CISC 131
Introduction to Programming and Problem Solving
Spring 2020
Fractional Values in the Computer

Due: **Monday, April 6, 2020**
Points: **none**

Solutions to Problems from Lecture 6

Here is my solution to the acceleration color change exercise. Your implementation may have used an if statement but, as you can see, it isn't really necessary.

```
window.onload = function()
{
 var element;
 element         = document.getElementById("box");
 element.onclick = function() { changeColor ( element, 5000 ); };
};

function changeColor ( element, timeDelay )
{
 element.style.backgroundColor = getRandomRGB();
 element.onclick               = null;
 timeDelay                     = Math.max ( 100, timeDelay*.9);
 window.setTimeout(function() { changeColor(element, timeDelay);}, timeDelay );
}
```

Start of Today's Lecture

The computer does numeric processing using base two – binary. It uses binary because it is made up of switches and a switch has only two positions (states). A switch is either open or closed, If the switch is closed, current will flow through the switch. If the switch is open, current will not flow. We interpret these two states (current flowing, current not flowing) as the values one and zero. All processing done by the computer is based on this principle.

In order to represent a number in the computer, you need to set aside a group of these switches to represent the value of the number. Each switch can store one digit and each of those digits can store only one of two possible values.

The number you want to represent probably uses base ten (decimal). Each digit of a decimal number can store one of ten possible values. So, in order to store the decimal number on the computer, the decimal number must be converted from a representation that allows ten possible values per digit into a representation that allows only two possible values per digit. That is, the number must be converted from base ten into base two.

We have often converted integers from one base to another base and, as you may recall, the conversion of integer values from one base to a different base and back again is always done with no loss of accuracy. That is, if you convert an integer from base *X* to base *Y* and then convert it from base *Y* back to base *X*, the result before the conversions will be the same as the result after the conversions.

Each digit in an integer number that is represented using a positional number system (the system modern societies use) has its ultimate value determined by the intrinsic value of the digit (7 or 9, for example) multiplied by an amount that is derived from the position of the digit in the number.

The rightmost digit of the integer is multiplied by one. The digit to its immediate left is multiplied by the base (ten, if the number is represented in decimal, two if the number is represented in binary). Succeeding digits to the left are multiplied by the base squared, the base cubed, etc, just as we have done in class.

If you count the digits from rightmost to leftmost and start your count at zero, the count that is assigned to any given digit will tell you how to calculate the multiplier for that digit. The count is the power to which the base (ten for decimal, two for binary) is raised to determine the multiplier. Here is an example using the integer 134 represented in base five:

Count	2	1	0
Digits	1	3	4

Since the base is five, the decimal value of the number would be determined this way:

$(4 * 5^{0}) + (3 * 5^{1}) + (1 * 5^{2})$ which is

$(4 * 1) + (3 * 5) + (1 * 25)$ which is 44 in decimal.

A fractional value is a value that is greater than zero but less than one and this same counting scheme is used to represent fractional values. Now, however, you start counting with the digit immediately to the right of the radix point (decimal point for numbers represented in decimal), count from left to right, start the count at negative one, and subtract one from the count rather than adding one to the count.

The count will, again, tell you how to determine the multiplier. As it was with integer values, the count is the power to which the base is raised to determine the multiplier value. Here is an example showing the base five number .321

Count	−1	−2	−3
Digits	3	2	1

Since the base is five, the decimal value of the number would be determined this way:

$(3 * 5^{-1}) + (2 * 5^{-2}) + (1 * 5^{-3})$ which is

$(3 * 1/5) + (2 * 1/25) + (1 * 1/125)$ which is

$(3 * 25/125) + (2 * 5/125) + (1 * 1/125)$ which is

$86/125$ which is $.688$ in decimal.

The same approach is used for any base in a positional numbering system. The only thing that changes from one base to another base is the number that is raised to the power. For base ten, ten is raised to the power, for base twenty-one, twenty-one is raised to the power, etc.

Converting a number from one base to a different base requires finding a series of digits in the new base that represent the same value as the original number does in its original base. This is always possible with integers but rarely possible with fractional values. As an example, let's say we wanted to change the decimal value .123 into its equivalent base five value. Here is a table of base five values and there decimal equivalents

	5^{-1}	5^{-2}	5^{-3}	5^{-4}	5^{-5}
Decimal Value:	.2	.04	.008	.0016	.00032

We are converting into base five. Each base five digit has one of five possible values: zero, one, two, three, or four. Thus any base five digit can represent one of five possible decimal values. These decimal values are found by multiplying one of the five possible values of the base five digit by the base five power represented by the position of that digit in the number. For example, in the table above you can see that the first digit to the right of the radix point in base five is 5^{-1} which has the decimal equivalent of .2. The five possible decimal values that can be represented by this digit are:

Base Five Digit	Decimal Equivalent
0	0
1	.2
2	.4
3	.6
4	.8

When converting a number into base five, we determine the value to be assigned to each base five digit by asking this question: is the number we want to convert greater than or equal to a value that can be represented by this digit? If it is, we choose the setting of that digit to be the one that produces the greatest value that is still less than or equal to the value we are trying to convert and then subtract. The value represented by the setting is then subtracted from the value we want to convert giving us a new value to be converted.

Now we follow the same process with this new value and use the base five digit to the immediate right of the digit whose setting we just determined. This continues until the value we want to convert has been reduced to zero or we realize that an exact conversion is not possible.

Here is a table that shows the decimal value equivalents for each of the five possible base five digits for each of the first five negative powers of five.

