CS216: Spring 2005
Problem Set 5 Comments

Question 1: Consider a simple hypothetical smart card with one 8-bit register using big-endian encoding. The current used for each operation scales linearly with the number of bits in the register that flip (either a 0 becomes a 1, or a 1 becomes a 0). The card implements a simple counter: the value in the register increases by one each clock tick. So, if the register initially contains *0000 0011* after the next clock ticks the resulting value in the register will be *0000 0100* and 3 units of current will have been consumed.

Suppose you measure the current consumed as shown below. What possible values could be stored on the card after clock tick 6?

For this question, the main thing to notice is how much current is consumed when a number is incremented. Let's look at some values of the current:

Value	Binary Rep	Current
0	0000 0000	8
1	0000 0001	1
2	0000 0010	**2**
3	0000 0011	**1**
4	0000 0100	**3**
5	0000 0101	**1**
6	0000 0110	**2**
7	0000 0111	**1**
8	0000 1000	4
9	0000 1001	1
10	0000 1010	**2**
11	0000 1011	**1**
12	0000 1100	**3**
13	0000 1101	**1**
14	0000 1110	**2**
15	0000 1111	**1**
16	0001 0000	5

...

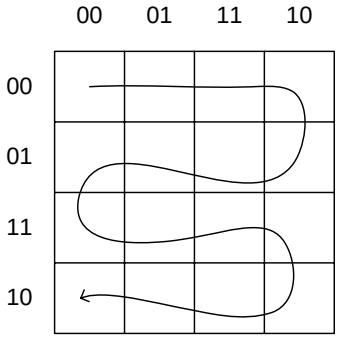
If you look at the current values, you can see that there are 2 places (in bold) where the sequence of current values would be equivalent to the graph given. So if you continue the pattern you can soon realize that any number with the last 3 bits equal to 1 are all possible.

Question 2: What is the fewest number of additional clock tick current readings that could be enough to uniquely determine the exact value stored on the card?

It is a "trick" question, because it is not possible to uniquely determine the number, since the pattern keeps repeating. Once you are at the bit pattern 1111 1111 or 0111 1111, you'd have to flip 8 bits, so you would never be able to differentiate between the two.

Question 3: Suggest a number representation the smart card designers could use that would be less vulnerable to power analysis attacks. If necessary, you may add extra bits to the register, although this is undesirable as it increases the cost of the smart card. Analyze the feasibility of a power analysis attack to determine the number stored in your card.

A better number representation would be if each consecutive digit required only one bit to flip. If you have taken DLD (CS/ECE 230) you might have realized that Karnaugh maps are a good way to do this. Basically suppose your number is 4 bits, you can use the following table (map numbers in the order of the arrow)



So basically you would represent your numbers as follows:

0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

For those that have not taken DLD: a Karnaugh map is a way to simplify logical functions. The only thing you need to know is that the numbers on top and the side represent the value of the four bites.

You would just extend this with another 4 digits to create an 8-bit number.

4 Problem 4

4.1 Pitfall

One way to get the size of int, long or any type in C is to use `sizeof` and multiply by 8. The problem is we do not know the number of bits for one byte. Although we are all know that one byte is 8 bits, historically one byte first refer to 6 or 7 bits and others number of bits also.

4.2 Solution

There are many ways to determine the size of `int` and `long` depends on the assumption we made.

Suppose we assume that both int and long in C use 2's complement to store the value. We can do the following

```
int determinebit() {
  int i;
  // Change the type to long, int, short or char
  type c;
  // for each loop, multiply c by 2, and increase the counter by one
  for(c=i=1;c>0;i++, c<<=1);
  return i;
}
```

The solution above has a problem when we want to determine the number of bits for unsigned. Another assumption we can make instead of 2's complement is the result of the arithmetic operation that is overflow is the part that is not overflow.

```
int determinebit() {
  int i;
  // Change the type to long, int, short or char, or unsigned
  type c;
  // for each loop, multiply c by 2, and increase the counter by one
  for(c=i=0;c;i++, c<<=1);
  return i;
}
```

5 Problem 5

5.1 Pitfall

The big pitfall here is value 1 in little endian will be represented as

10000000 00000000 00000000 00000000

and

00000000 00000000 00000000 00000001

for big endian.

