Appendix 2
Number Representations

There are many different ways to represent whole numbers. While we are comfortable counting in decimal (0,1,2,3,4,5,6,7,8,9,10,11,12,...), that is only one set of names for those values. There are other **numeral systems** (ways of representing numbers), such as Roman numerals: I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII,.... We also can count in base-one, which are tally-marks: |, ||, |||, ||||, 𝍸, 𝍸|, 𝍸||, ...

The important idea right now is that we have different ways to label the values, but it doesn't change the quantity itself. If there are 12 beans in a pile on the table, it doesn't matter what I call it: 12, XII, a dozen, 10_{12}, there are still that many beans sitting there.

We will discuss some other systems for naming whole numbers. Specifically, we will look at base 2 (binary) and base 16 (hexadecimal). For historical reasons, computer scientists are also occasionally interested in base 8 (octal).

Counting Up

We first cover a chart that shows how to count upwards in various bases. Whatever the base is, we cycle through the symbols for that base, and when we run out of symbols we clock over into the next column (like from 9 to 10, or from 599 to 600).

Each row gives the various names from various bases for the same quantity. So the base 10 number 13 is also represented as 1101 in base 2, as D in base 16 (and 15 in base 8).

We will use this chart throughout our discussion of numeral systems.

Note: in order to differentiate between numbers in various bases, we write the base as a subscript. Thus we can tell that 101_{10}, 101_2, and 101_{16} are all different values (101_{10}, 5_{10}, 257_{10} respectively). If we leave off a base, it is either base 10, or it is obvious (such as F8=F8$_{16}$).

Base 10:	Base 2:	Base 16:	Base 8:
0	0	0	0
1	1	1	1
2	**10**	2	2
3	11	3	3
4	**100**	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	**1000**	8	**10**
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	**10000**	**10**	20
17	10001	11	21
18	10010	12	22
19	10011	13	23
...	...	...	...

Different Bases

Let's think a bit more carefully about how we already represent numbers in decimal; then we can use the same description with a different base than 10. We usually assume base 10 when we write numbers, but we will often add subscripts to indicate the base, such as 245_{10}, 245_{16}, $2A8_{16}$, 10_{10}, 10_{2}, 10_{16}, etc.

Decimal

The base for our numbers is 10, and so we call decimal base 10. This base number indicates both how many symbols we have, as well as the values of each column in our numbers:

base 10: ten symbols. 0 1 2 3 4 5 6 7 8 9

base 10: column values:		…	10^3	10^2	10^1	10^0
	=	…	1000	100	10	1

We define an ordering to the symbols, e.g. so that $0 < 1 < 2 < \ldots < 8 < 9$. To increment our number means to represent a number that is one larger than previous. Our symbols are used, odometer-style, so that after we increment upwards and exhaust all the choices of symbol in one column, we can "roll over", or "overflow": we add one to the next column, and restart our numbering in the current column.

Roll-over examples:

9	+1→	10
29	+1→	30
419	+1→	420
1999	+1→	2000

We know of other situations where we have this overflow/roll-over situation. (Consider adding one more unit in each case):

- after 2:59pm, we reach 3:00pm.
- after February 29th, 11:59pm, we reach March 1st, 12:00am.
    - The whole 12-is-the-start-of-the-next-day bit also emphasizes that we can choose any ordering for our symbols, as well.
- after 1 quart 3 cups, we reach 2 quarts zero cups (because there are 4 cups in a quart).
- after 3 weeks and 6 days, we have 4 weeks and zero days.
- after using 2 dimes and 4 pennies for 24 cents' change, we would use 1 quarter to make 25 cents' change. Note that the 'column' values here are not all powers of the same base: 1, 5, 10, 25. Thus some of the useful properties we rely on for other numeral systems are not available in counting out the best change. For instance, we didn't clock over to get 2 dimes and 1 nickel, we combined those to get 1 quarter. Incidentally, if we had 20-cent pieces instead of quarters, this situation wouldn't arise: 20 is a multiple of all the previous pieces, unlike 25, so the best change would not have this same issue.

