e.g. $10^{-8} \to 10^{-9}$
- or eps is too small, and equal values are treated as unequal
  then you should increase eps — e.g. $10^{-8} \to 10^{-7}$

Technical Slide

Lesson: Insiduous numbers

Video: More on floating point

Order of computations

Order of computations

- ```
  >>> (1e18 + 1) - 1e18
  0.0
  ```
  Values of different magnitude sum up with large errors! Try to avoid that
  ```

Order of computations

- `>>> (1e18 + 1) - 1e18`
  `0.0`
  Values of different magnitude sum up with large errors! Try to avoid that
- $x^2 - y^2 = (x - y) \cdot (x + y)$

Order of computations

- ```
  >>> (1e18 + 1) - 1e18
  0.0
  ```
  Values of different magnitude sum up with large errors! Try to avoid that
- $x^2 - y^2 = (x - y) \cdot (x + y)$
  ```
  >>> y = 1e9
  >>> x = y + 1
  >>> x**2 - y**2
  2000000000.0
  >>> (x - y) * (x + y)
  2000000001.0
  ```
  ```

Order of computations

- ```
  >>> (1e18 + 1) - 1e18
  0.0
  ```
  Values of different magnitude sum up with large errors! Try to avoid that
- $x^2 - y^2 = (x - y) \cdot (x + y)$
  ```
  >>> y = 1e9
  >>> x = y + 1
  >>> x**2 - y**2
  2000000000.0
  >>> (x - y) * (x + y)
  2000000001.0
  ```
  $(10^{18} + 2 \cdot 10^9 + 1) - 10^{18}$
  $(10^9 + 1 - 10^9) \cdot (10^9 + 1 + 10^9) = 1 \cdot (2 \cdot 10^9 + 1)$
  ```

Other types

Other types

- Single precision `float` — 32 bit analogue of `double`
  Do not use!

Other types

- Single precision `float` — 32 bit analogue of `double`
  Do not use!
- C++: `long double` — 80 or 64 bit, depending on the compiler

Other types

- Single precision `float` — 32 bit analogue of `double`
  Do not use!
- C++: `long double` — 80 or 64 bit, depending on the compiler
  `cout << numeric_limits<long double>::digits;`
  64 or 53

Other types

- Single precision `float` — 32 bit analogue of `double`
  Do not use!
- C++: `long double` — 80 or 64 bit, depending on the compiler
  `cout << numeric_limits<long double>::digits;`
  64 or 53
- Java/Python: `BigDecimal`/`decimal` — as many leading digits as needed
  but costs space and time

Special values of double

- `cout << 1.0 / 0;`
  `inf`
  Positive infinity

Special values of double

- `cout << 1.0 / 0;`
  `inf`
  Positive infinity
- `cout << 1.0 / 0 - 1.0 / 0 << '\n';`
  `-nan`
  Not a number

Special values of double

- `cout << 1.0 / 0;`
  `inf`
  Positive infinity
- `cout << 1.0 / 0 - 1.0 / 0 << '\n';`
  `-nan`
  Not a number
- You want to avoid `inf` and `nan` in output
  `cout << sqrt(-1e-9);`
  `-nan`

Special values of double

- `cout << 1.0 / 0;`
  `inf`
  Positive infinity
- `cout << 1.0 / 0 - 1.0 / 0 << '\n';`
  `-nan`
  Not a number
- You want to avoid `inf` and `nan` in output
  `cout << sqrt(-1e-9);`
  `-nan`
  `sqrt(x)` — could lead to nan
  `sqrt(max(x, 0))` — good

Summary

- Doubles always come with errors

Summary

- Doubles always come with errors
- Use integers where possible

Summary

- Doubles always come with errors
- Use integers where possible
- Always compare doubles with `eps`
  Never use ==

Summary

- Doubles always come with errors
- Use integers where possible
- Always compare doubles with `eps`
  Never use ==
- Reorder computations — try not to add values of different magnitude

Summary

- Doubles always come with errors
- Use integers where possible
- Always compare doubles with `eps`
  Never use ==
- Reorder computations — try not to add values of different magnitude
- Watch out for `inf` and `nan`