# Homework #1

Due date: 3/22

## Binary quicksort (Radix-exchange sort)

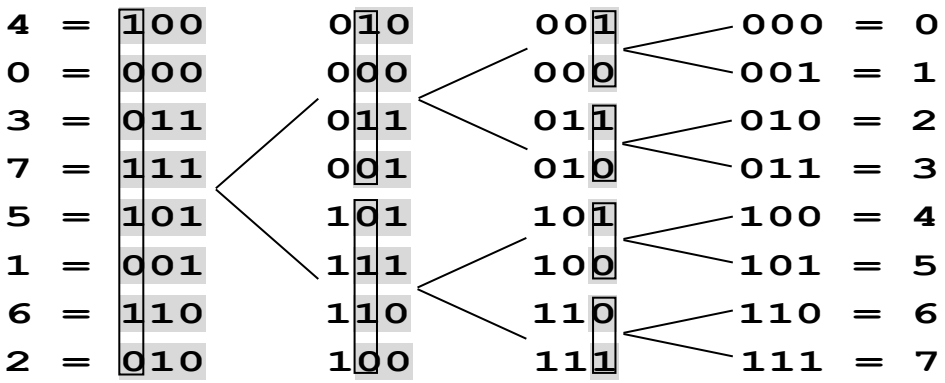
This homework generalizes Problem 2008-16 (Problem 16 of 2008 question base) of Collegiate Programming Exam. Problem 2008-16 considers only unsigned integers, whereas this homework extends it to signed integers and floating-point numbers (of `float` type only), too.

Binary quicksort (or radix-exchange sort) is a variant of quicksort that sorts a sequence of numbers bit-by-bit. Basically, it proceeds as follows:

Step 1: Partition the array into two subarrays according to the leftmost bits of the numbers so that one subarray contains numbers beginning with a 0 bit, and the other contains numbers beginning with a 1 bit.
Step 2: Recursively sort the two subarrays of numbers with 1 fewer bit (since the leftmost bits have already been sorted in Step 1).

### Sorting unsigned integers

As an example, the process of sorting the sequence 4, 0, 3, 7, 5, 1, 6, 2 of 3-bit unsigned integers is illustrated below. The bits boxed together are the leftmost bits of the shaded integers of the subarray to be sorted.



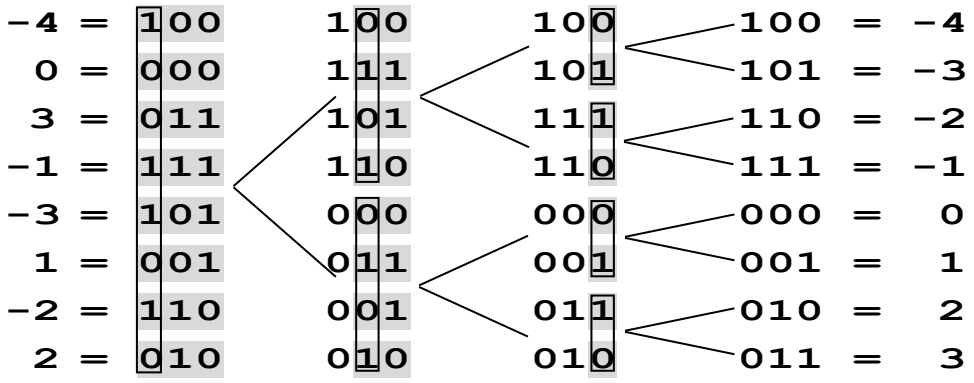

The first stage partitions the array into a subarray with integers having leading 0 bits and a subarray with integers having leading 1 bits. The first subarray is then partitioned into a subarray with integers having leading 00 bits and a

subarray with integers having leading 01 bits. The process stops when the bits are exhausted.
Note: The order of the integers in a subarray is immaterial.

## Sorting signed integers

For unsigned integers,
numbers beginning with a 0 bit $<$ numbers beginning with a 1 bit
for all bits.
However, for signed integers, there are two cases:
Case 1: Sign bit
numbers beginning with a 1 bit $<$ numbers beginning with a 0 bit
Case 2: All the other bits
numbers beginning with a 0 bit $<$ numbers beginning with a 1 bit

For example,the process of sorting the sequence -4, 0, 3, -1, -3, 1, -2, 2 of 3-bit signed integers is illustrated below.




## Sorting floating-point numbers

For floating-point numbers, we have
Case 1: Sign bit
numbers beginning with a 1 bit $<$ numbers beginning with a 0 bit
Case 2: All the other bits
2.1 for numbers $< 0$
numbers beginning with a 1 bit $<$ numbers beginning with a 0 bit
2.2 for numbers $\geq 0$
numbers beginning with a 0 bit $<$ numbers beginning with a 1 bit

For illustration purpose, suppose that $xyz$ is a 3-bit floating-point number that represents the binary number

