作业一

本次作业一共有3道大题：

- 第1题和第2题是整数与浮点数计算，考察对知识点的掌握；
- 第3题是实践题，通过与实际编程结合，考察对知识点的理解与应用。

Q1. 整数计算

Q1.1. 二进制表示

在所有由 4个 `1` 和 4个 `0` 组成的 8位二进制整数（补码）中，最小的带符号数和最大的带符号数分别是多少？答案请以十进制表示。

- 最小的带符号数：_____
- 最大的带符号数：_____

Q1.2. 补码计算

X、Y的数据位宽均为16位。已知$[X]_{补码} = 0\text{x}0033$，$[Y]_{补码} = 0\text{x}\text{DE5A}$，则$[X-Y]_{补码}$和$[X+Y]_{补码}$的值，计算结果用十六进制补码表示。

- $[X-Y]_{补码} =$ _____
- $[X+Y]_{补码} =$ _____

Q2. 浮点数计算

Q2.1. 定点数与浮点数哪个多?

给定相同的字长（例如32位），能表示的定点数和浮点数哪个更多？请给出你的理由。

Q2.2. 9位浮点数表示

假设存在一种符合IEEE浮点数标准的9位浮点数，由1个符号位、4位阶码、4位尾数组成，数值表示仍遵循 $V = (-1)^s \cdot M \cdot 2^E$。请在下表中填空：

描述	9位二进制表示	M（十进制表示）	E（十进制表示）	V（十进制表示）
3.5				3.5
大于0的最小浮点数				

Q3. 实践题

Q3.1. 溢出检查

带符号数 `si1` 和 `si2` 符号位相同，先检查二者相加后是否会产生加法溢出：若溢出返回 `true` ，未溢出返回 `false` ，并将结果写进 `*sum` 中。

```
bool checkAddOF(int si1, int si2, int* sum) {
    unsigned usum = (unsigned)(si1) + si2;
    const unsigned MY_INT_MIN = __表达式(1)__; // 用位运算实现
    if ((__表达式(2)__) & MY_INT_MIN)
        return true;
    else {
        *sum = si1 + si2;
        return false;
    }
}
```

你需要填写 `__表达式(1)__` 和 `__表达式(2)__` （其中 `__表达式(1)__` 要求使用位运算实现，不能调标准库中 `INT_MIN` 的实现），并解释它的工作原理。

表达式	具体的表达式（按C语言风格）
`__表达式(1)__`	
`__表达式(2)__`	

（同号）加法溢出是指 _____

为了完成上述判断，在 `if()` 中通过与 `MY_INT_MIN` 进行按位与来 _____，若值非零则表示发生了溢出；若为零表示无溢出，正常赋值为 `*sum` 。

Q3.2. 字节序

在网络传输数据时，发送方需要将本地的数据通过转换为网络字节序后再发送；接收方接收后，需要转换为本地字节序后才能使用。这个转换可以调用 `htonl()` 和 `ntohl()` 方法来完成。

> **htonl() 和 ntohl()**
>
> hton 表示由 本地 (**h**ost) 字节序转为 网络 (**n**etwork) 字节序；后缀 `l` 表示 `long` ，即32位整数（对应的 `s` 表示 `short` ，即16位整数）。

```
// byte_order.c
#include <arpa/inet.h>
#include <stdio.h>

void printMemory(void *p, int size) {
    unsigned char *p1 = (unsigned char *)p;
    for (int i = 0; i < size; i++) {
        printf("%02x ", p1[i]);
    }
    printf("\n");
}

int main(int argc, char const *argv[]) {
    int a = 0x12345678;
    printf("Original: "); printMemory(&a, sizeof(a));

    a = htonl(a); // <1>
    printf("Try htonl(): "); printMemory(&a, sizeof(a));

    a = ntohl(a); // <2>
    printf("Then ntohl(): "); printMemory(&a, sizeof(a));
    return 0;
}
```

上述示例代码的编译与执行结果如下所示：

```
$ gcc byte_order.c -o byte_order && ./byte_order
Original: 78 56 34 12
Try htonl(): 12 34 56 78
Then ntohl(): 78 56 34 12
```

> 请确保你已安装实验导引在本地配置好Linux环境并测试过 `gcc` 。

Q3.3.1. 本地序 vs 网络序

根据上述执行结果，请问本地的字节序属于哪一种字节序？网络字节序又属于哪一种？

Q3.3.2. 大小端序转换

试补齐下述宏定义中缺失的数值，使得 `htonl()` 能够正确工作，并解释该宏定义的功能。

```
// glibc/bits/byteswap.h

/* Swap bytes in 32-bit value. */
#define __bswap_constant_32(x)
    ((((x) & __空格(1)__) >> 24) | (((x) & __空格(2)__) >> __空格(3)__) | \
     (((x) & __空格(4)__) <<  8) | (((x) & 0x000000ffu) << __空格(5)__))
```

补充：在C语言中， `0x000000ffu` 的意思是将 `0x000000ff` 转换为无符号整数。注意观察，位运算的格式为 `((x) & __无符号十六进制数__) >> __十进制常值__` ，按照这个格式补齐 `__空格(n)__` 的值。

空格	数值（按上述格式填写十六进制或十进制数）
`__空格(1)__`	
`__空格(2)__`	
`__空格(3)__`	
`__空格(4)__`	
`__空格(5)__`	

该宏定义完成的功能为 _____

Q3.3.3. 思考题

在 `byte_order.c` 中，如果对换 `a = htonl(a);` （注释 `<1>` 处）和 `a = ntohl(a);` （注释 `<2>` 处），程序的输出结果会有变化吗？请从 `htonl` 与 `ntohl` 的实现来解释原因。