The problem is it is little endian in term of byte, that means, 1 in little endian is

00000001 00000000 00000000 00000000

Another big pitfall is that many bitwise arithmetics operations only can be used to determine endianess. I do not see it is possible given that each bitwise operation base on the input value that it is all encode in the same way, either little or big endian.

5.2 Solution

```
void main(){
  int x = 0x01020304;
  char *t = &x;
  if (*t == 1)
     printf(\"Big Endian\");
  else if (*t == 4)
     printf(\"Little Endian\");
  else
    printf(\"Unknown !!!\");
  return 0;
}
```

6 Number Representation (Questions 6 through 10)

Recall that if the number $D_iD_{i-1}D_{i-2}\ldots D_0$ is in base k, that means the value of

$$D_iD_{i-1}D_{i-2}\ldots D_0 = D_ik^i + D_{i-1}k^{i-1} + \cdots D_1k + D_0.$$

For example $122_{10} = 1 \cdot 100 + 2 \cdot 10 + 2$.

The same idea can apply in this problem. However, using base 10 is not so efficient, we may need a better representation. There is no best representation, instead we need to make tradeoffs depending on the anticipated applications such as how often we need to parse string, or unparse, how much memory we care, need, how fast we want to do multiplication, and how much time we can spend.

6.1 String Representation

If we want to store value in base 10, one simple method to use is to use string as if it is a number. Parse and unparse would be fast, very fast. However, we need many instructions to do plus and multiply operation. Plus, there is almost no way to know if there is a carry out or not, hence, we created complexity on this.

6.2 Array Representation

Since memory is cheap, we may allocated more memory for the carry out. One may declare it like this

```
// l is the length of the number
// s is the total length of memory it is allocated ( may not need to use )
// d is the array of digit
typedef struct {
  int l,s,*d;
} *bignum,Bignum;

#define max(a,b) ((a)>(b))?(a):(b)
#define min(a,b) ((a)<(b))?(a):(b)
#define BASE 10000
#define LBASE 4

bignum Biginit(int s)
{
  bignum t=(bignum)malloc(sizeof(Bignum));
  t->d=(int*)malloc(sizeof(int)*s);
  memset(t->d,0,sizeof(int)*s);
```

```
  t->s=s;
  t->l=1;
  return t;
}

bignum bignum_create(char *b)
{
  bignum a;
  int i,k,l,p,j=strlen(b);
  // Allocate memory
  a =Biginit((j/LBASE)+1);
  // Divide number to chunks
  for (a->l=(j/LBASE)+!!(j%LBASE),l=0,i=0;i<j;l++) {
    a->d[l]=0;
    for(p=1,k=0;k<LBASE && i<j;k++,i++,p*=10)
      a->d[l]+=(b[j-i-1]-'0')*p;
  }
  return a;
}
// A compact way to unparse
char * bignum_unparse(bignum a)
{
  int i;
  char *strout,*tmpstr;
  char tmp[10];
  strout = (char*)malloc(sizeof(char)*(a->l*LBASE+1));
  sprintf(tmp,"%%0%dd",LBASE);
  sprintf(strout,"%d",a->d[a->l-1]);
  tmpstr = strout+strlen(strout);
  for(i=a->l-2;i>=0;i--, tmpstr+=LBASE)
    sprintf(tmpstr,tmp,a->d[i]);
  return strout;
}
```