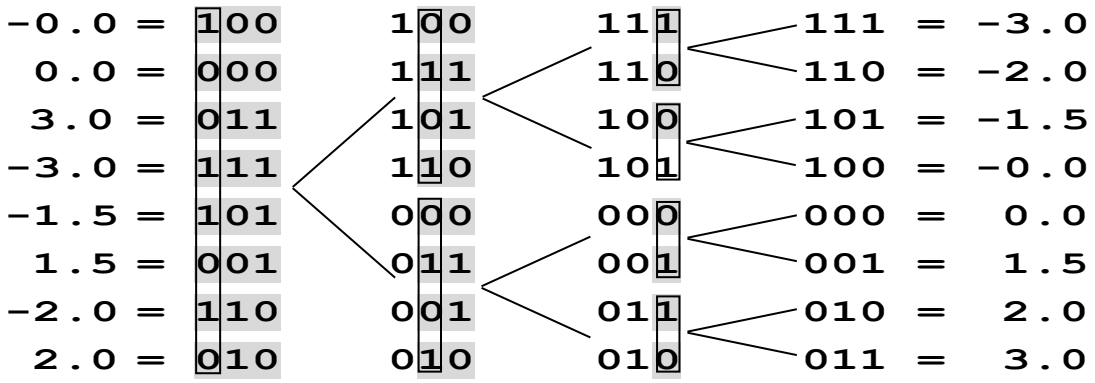
$(-1)^x \times 1.z \times 2^y$

That is, $x$ is the sign bit, $y$ is the exponent, and $z$ is the fraction – similar to IEEE 754 representation of floating-point numbers.

For example,

$111_2 = ((-1)^1 \times 1.1 \times 2^1)_2 = -3.0_{10}$

Now, consider sorting the sequence -0.0, 0.0, 3.0, -3.0, -1.5, 1.5, -2.0, 2.0 of 3-bit floating-point numbers:




In this homework, you are asked to write a generic function template that implements binary quicksort on integral types and an explicit specialization for **`float`** type.

// generic function template for integral types

```
template<typename T>
void binary_qsort(T*,int);
```

Hint

Use **`numeric_limits<T>::is_signed`** to determine if **`T`** is a signed or unsigned integral type.

// explicit specialization for **`float`**

```
template<>
void binary_qsort(float* a,int);
```

Hint

Since bitwise operations don't apply to floating-point numbers, we have to **reinterpret** the floating-point numbers as integral numbers within this specialization, say

```
int* x=reinterpret_cast<int*>(a);
```
and then binary-quicksort the array **x** with the understanding that it actually contains floating-point numbers.
Note: Here we tacitly assume that **float** and **int** have the same size.

It is up to you to decide how to implement the generic function template and the specialization. Nonetheless, it is suggested that the following two function templates be employed.

```
template<typename T>
int partition(T* a,int l,int h,int b,int pivot);
```
Hints
1 The value of **pivot** is either 0 or 1.
2 This function uses **pivot** to partition the array **a[l..h]**, according to the value of bit **b**, into two subarrays **a[l..m]** and **a[m+1..h]** so that
the value of bit **b** of each number in **a[l..m]** = **pivot**,
and
the value of bit **b** of each number in **a[m+1..h]** ≠ **pivot** (= **1-pivot**)
Comment
In case numbers beginning with a 0 bit < numbers beginning with a 1 bit, set **pivot** = 0.
In case numbers beginning with a 1 bit < numbers beginning with a 0 bit, set **pivot** = 1.
3 The value of **m** is returned as the function value.

```
template<typename T>
void binary_qsort(T* a,int l,int h,int b,int pivot);
```
Hints
1 This function sorts bit **b** to bit 0, in that order, of all numbers in the array **a[l..h]**, using **pivot** to guide the partition.
2 For example, to sort
```
unsigned a[10];
```
we may invoke
```
binary_qsort(a,0,9,8*sizeof(unsigned)-1,0);
```
Assuming that an unsigned integer occupies 4 bytes, this call sorts bit 31, bit 30, …, bit 0, in that order, of all numbers in the array. In this case, the pivot is 0, because, for all bits of unsigned integers, numbers beginning with a 0 bit < numbers beginning with a 1 bit.

### Final remarks

1 Why do we need the specialization for **`float`**? Put differently, why cannot we simply write the generic function template as

```
template<typename T>
void binary_qsort(T* a,int n)
{
    if (numeric_limits<T>::is_integer)
    // sort signed and unsigned integral numbers here
    else
    // sort float numbers here
    // (assume that we consider only  float type)
}
```

2 This homework doesn't complete the implementation of binary quicksort for lack of the specializations for **`double`** and **`long double`**. This is left as an exercise for you. (Don't turn it in, of course.)

## Sample test

Suffice it to run the sample test given in file hw1test.cpp

## Sample output

```
7 77 777 7777777 88888888 999999999 2147483646 2147483647 2147483648 4294967294
4294967295

55 66 77 88 99 127 128 129 155 166 177 188 199 254 255

-2147483648 -2147483647 -999999999 -88888888 -7777777 -777 -77 -7 7 77 777 77777
77 88888888 999999999 2147483646 2147483647

-1.#INF -9.9 -8.8 -7.7 -6.6 -5.5 -4.4 -3.3 -2.2 -1.1 0 1.1 2.2 3.3 4.4 5.5 6.6 7
.7 8.8 9.9 1.#INF
```