2015-11923 최한결

```
172  int bitOr(int x, int y)
173  {
174    int z = ~x & ~y;
175    return ~z;
176  }
```

드 모르간의 법칙 ~(~x & ~y) = x | y 을 이용해 간단하게 구현 가능하다.

```
186  int isAsciiDigit(int x)
187  {
188    int a = x & (~0x3f); //should be 0
189    int b = 0x10 + x;
190    int c = 0x06 + x;
191    b = b >> 6; //should be 1
192    c = c >> 6; //should be 0
193    return (!a) & b & (!c);
194  }
```

x가 0x30 과 0x39 사이의 값이면 오른쪽으로부터 8번째 이후 bit들은 모두 0이여야 하므로 x가 ascii digit일 때 a는 0이다. a가 0을 만족시키는 수들은 7개의 비트 내에서 수를 비교 가능하다.
X >= 0x30이므로 x + 0x10을 하면 오른쪽으로부터 7번째 bit값(b)은 1, x <= 0x39이므로 x + 0x06의 오른쪽으로부터 7번째 bit값(c)은 0이 되어야 한다.

```
203  int getByte(int x, int n)
204  {
205    int y = x >> (n << 3);
206    return y & 0xFF;
207  }
```

n번째 byte의 값이므로 x를 8n bit만큼 오른쪽으로 이동 후 제일 오른쪽 1byte만 return시킨다.

```
217  int byteSwap(int x, int n, int m)
218  {
219    int nbyte = n << 3;
220    int mbyte = m << 3;
221    int nth = (x >> nbyte) & 0xff;
222    int mth = (x >> mbyte) & 0xff;
223    int nmEmpty = ~((0xff << nbyte) | (0xff << mbyte)) & x;
224    int answer;
225    nth = nth << mbyte;
226    mth = mth << nbyte;
227    answer = nmEmpty | nth | mth;
228    return answer;
229  }
```

nbyte와 mbyte의 값을 추출 후 각각 nth, mth라고 하고 x에서 n번째와 m번째 bytes를 비운 값을 nmEmpty라고 한다. nmEmpty에 mbyte 이동한 nth와 nbyte이동한 mth를 넣으면 swap이 완료된다.

```
237  int bang(int x) {
238    //there should be no '1' so use 'or' for all bits
239    int a = (x << 16) | x;
240    int b = (a << 8) | a;
241    int c = (b << 4) | b;
242    int d = (c << 2) | c;
243    int e = (d << 1) | d; //if(x has an '1') 100000... else 0
244    int f = (e >> 31) + 1; //if(x has an '1') 111111...  + 1 = 0 else 00000... + 1 =
245    return f;
246  }
```

x가 0이면 의 모든 bits가 모두 0 값을 가져야 하므로 모든 bits를 각 bit끼리 논리합(or)으로 묶어도 0이 나와야 하고 x가 0이 아니면 모든 bit들의 논리합이 1이 되어야 한다. 32bits를 반씩 나눠가며 논리합을 했을 때 e의 제일 왼쪽 bit는 x가 0이면 0, 그렇지 않으면 1의 값을 가진다. 따라서 e를 오른쪽으로 31bit 이동하면 x가 0일 경우 0, 그렇지 않으면 -1의 값을 가진다. 여기에 1을 더해주면 !x를 구현 가능하다.

```
255  int fitsShort(int x)
256  {
257    // 2^16
258    int before16bits = (((x << 16) >> 16) ^ x) >> 16;
259    return !before16bits;
260  }
```

