50000 * 50000 → ??? negative

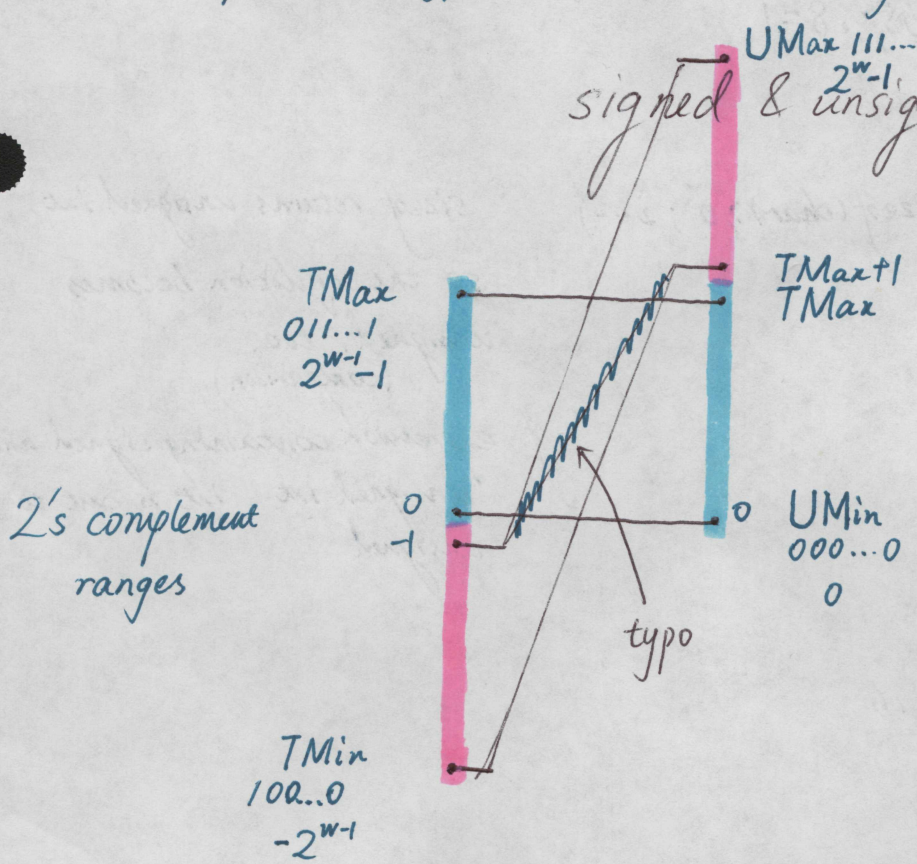
300 * 400 * 500 * 600 → ??? 1640261632 overflow

fact: use finite representation to represent things that are infinite potentially

so there are compromises: overflow, round-off problems

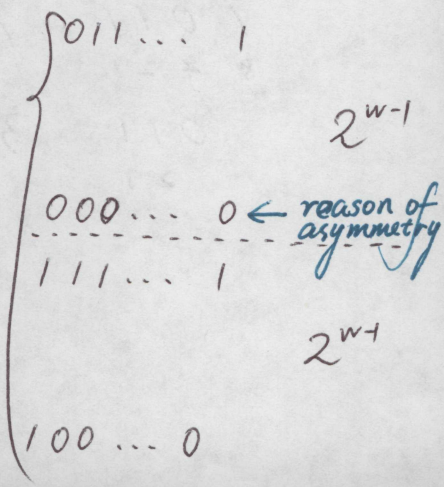
Java primitive types: byte short int long float double boolean char

C data types: short int long float double char



$|TMax| = |TMin| - 1$

$UMax = 2 * TMax + 1$



```
if (x<0)
    return -x;
else
    return x;
```

give TMin as input. then 100...0 → 100...0
which is -2147483648

not(x) +1 "complement & increment"
x

bit pattern is maintained,
but reinterpreted

```
for (unsigned int i=n-1; i>=0; i--)
    f(a[i])
```

loop will go forever, will not stop as you want
an unsigned int is never negative

```
int x=4
printf("%d", x>>65)
```

↑ 65%8=1

the compiler warns you
but will print 2

```
for (int i=n-1; i-sizeof(char)>=0; i--)
    f(a[i])
```

sizeof returns unsigned int,
so the condition becomes unsigned, too.
(comparison)

if expression containing signed and unsigned int, int is cast to unsigned

unsigned truncate

```
1 1 0 1 1      27
16 8 4 2 1
  1 0 1 1      11
  8 4 2 1
    0 1 1      3
      2 1
```

mod 16

mod 8

unsigned addition W=4 (to get the intuition) 0...15

```
 1 1 0 1   13
 0 1 0 1    5
-----------
X0 0 1 0   18 → 2
```