Example: When we see the number 423 (meaning 423_{10}), we know that it equals 400+20+3. We even say "four hundred twenty-three", because base 10 is so embedded in our numeral system. We are seeing that each column in the number has a weighting that is found via the base. Each column is worth 10 raised to a successively higher power, *starting with zero exponent at the rightmost column*:

Digits of 423_{10}:	...(0)	(0)	4	2	3
Column Values:	...	$10^3 = 1000$	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
value in each column:	(0)	0	4*100	2*10	3*1

$\rightarrow$sum: $400+20+3 = 423_{10}$

Hexadecimal

Hexadecimal is base 16. Since we need 16 symbols, and we only have ten common numeric symbols, we just borrow the first 6 letters of the alphabet to complete our set.

base 16: sixteen symbols. 0 1 2 3 4 5 6 7 8 9 A B C D E F

base 16 column values: ... 16^3 16^2 16^1 16^0
... 4096 256 16 1

Roll-over is the exact same idea as before: when we run out of symbols in a column, we start over in this column with zero, while adding one to the column to the left.

F $\rightarrow$ 10
3F $\rightarrow$ 40
FFF $\rightarrow$ 1000
7FFF $\rightarrow$ 8000
9 $\rightarrow$ A (not roll-over, just the next symbol of 16)

Example: $1AD5_{16}$. We first construct our columns' values, and then think of the number in each column representing that many times the column's value. Placing column values beneath our number's symbols:

Hexadigits:		1	A	D	5
Column Values: (written in base 10)	...	$16^3 = 4096$	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
value in each column:	...	$1_{10}*4096_{10}$	$10_{10}*256_{10}$	$13_{10}*16_{10}$	$5_{10}*1_{10}$

$\rightarrow$ sum: $(1*4096) + (10*256) + (13*16) + (5*1) = 6869_{10}$

We see that our number is worth $(1_{16}*4096_{10}) + (A_{16}*256_{10}) + (D_{16}*16_{10}) + (5_{16}*1_{10})$. We look up A in our chart and realize it's just 10_{10}, and similarly that D is just 13_{10}. ($1_{16}=1_{10}$ and $5_{16}=5_{10}$). So we have, all in base 10, $(1*4096) + (10*256) + (13*16) + (5*1) = 6869_{10}$. We actually prefer to just write the second version, where everything is written in base 10. We are most comfortable doing calculations in decimal, and mixing bases within calculations is simply an invitation for calculations errors.

Binary

Binary is base 2. Thus there are only two symbols, and columns are worth powers of two. But all of our ideas carry over:

base 2: two symbols. 0 1

base 2 column values: ... 2^6 2^5 2^4 2^3 2^2 2^1 2^0

= ... 64 32 16 8 4 2 1

Roll-over examples: (affected bits are highlighted)

1 → 10
111 → 1000
1101 → 1110

With only two symbols, we roll over quite frequently. Every other increment involves roll-over!

Conversions Between Bases

After looking at the initial chart and reading through our descriptions of each base, we can now discuss how to convert between one representation and another.

Other bases to decimal

To convert from non-decimal bases to decimal, we follow the process described above to figure out what the value being represented would look like in base ten. Here is an algorithmic description:

1. Using the number's original base and exponents from zero and up, figure out the value of each column that your number utilizes.
2. For each column, multiply the column's value by the quantity represented at that column. (Write all of this in base 10 for simplicity's sake – instead of B_{16}, we prefer to write 11_{10}, or even just 11).
3. Add these results from each column to reach the base 10 result.

Examples

- What is $E3_{16}$, in base 10?
  $(14*16^1) + (3*16^0) = 14*16 + 3*1 = 224 + 3 = 227_{10}$.

- What is 103_{16}, in base 10?
  $(1*16^2) + (0*16^1) + (3*16^0) = 1*256 + 0*16 + 3*1 = 256+0+3 = 259_{10}$.

- What is 1101_2, in base 10?
  $(1*2^3) + (1*2^2) + (0*2^1) + (1*2^0) = 1*8 + 1*4 + 0*2 + 1*1 = 13_{10}$.

- What is 100010_2, in base 10?
  $(1*2^5) + 0 + 0 + 0 + (1*2^1) + 0 = 32 + 2 = 34_{10}$.
  - note that we might as well only focus on the non-zero columns to speed up the calculation.

- What is 47_8, in base 10?
  $(4*8^1) + (7*8^0) = 32 + 7 = 39_{10}$.

Your Turn!

- Convert the following to decimal:
  - 10110_2. 1111_2. 1010_2.
  - 36_{16}. $1A_{16}$. $B2_{16}$. $E47_{16}$
  - 31_8. 10210_3. $3H_{20}$ (just use more letters in order for the symbols)

Decimal to Other Bases

We use a different approach to convert from decimal to other bases. The reason we don't just use the previous version (it could work just fine) is that we are not trained in performing addition and multiplication in other bases, so we prefer an approach that allows us to do our calculations in decimal.