Base Five Digit	5^{-1}	5^{-2}	5^{-3}	5^{-4}	5^{-5}
0	0.00000	0.00000	0.00000	0.00000	0.00000
1	0.20000	0.04000	0.00800	0.00160	0.00032
2	0.40000	0.08000	0.01600	0.00320	0.00064
3	0.60000	0.12000	0.02400	0.00480	0.00096
4	0.80000	0.16000	0.03200	0.00640	0.00128

Let's convert the decimal value .123 into its base five equivalent. We start with the 5^{-1} position. Other than zero, each decimal value in the range of for that digit is greater than .123, so we know that that position in the resulting base five number will have the setting of zero.

We now move to the next base five position of 5^{-2}. There are three values in the range represented by that digit that are less than or equal to .123 decimal We choose the largest of the them and set the base five digit to three. That means we can represent .12 of the .123 decimal value leaving .003 left to represent. Our base five equivalent number now has two digits: .03.

Now move to the next base five position of 5^{-3}. There is no value in the range of non-zero values that is less than or equal to .003 so the digit assigned to this position must be zero. The base five equivalent number is now: .030.

Applying the same process to the 5^{-4} position, we can assign one to that digit value. The base five equivalent number is now: .0301 and we reduce .003 by .016 and have .0014 as the remaining decimal value to be converted.

Now go to the 5^{-5} position, we can assign four to that digit value. The base five equivalent number is now: .03014 and we reduce .0014 by .00128 and have .00012 as the remaining decimal value to be converted.

We could continue this process until the remaining decimal value to be converted was zero or we decided that the remaining value was too small to worry about. While it is possible for some values to be converted exactly (which means that the remaining value to be converted became zero), most values either won't convert exactly or it would take far too many digits in the new base to represent them.

This is the major problem when converting fractional values from decimal into binary for use on the computer. The computer allocates each number a fixed number of binary digits that can be used to represent the number no matter how big or how small the number is. If all digits are used before the number to be converted becomes zero, then the binary result is only an approximation of the original decimal value. The hardware and software does what it can to keep these values within the proper range but it cannot always produce the results you desire or expect.

Let's convert the decimal value .123 into base two. To keep the example simple, assume that the computer has only seven bits (**bi**nary digi**t**s) in which to represent the value. Here is the value of each position starting at the radix point and working to the right.

	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
Decimal Value:	.5	.25	.125	.0625	.03125	.015625	.0078125

One nice thing about binary is that, while you still need to do the addition, you no longer need to do the multiplication when converting the number. The reason is that binary has only two values: zero and one. Multiplying by zero results in zero. Multiplying by one doesn't change the value. So, the table of possible binary values is much smaller than it was for the base five example shown above.

Base Two Digit	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
0	0	0	0	0	0	0	0
1	.5	.25	.125	.0625	.03125	.015625	.0078125

Using the same approach as described above for the base five example, we see that 2^{-1} is larger than .123 so we know the first digit of the result is zero. So the result now looks like this: .0

It is the same situation for the 2^{-2} and the 2^{-3} positions. Both are larger than .123, so we must use zero for both those digits. The binary number now has three digits, all of them zero: .000

At position 2^{-4} we find the first value that is less than or equal to .123 decimal. Therefore, we place the one in our result and subtract .0625 from .123 leaving .0605 left to convert. The binary result now looks like this: .0001

Since .0605 is greater than or equal to .03125, the value of the 2^{-5} position, we can place another one in our result and subtract .03125 from .0605 giving .02925. The result is now .00011

Using the same approach for 2^{-6}, the result becomes .000111 and the value left to convert becomes .013625. After applying the 2^{-7} position, the binary result is .0001111 and the value left to convert is .0058125.

Recall that we converted .123 decimal into base five, the value left to convert was .00012 after using only five base five digits. This is quite a bit less than the value left to convert when using

seven binary digits. Do you see why you will need more digits as the base you are converting to gets smaller? Each digit represents less in a smaller base than it would in a greater base.

Since we assumed we had only seven bits in which to represent the number and the value left to convert is still greater than zero, our binary result is just an approximation of the decimal value. If we had more binary digits, we could reduce it closer to zero but, for many values, it would never become zero.

To see what happens on the computer, we can write a short function that will show the result of adding .1 (one tenth) to a sum. If the sum starts at zero and we add .1 to it ten times, then the sum should become one. Here is a function that will help visualize what the computer is doing. The function contains a loop that executes ten times. Each time through the loop, the function adds .1 to *sum* and then concatenates sum onto a *String result*. The function returns *result*.

```
function detail()
{
 var i;
 var result;
 var sum;

 result = "";

 sum    = 0;
 i      = 0;
 while ( i<10 )
 {
  sum    = sum + .1;
  result = result + sum + "\n"; // new line character
  i = i + 1;
 }
 return result;
}
```

Put the function in a program and use this as your *window.load* function:

```
window.onload = function() { window.alert(detail()); };
```

Launch the browser. Are the results what you expected? Notice that some of the values are exact, some are smaller than what they should be, while others are greater than what they should be. They aren't much smaller or much greater but they are not exact and that is the important part. Assume, for example, you are writing a program that needs to produce a value in dollars and cents. Having it generate .249999999 instead of .25 is close but no one would consider it correct.

There is also a very important warning for programmers in the results shown in the sample program: when comparing numeric values, you must *never* rely on the result of comparing two non-integer values. Remember that integers always convert exactly so comparing integers is never a problem but comparing non-integers is always a huge problem. If *x* and *y* contain non-integer numbers, you must never try to compare them for equality such as:

```
x === y
```

Do you understand why? If not, ask yourself how long the loop will run in the following function:

```
function danger()
{
 var sum;
 sum    = 0;
 while ( sum !== 1.0 ){ sum = sum + .1; }
 return sum;
}
```