6.2.1 Adding

Adding in this one is easy if the type of array that we use have enough bit space for overflow. For example `unsigned int` can hold up to 2^{32}. One may use base $1000,000,000$ and have problem when they want to add $999,999,999$ to 1. This is not impossible to do, but would take more effort to do.

```
bignum bignum_add(bignum a,bignum b)
{
// t
  int i,t,n;
  bignum c;
  n=max(a->l,b->l);
  c= Biginit(n+1);
  for(t=i=0;i<n||t;i++)  {
    // c->d[i] = t
    // if i < the length of a->l, add a->d[i] to c->d[i]
    // if i < the length of b->l, add b->d[i] to c->d[i]
    c->d[i]=((i<a->l)?a->d[i]:0)+((i<b->l)?b->d[i]:0)+t;
    t=c->d[i]/BASE;
```

```
    c->d[i]%=BASE;
  }
  c->l=i;
  return c;
}
```

This is clearly $\Theta(n)$ where n is the length of the number.

6.2.2 Multiplication

One arguably the easiest way to implement multiplication for this kind of data structure is to do like long multiplication.

```
bignum bignum_mult(bignum a,bignum b){
  int i,j;
  bignum c;
  c = Biginit(a->l+b->l+2);
  for(i=0;i<a->l;i++){
    for(j=0;j<b->l;j++)
      c->d[i+j]+=a->d[i]*b->d[j];
    c->l = c->s;
    // Carry is to be present in the next code
    carry(c);
  }
  return c;
}
```

The asymtotic running time is clearly $\Theta(n^2)$.

How fast can we do multiplication? One may think of divided and conquer to make a better multiplcation. If we know Strassen's for matrix multiplication, you might try to do the same with this. Instead of multiply an integer to the bignum, we abstract it in the meaning ful way.

Suppose we want to multiply X and Y. We can rewrite X and Y as $aB^k + b$ and $cB^k + d$ respectively where B is the base.

Hence

$$XY = acB^{2k} + (ad + bc)B^k + bd$$

We know that $(a+b)(c+d) = ac+bd+ad+bc$, hence $(ac+bd+ad+bc)B^k+acB^{2k}-acB^k+bd-bdB^k = acB^{2k} + (ad + bc)B^k + bd = XY$.

So, instead of multiply using long multiplication directly of XY, we break X and Y and multiply $(a + c)(b + c), bd$, and ac. Note that k is arbitary positive number, so if we pick $k = n/2$ where n is the size of the array. So every multiplications, we have in this algorithm is the size of $n/2$. We can formulate the running time as

$$f(n) = 3f(n/2) + cO(n)$$

Solving this equation, we have $f(n) \in \Theta(n^{\lg_2(3)}) \approx \Theta(n^{1.586})$.

This algorithm name is Karatsuba multiplication.

So how fast can we do multiplication? The very similar to this approach, but instead of divide number to two, it divides the number to three parts. It is called Toom-Cook.

In theory , we can always do multiplication in $\Theta(n^{1+\epsilon})$ where $\epsilon > 0$ with the more number of parts we want to divides, however, we introduce more constant, which is not account in the theory. Usually, people use Toom-Cook for large number then at some threshold change it to Karatsuba, and simple long multiplication.

Can we do it faster than $\Theta(n^{1+\epsilon})$, yes. We can view number as a series of digits, and apply Fourier transform method. The fastest method known has time complexity of $O(n \ln(n) \ln(\ln(n)))$ found by Schonhage and Strassen.

There is also another suggest by Knuth that we can actually create a constant running time multiplication algorithm provided that we have infinitely many resources.