-2^15<= x <2^15인 x는 왼쪽 17개의 bit들이 모두 0이거나 모두 1이여야 한다. ((x << 16) >> 16)으로 왼쪽 17번째 bit에 따라 왼쪽 16개의 bit들이 모두 0 혹은 1의 값을 가진다. 이를 x와 xor 시켰을 때 왼쪽 16개 bit가 0이 아니라는 것은 x가 -2^15<= x <2^15 이 범위를 만족시키지 않는다는 뜻이다. 따라서 왼쪽 16개 비트에 !연산을 해 x가 범위 내에 있을 때만 1을 출력 가능하다.

```
268  int isTmax(int x)
269  {
270    int tMin = x + 1; //if x==Tmax tMin==Tmin
271    int zero = tMin + x + 1; //if x==Tmax zero=0
272    int answer = !tMin | zero;
273    return !answer;
274  }
```

x가 0x7fffffff이면 tMin은 0x80000000이고 zero는 0이다. x가 -1이면 tMin은 0이고 zero도 0이다. x가 0x7fffffff이나 -1이 아니면 zero는 0이 아니다. 따라서 tMin이 0이 아니고 zero가 0일 때 x는 0x7fffffff이다.

```
282  int negate(int x)
283  {
284    int answer = ~x + 1;
285    return answer;
286  }
```

x의 2의 보수를 구하면 된다.

```
296  int sign(int x)
297  {
298    int sign = x >> 31;       //if(x==negative) sign = 0xffffffff else -> sign = 0
299    int b = (sign << 1) + 1; //if(x==negative) b = 0 else b = 1
300    int c = ~(!x) + 1;        //if(x==0) c=-1 else c=0
301    int d = b + c;
302    return d;
303  }
```

Sign 은 x가 음수일 때 -1, 0 혹은 양수일 때 0이고 b는 x가 음수이면 -1, 0 혹은 양수일 때 1이다. c는 x가 0일 때 -1이고 그렇지 않으면 0이다. 따라서 d는 x가 음수일 때 -1, 0일 때 0, 양수일 때 1이다.

```
311  int addOK(int x, int y)
312  {
313    int sum = x + y;
314    int xySignEqual = x ^ y;    //if(x+y overflow) 0---...
315    int xSumSignEqual = x ^ sum; //if(x+y overflow) 1---...
316    int answer = ~xySignEqual & xSumSignEqual;  //if(x+y overflow) 1---...
317    answer = answer >> 31;
318    return !answer;
319  }
```

overflow가 일어나는 경우는 x와 y와 sum의 왼쪽 첫번째 bit가 각각 1,1,0 혹은 0,0,1일 때 이다. 즉 x와 y의 부호가 같고 x와 sum의 부호가 달라야 한다. 따라서 xySignEqual이 0, xSumSignEqual이 1로 시작해야 하면 overflow이므로 !answer은 overflow가 일어나면 0, 일어나지 않으면 1이다.

```
327  int isPositive(int x)
328  {
329    int isZero = ~!(~x + 1) + 1; //if 0 -> -1 else 0
330    int signBit = x >> 31;
331    int result = !signBit + isZero;
332    return result;
333  }
```

signBit가 x의 부호를 결정하므로 isZero를 통해 0일 때만 signBit에 1을 빼주는 방식으로 구현했다.

```
343  int satMul2(int x)
344  {
345    int mul2 = x << 1;
346    int sign = mul2 >> 31;              //if - -> 000000 else + -> 11111
347    int isOverflow = (mul2 ^ x) >> 31; // if overflow -> 0xffffffff else -> 0x0000000
348    int tMax = ~(1 << 31);
349    int overflowAdd = tMax & isOverflow; //if overflow -> 0x7fffffff else -> 0x000000
350    sign = !sign & isOverflow;           // minus & overflow-> 1  else-> 0
351    mul2 = (mul2 & ~isOverflow) + overflowAdd + sign;
352    return mul2;
353  }
```

왼쪽으로 1비트 움직이면 2를 곱한 값과 같다. overflow가 일어나는 경우만 처리하면 되고 2를 곱했을 때 overflow가 일어나는 경우는 왼쪽 두비트가 01혹은 10으로 시작할 때이다. 따라서 isOverflow를 통해 overflow가 일어날 때만 tMax가 되는 overflowAdd를 구하고 x가 음수이고 overflow가 일어날 때만 1이 되는 sign을 통해 overflow시에 tMax를 더하고 음수일 경우 1을 더 더해 satMul을 구현했다.

```
362  int absVal(int x)
363  {
364    int sign = x >> 31;
365    int negPlus1 = sign & 0x01; //if negative -> 1 else -> 0
366    int negNot = x ^ sign;      //if negative -> ~x else -> x
367    int result = negNot + negPlus1;
368    return result;
369  }
```

