Objectives

1. To learn how to sort integers using radix sort.
2. To gain more experience with bit-wise operations in C programs.
3. To gain better understanding on the binary representation of integers and float point numbers.

Problem 1

Write a C program that sorts the ***unsigned integers*** provided through user inputs (***not from the command line***). Your program can assume that the count of the integers is not greater than 100, and each integer is 4-byte long. **Your program must use radix sort. It will NOT receive any points if it uses other sorting algorithms.**

Problem 2

Write a C program that sorts the ***real numbers*** provided through user inputs (***not from the command line***). Your program can assume that the count of the real numbers is not greater than 100. The real numbers can be processed as 4-byte float point numbers. Note that the numbers may be a mixture of positive numbers and negative numbers. **Your program must use radix sort. It will NOT receive any points if it uses other sorting algorithms.**

Instructions

The program should ask user for a count before it reads into the numbers. For saving the numbers and for the buckets, your program can use arrays, and the size of each array or bucket is 100. Another choice for this is to allocate 3 pieces of memory spaces using *malloc()* calls based on the count. Then, in a loop, the program asks user for the numbers (`scanf("%d", …)` or `scanf("%d", …)`), and save the numbers in an array or one piece of the memory space.

When sorting the numbers, ***use binary representation and bitwise operations to extract bits and then use the value of the bits as bucket indexes. DO NOT translate the numbers into strings (e.g., strings consist of '1's and '0's, or strings consist of digit characters like "31415926"), because this unnecessarily increases programming complexity and computation workload. DO NOT extract digits and use digits as bucket indexes.*** For example, the following code extracts a digit from *num* and use the digit as bucket index (*buck_idx*) in round *rnd_idx*. This method CANNOT be used in your program, because it is also much more costly than extracting bits.

```
buck_idx=(num /pow(10, rnd_idx)) % 10;
```

The program prints out the sorted numbers *with one number on each line*.

Applying bitwise operations directly to floating point numbers is not allowed in C. To apply bitwise operations on floating point numbers, your program needs to "instruct" the system to interpret the data as unsigned integers, which have the same length as floating point numbers. You can do this with typecasting using a pointer (check the related slides). This is also illustrated using the following example.

```
float f=123.45;
unsigned int *p = (unsigned int *) &f, value_of_bits4to7 = (*p & 0xF0)>>4;
```

Analyzing the following program and its output will also help you understand how to radix-sort float point values.

```
#include <stdlib.h>
main(){
   int i;
   float value, f[20];
   /* typecasting using a pointer */
   unsigned int *p = (unsigned int *) f;

   value = -10.5;
   for( i = 0; i < 20; i++) {
      f[i] = value;
      value = value + 1;
   } /* f has 20 float point numbers in ascending order */

   for( i = 0; i < 20; i++)
       printf("%.2f\t%u\n", f[i], p[i], (p[i]&0xFFFF0000)>>16);
   /* show the float point numbers (column 1) and the corresponding
    * unsigned int values (column 2). Bit-wise operations are used to
    * obtain the last column, showing that bit-wise operations can be
    * used after typecasting float point numbers to unsigned integers.
    */
}
```

Check the output of the program. The first column shows float point numbers in ascending order. Check column 2, which shows that sorting the corresponding unsigned int values in ascending order is not enough to ensure that the float point values are also in ascending order. By checking column 2, you can understand better the two methods for sorting float point numbers in the slides.

Testing

Test your programs manually first. Manually type in inputs, and check output.
To fully test your programs with more numbers, modify and use the following scripts. $1 of the script is the count.
Take screenshot when you test your programs manually, such that we can see the numbers provided to the programs and the output of the programs.

Bash script for testing the program radix-sorting unsigned integers:

```
#!/bin/bash
count=$1
rm input
rm your_output
rm standard_output
echo "================== input ======================"
echo ${count} | tee input
for (( i=0; i<${count}; i++ ));
do
```

```
  echo $RANDOM
done | tee -a input
echo "============= execution result ================"
cat input | PATH_NAME_OF_YOUR_PROGRAM | tee your_output
tail -n +2 input | sort -n > standard_output
echo "====== differences from correct result ======="
diff  your_output  standard_output
```

Bash script for testing the program radix-sorting float point numbers:

```
#!/bin/bash
count=$1
rm input
rm your_output
rm standard_output
echo "================== input ======================"
echo ${count} | tee input
for (( i=0; i<${count}; i++ ));
do
  printf "%d.%d\n" $((($RANDOM-$RANDOM)%1000)) $RANDOM
done | tee -a input
echo "============= execution result ================"
cat input | YOUR_PROGRAM | xargs printf "%.2f\n" | tee your_output
tail -n +2 input | sort -n | xargs printf "%.2f\n" > standard_output
echo "====== differences from correct result ======="
diff  your_output  standard_output
```