Below is the implementation of Karatsuba algorithm,

```
// The carry out here is not a simple carry out, it is the carry out that allow some negative number.

bignum carry(bignum a)
{
  int t,i;
  for(t=i=0;i<a->l || t;i++) {
    a->d[i]=(i<a->l)?a->d[i]:0;
    a->d[i]+=t;
    if (a->d[i]<0){
      t=a->d[i]/BASE-1;
      a->d[i]+=(-t)*BASE;
    }
    else{
      t=a->d[i]/BASE;
      a->d[i]%=BASE;
    }
  }
  while(i>0 && !a->d[i-1]) i--;
  a->l=i+!i;
  return a;
}


bignum multi(bignum x,bignum y,bignum o)
{
  int n=max(x->l,y->l);
  int i;
  int s=n>>1;
  Bignum A,B,C,D;
  bignum ab,cd,t;
  if (x->l==0|| y->l==0)  {
    o->d[0]=0; o->l=1; carry(o);
  }  else
    if (n==1)
      {
o->d[0]=x->d[0]*y->d[0];
o->l=1;
carry(o);
      }
    else  {
      // x = A*B^s + B
      A.d=x->d+s;   A.l=max(min(n-s,x->l-s),0); A.s= A.l;
      B.d=x->d;     B.l=min(s,x->l);            B.s= B.l;
      // y = C*B^s + D
      C.d=y->d+s;   C.l=max(min(n-s,y->l-s),0); C.s= C.l;
      D.d=y->d;     D.l=min(s,y->l);            D.s= D.l;
      ab=bignum_add(&A,&B);
      cd=bignum_add(&C,&D);
      t=Biginit(2*(n+4));
```

```
      multi(ab,cd,t);
      memset(o->d,0,sizeof(int)*o->s);
      o->l=0;
      for(i=0;i<t->l;i++)  o->d[i+s]=t->d[i];
      multi(&A,&C,t);
      for(i=0;i<t->l;i++)   {
o->d[i+(s<<1)]+=t->d[i];
o->d[i+s]-=t->d[i];
      }
      multi(&B,&D,t);
      for(i=0;i<t->l;i++) {
o->d[i]+=t->d[i];
o->d[i+s]-=t->d[i];
      }
      o->l=o->s;
      carry(o);

      bignum_free(ab);
      bignum_free(cd);
      bignum_free(t);
    }
  return o;
}
bignum bignum_mult2(bignum a,bignum b){
  bignum c;
  c = Biginit(a->l+b->l+2);
  return multi(a,b,c);
}
```

6.3 Encode in the array, threat number as power of 2 base, i.e. $2^{16}, 2^{32}$

One may argue that we waste memory space with base power of 10. (although memory is cheap, some people case).

Now, we have the problem call, radix conversion, convert binary to decimal or power of 10 or another way around.

For `bignum_create`, we need to convert decimal to binary.

```
// Psuedo, not C
bignum x = bignum(0);
int i,len = strlen(str);
// str is the string of number, i.e. "13455678"

for(i=0;i<len;i++)
   x = big_add_int(big_mulint(x,10),str[i]-'0');
```

The algorithm above is of order $\Theta(n^2)$. Another approach is to group the number together before multiply, which has the same running time complexity, but decrease the constant.

We also need to convert binary back to decimal in `unparse`.

This can be done by a simple base convertion, which lead to $\Theta(n^2)$.

Note that both convert from decimal to binary and binary to decimal can be done using divide and conquor also, which the running time is $\Theta(n(\ln n)^2)$ depends on the implementation of multiplication and divide operations.

Note that the main reason why using number base system greater the square root of biggest number system you can use is not a good idea is that it is hard to do multiplication, and divide.

```
// limb -> unsigned int
// Note that there is a single assembly instruction to do this operation on 0x86.
inline void mulhilow(limb *hi, limb *low, limb m0, limb m1){

  // This is slow, but I don't want to use assembly
  //  Suppose that limb is the biggest type that can operate on that machine
  limb adbc,bd ;
  bd =  (m1 &LOMASK) * (m0 & LOMASK);
  adbc =  (m1 >> HALF) * (m0 & LOMASK)  + (bd>> HALF) ;
  bd &= LOMASK;
  adbc +=  (m1 &LOMASK) * (m0 >> HALF);
  *hi =  (m1 >>HALF ) * (m0 >> HALF) + (adbc >> HALF) ;
  *low =   (adbc <<HALF)+bd;
}
```