Sign은 x가 음수이면 0xffffffff 양수이면 0인 수이다. 이를 통해 x가 음수일 때는 x의 2의 보수 ~x+1을 구하고 그렇지 않을 땐 x를 구할 수 있는 두 수 negPlus1, negNot을 만들었다.

```
382  unsigned float_neg(unsigned uf)
383  {
384    unsigned exponent = (uf << 1) >> 24;
385    unsigned isNaN = !(exponent ^ 0xff) & !!(uf << 9); //exponent==0xff,fraction!=0
386    unsigned signInv = 1 << 31;
387    if (isNaN)
388      return uf;
389    return uf ^ signInv;
390  }
```

isNaN은 exponent가 0xff이고 fraction이 0이 아닐때 1이다. 따라서 x가 NaN일 때 uf를 그대로 리턴하고 그렇지 않으면 0x80000000인 signInv를 통해 uf의 sign만 바꿔준 후 리턴한다.

```
402  unsigned float_half(unsigned uf)
403  {
404    unsigned exponent = (uf << 1) >> 24;
405    unsigned isNaN = !(exponent ^ 0xff);
406    unsigned sign = uf >> 31;
407    unsigned expFraction = (uf << 1) >> 1;
408    unsigned fraction = (uf << 9) >> 9;
409    unsigned halfExp = exponent + 0xff;
410    if (isNaN)
411      return uf;
412    if (exponent < 2)
413    {
414      if ((expFraction & 0x3) == 0x3)
415        return (sign << 31) + ((expFraction + 1) >> 1);
416      return (sign << 31) + (expFraction >> 1);
417    }
418    return (sign << 31) + ((halfExp << 24) >> 1) + fraction;
419  }
```

uf가 NaN 혹은 ±infinite일 때는 uf를 그대로 출력하고 exponent가 2보다 작지 않을 때는 exponent만 오른쪽으로 한 bit이동한다. Exponent가 1 혹은 0일 때는 exponent와 fraction을 합쳐서 오른쪽으로 한 bit 이동한다. 이 때 오른쪽 두 bits가 11일 때는 반올림해 1을 더해준다.

```
435  int float_f2i(unsigned uf)
436  {
437    unsigned exponent = (uf << 1) >> 24;
438    unsigned sign = uf >> 31;
439    unsigned fraction = (uf << 9);
440    unsigned answer = 1;
441    if(exponent < 127)
442      return 0;
443    else if (exponent < 158){
444      exponent = exponent + 0x81;
445      while(exponent << 24){
446        answer = answer << 1;
447        if(fraction & 0x80000000){
448          answer = answer + 1;
449        }
450        fraction = fraction << 1;
451        exponent = exponent + 0x7f;
452      }
453      if(sign)
454        answer = ~answer + 1;
455      return answer;
456    }
457    return 0x80000000u;
458  }
```

exponent가 exponent <158일 경우에만 uf가 int의 범위 안에 있다. exponent가 127보다 작을 때는 uf를 십진수로 변환한 값이 -1보다 크고 1보다 작은 값이므로 0을 리턴하고 exponent가 158보다 클 때는 int의 범위를 벗어나는 값이므로 0x80000000u를 리턴한다.
Exponent가 127 < exponent < 158일 때는 while문에서 exponent에 1을 뺄 때 마다 answer와 fraction을 왼쪽으로 한 bit씩 움직이고 fraction에서 overflow되는 bit를 answer에 저장한다. 루프가 끝나면 uf의 fraction이 exponent의 크기만큼 answer에 저장되고 남은 fraction은 버려진다. 그 이후 sign값에 따라 sign이 1일 때만 answer에 2의 보수를 취해주어 음수를 나타내고 반환한다.