Version 1: Subtracting column-values until we have nothing left.

Think of a number that you want to represent in some base; as an analogy, think of your number as a pile of beans.

- Let's assume we have 253 beans.

You want to put all the beans into smaller groups, which happen to be of sizes 1000, 100, 10, and 1. Starting with the largest groupings possible, we first take out as many of that large group-size as possible.

- we take out two 100-bean groups, leaving us with 53 beans left.

Next, we look at the next-largest grouping, and see how many piles of beans of that size we are able to take from our pile of 53 beans.

- we take out five 10-bean groups, leaving us with 3 left.

We continue through each smaller-sized column, until we run out of beans or reach the right-most column (at which point we'd better have run out of beans too, or else we made a mistake somewhere in our actions).

- we take out three 1-bean groups, leaving us with no beans left over.

This same process works, no matter what base you started in (how you initially label your pile of beans), or what base you're going to (the various group-sizes/column values that are allowed).

Your Turn!
Again consider 253_{10} beans. But now we want to take out piles of sizes 512, 64, 8, and 1.
→ how many 512-sized piles do you get? (how many are left over?)
→ how many 64-sized piles do you get? (how many are left over?)
→ how many 8-sized piles do you get? (how many are left over?)
→ how many 1-sized piles do you get? (how many are left over?)

→ What base is this representing, and now what would 253_{10} beans look like in that base?

Algorithm:

- Find the target base's column values, up to the last column that isn't bigger than your quantity (your starting number). A column exactly equal to your number should be included.
- Start with that leftmost/largest column. For the current column, as many times as you can, subtract the column's value without getting a negative result. Each time you do so, add one to this column as part of your result. (Like trading 25 pennies for 1 quarter).
- Now that you can't subtract that column any more, you have less than the column's value; move one column to the right, and repeat.
- As soon as you run out of quantity to redistribute (no more beans), place a zero in any remaining columns; your entire result is complete.

Example: Convert 53 to binary.

1. find out the column values. Any columns bigger than our number are filled with zeroes (and thus we don't show them in the final result). Value remaining: 53.

64	32	16	8	4	2	1
0						

2. Looking at the 32 column (=2^5), we see that 32≤53, so we put a 1 in the 32 column and subtract 32 from our number. (32 was the largest column value that was ≤53, so that is where we begin).

64	32	16	8	4	2	1
0	1					

Remaining value: 53-32 = 21.

3. proceed to the next column, again seeing if we should transfer part of our remaining value into this column. Repeat through all remaining columns.

16≤21. → 1 in the 16's column. Remaining value: 21-16 = 5.

64	32	16	8	4	2	1
0	1	1				

8>5. → 0 in the 8's column. Remaining value: 5-0 = 5.

64	32	16	8	4	2	1
0	1	1	0			

4≤5. → 1 in the 4's column. Remaining value: 5-4=1.

64	32	16	8	4	2	1
0	1	1	0	1		

2>1. → 0 in the 2's column. Remaining value: 1-0=1.

64	32	16	8	4	2	1
0	1	1	0	1	0	

1≤1. → 1 in 1's column. Remaining value: 1-1=0.

64	32	16	8	4	2	1
0	1	1	0	1	0	1

Final Result: 110101_2.

Example: Convert 746 to hexadecimal.

Again, we start with listing out column values in the target base until we've gotten large enough to know that we must have a zero in the largest column – this tells us how many columns our number will need, and tells us where to stop.

Remaining value: 746.

0			
4096	256	16	1

256≤746. We also ask ourselves, "how many 256's can I extract from 746?" Since we can remove two 256's without going negative, we place a 2 in that column:

256≤746. 2*256≤746. 3*256>746 (too many). → 2 in 256's column. Remaining value: 746-2*256 = 234.

0	2		
4096	256	16	1

16≤234. "How many 16's can I subtract from 234?" 14*16≤234, but 15*16>234. → 14 in 16's column (represented by E). Remaining value: 234 – (14*16) = 234 – 224 = 10.

0	2	E	
4096	256	16	1

1≤10. "How many 1's can I subtract from 10?" Ten 1's will fit → 10 in 1's column (represented by A). Remaining value: 10 – (1*10)=0.