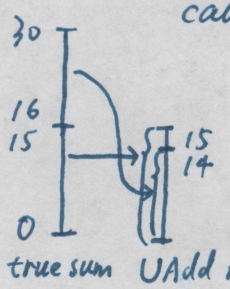
overflow, differs by 16

4 bits can't represent 18

just discard without warnings messages, errors

W bits can represent up to 2^W-1 for unsigned int, so the sum of two ints would be up to $(2^W-1)*2 = 2^{W+1}-2$, for sum larger than or equal to 2^W, we subtract it by 2^W. e.g. if W=4

called overflow



signed addition W=4 -8...7

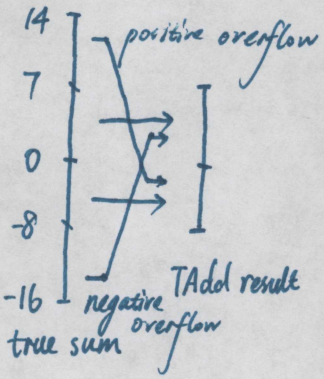
```
 1 1 0 1  -3
 0 1 0 1   5
-----------
X0 0 1 0   2
```

discard carry

OK

```
 1 0 1 1  -5
 0 0 1 1   3
-----------
 1 1 1 0  -2
```

OK




```
 1 1 0 1  -3
 1 0 1 0  -6
-----------
X0 1 1 1  +7
          -9→
```

negative overflow differs by 16

4 bits can't represent -9

```
 0 1 1 1   7
 0 1 0 1   5
-----------
 1 1 0 0  -4
          12→
```

positive overflow differs by 16

signed bit represents -8, not +8

unsigned multiplication

0 0 1 1 3
× 0 1 0 1 5
1 1 1 1 15 OK

0 1 0 1 5
× 0 1 0 1 5
0 0 0 1 | 0 0 1 25 → 9 = 25 % 16

signed multiplication

5*4
0 0 0 1 | 0 1 0 0 4
20 →

5*5
0 0 0 1 | 1 0 0 1 -7
25 →

1 1 0 1 -3
× 1 1 1 0 -2
0 1 1 0 6 OK

trick: -3 → 13u
-2 → 14u
13 × 14 = 182 . 66

Power-of-2 Multiply with Shift for both signed & unsigned

$$x = \sum_{i=0}^{w-1} 2^i$$

$$x \cdot 2^k = 2^k \cdot \sum_{i=0}^{w-1} 2^i = \sum_{i=0}^{w-1} 2^{i+k}$$

$w \rightarrow w+k \longrightarrow$ discard k bits

unsigned Power-of-2 Divide with Shift

0 1 1 0 6
>> 1 logical shift
0 0 1 1 3
>> 1 1.5 → 1
0 0 0 1 1

signed Power-of-2 Divide

1 0 1 0 -6
>> 1 arithmetic shift
1 1 0 1 -3
>> 1
1 1 1 0 -2

so we should add a bias
1 1 0 1
1
1 1 1 0
>> 1
1 1 1 1 -1

↑ compiler doing this

Why should we use unsigned? Java banished unsigned numbers and only use 2's complement to represent numbers

Should do explicit cast as much as possible

Rule of Thumb

$2^{10} = 1024 \sim 10^3$

$2^{20} \sim 10^6$

$2^{30} \sim 10^9$

$2^{40} \sim 10^{12}$

$2^{47} \sim 128\,TB$ ($TB = 10^{12}$) approximation

from programmer's perspective, memory is just a big array of bytes → virtual memory

for 64 bit machine, an address is represented in 64 bits. But the maximum address you're allowed to use is 47 bits. When you run a program, the operating system only allows certain region of memory to be referenced within the possible range. Access other regions will create segmentation fault error. Idea of big array of bytes is taken care by the operating system and computer hardware, invisible to programmers, even machine-level.

gcc { -m32 32bit code; -m64 64 bit code }

the hardware itself doesn't necessarily define what the word size is, it's the combination of hardware and complier to determine what the word size to use in the program.

most significant ← least significant

两个数字：0X12345678, 0X11223344

| 0X78 | 0X56 | 0X34 | 0X12 | 0X44 | 0X33 | 0X22 | 0X11 |

→ 内存地址增长方向

| 0X12 | 0X34 | 0X56 | 0X78 | 0X11 | 0X22 | 0X33 | 0X44 |

little endian x86, ARM processors

least significant byte has lowest address

变量指针转换时地址保持不变

big endian Sun, PPC Mac, Internet (TCP/IP)

least significant byte has highest address

内存顺序和书写顺序一致

for negative numbers so sign bit first in big endian machine

C puzzles

if $x < 0$, then $x*2$ may not be negative

example : $2 * TMin = 0$

casting it to unsigned $ux > 0$

if $x \& 7 = 7$ (last 3 bits are all 1s)

then $x << 30$ 00...0111

11...000 → negative

if $x > y$ $-x < -y$: false

negative TMin is still TMin

100...0

~011...1

+1 100...0

$x * x >= 0$: false

if $x > 0 \&\& y > 0$ $x + y > 0$: false

if $x >= 0$ $-x <= 0$: true

if $x <= 0$ $-x >= 0$: false TMin. Any positive number can be represented as negative, but TMin as a negative number cannot be represented as positive

$(x | -x) >> 31 == -1$: true for $x \neq 0$

$ux >> 3 == ux / 8$: true

$x >> 3 == x / 8$: false -1 right shift 3 bits is still -1

$x \& (x-1) != 0$: false TMin - 1 = TMax so TMin & (TMin - 1) == 0

get rid of the rightmost 1 in x

$ux > -1$: false signed integer compared with unsigned integer, promoted to unsigned

(unsigned int) -1 = UMax