0	2	E	A
4096	256	16	1

Final Result: $2EA_{16}$.

Your Turn!

- Convert 46_{10} to binary.
- Convert 243_{10} to binary.
- Convert 63_{10} to binary.
- Convert 67_{10} to hexadecimal.
- Convert 79_{10} to hexadecimal.
- Convert 5000_{10} to hexadecimal.
- As a warm up to the next section (after version 2), convert 59_{10} to both binary and hexadecimal.

Version 2. Division and remainders approach.

We already understand that we find out if a number is even by dividing it by two, and checking if the remainder is a 1 or a 0. It turns out that every even number has a 0 in the 1's column in binary representation, and all odd numbers have a 1 in the 1's column in binary.

Using the number 11_{10} as an example, let's try to use division to consider the next bits of our result. If we consider the result of division, it tells us how many 2's we got. 11/2 = 5R1. This just tells us that 5*2 + 1 = 11. If we consider the five 2's, how would the 5 look in binary? Well, we know 5/2 = 2R1, so it ends in a 1. How would 2 look in binary? Well, we know it ends in a 0. 2/2= 1R1. How would 1 look in binary? 1/2=0R1, so it ends in a 1. We have nothing left, so we're done.

If we keep on dividing by 2, and record the remainders from right to left, stopping when we get a quotient of zero, we can also find out the binary representation of a number:

What is 23_{10} in binary?

23/2	= 11R1	→	**1**
11/2	= 5R1	→	**1**1
5/2	= 2R1	→	**1**11
2/2	= 1R0	→	**0**111
1/2	= 0R1	→	**1**0111

Result: 10111_2. Again, notice that we found the rightmost bit first! We can remember this because finding the remainder when dividing by two (just once) tells us if the number is even or odd, which is evident in just the last (rightmost) column.

The same approach works for other bases; just divide by that base:

5364 / 16	= 335 R 4	4	
335/16	= 20 R 15	F4	(15_{10} represented by F_{16}).
20 / 16	= 1 R 4	4F4	
1 / 16	= 0 R 1	14F4	

Final answer: $14F4_{16}$.

Your Turn!

- Using this style of conversion, convert the following numbers to binary:
  - 13_{10} 8_{10} 23_{10}
- Using this style of conversion, convert the following numbers to hexadecimal.
  - 300_{10} 59_{10} 178_{10}

Converting between base 2 and base 16.

Based on the chart on the right (a smaller portion of the original chart shown above), we can see how four-bit binary patterns relate to single-symbol hexadecimal values. Leading zeroes have been added to the lower binary numbers so that we always see four bits. In order to convert from binary to hexadecimal, if we group a binary number in four bit groups starting from the right, we can use the chart to just convert 4 bits at a time. Conversely, we can convert from hexadecimal to binary by just converting each hexadecimal symbol to the corresponding 4-bit pattern.

Base 10:	Base 2:	Base 16:
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

This works because 2^4 =16. It's no coincidence that one base is worth a specific column of the other base ($16=2^4$). Hence we are using 4 bits to represent 16 values. Reconsider 59_{10} from before:

$59_{10} = 111011_2 = 3B_{16}$. Splitting the binary up as 4-bit groups ***starting from the right***, we get 11 1011. We can pad the leftmost group with 0's to make it 4 bits if it helps us think about it: **00**11 1011. 0011 = 3, 1011 = B. Result: $3B_{16}$.

Suppose we had some 32-bit number, where the bits were:

```
1010  0000  1111  1000  1111  0001  0110  1010
 A     0     F     8     F     1     6     A
```

= $A0F8F16A_{16}$. It is common to prefix a hex number with 0x instead of writing the $_{16}$ suffix: 0xA0F8F16A. This is because source code is just raw text; there's no superscript button in a plaintext editor ☺

Your Turn!

- Convert these binary numbers directly to hexadecimal:
  - 10 1010 1010 1010 1010
  - 10 1101
  - 1 0101 0101 0101 1000 1101 1011 0110 1001 1011 1110
  - 1111 1111 1111 1111 0000 1010 1010
  - 1111 1010 1100 1110
  - 1011 1110 1110 1111

- Convert these hexadecimal numbers to binary:
  - ACE 1234 1010 CAFE

Summary: Number Representations

You should be comfortable with the table of numbers that showed us how to count upwards. You should be comfortable converting between any two of these bases, in either direction: binary, decimal, hexdecimal.