<시스템프로그래밍 과제1> -201511646 나여영-

Problem	**isZero** (long x)
Description	x의 값이 0일 경우 이를 False로 보고 ! 연산하면 1(True)이 되고, 0이 아닐 경우 True로 보므로 ! 연산하면 0(False)이 된다.
Answer	**return** !x

Problem	**bitOr** (long x, long y)
Description	or 연산을 하게 되면 0 과 0을 연산했을 때만 0이 나오고 그렇지 않은 경우에 1이 나오게 된다. A or B가 True라는 말은 ~A와 ~B 모두가 False가 아니다 라는 말과 같으므로 ~((~x)&(~y))와 동치이다.
Answer	**return** ~((~x)&(~y));

Problem	**bitAnd** (long x, long y)
Description	And 연산을 하게 되면 1(True)과 1(True)을 연산했을 때만 1(True)이 나오고 그렇지 않은 경우에 0(False)이 나오게 된다. A & B 가 True라는 말은 ~A 또는 ~B 가 False가 아니다 라는 말과 같으므로 ~((~x)|(~y))와 동치이다.
Answer	**return** ~((~x)|(~y))

Problem	**minusOne** (void)
Description	-1은 1의 2의 보수이므로 0x01에다가 2의보수를 취해준 형태가 답이다.
Answer	**return** -0x01+1

Problem	**negate** (long x)
Description	입력받은 x의 음수값을 반환하는 함수로 x의 2의보수 형태를 취해주면 된다.
Answer	**return** -x+1

Problem	**bitXor** (long x, long y)
Description	쉽게 표현하면 (~(x&y))&(x|y) 인데 x&y를 연산했을 시 1과 1의 연산만 1, 나머지는 0이고 여기에 ~ 연산을 하면 1과 1의 연산만 0, 나머지 연산의 결과값은 1이 된다. 또한 x|y연산을 했을 때는 0과 0의 연산만 0, 나머지 연산의 결과값은 1이 된다. 이 둘을 &연산 하면 결국 0 과0, 1과 1의 연산의 결과 값은 0, 1과 0, 0과 1의 연산의 결과값은 1이 되므로 결국 ^연산의 결과값과 같아진다.

Answer	**return** (~(x&y))&(~((~x)&(~y)))

Problem	**isPositive** (long x)
Description	isPositive는 양수일 때 1을 리턴하고 x가 0 혹은 음수일때 0을 리턴하는 함수이다. 양수와 음수의 구별되는 차이는 최상위비트가 양수는 0, 음수는 1 이라는 점이다. 따라서 (x>>63)&0x01 를 실행하면 제일 마지막에 위치한 비트만 양수면 0, 음수면 1이 된다. 하지만 우리가 원하는 결과는 양수일 때 1, 음수일 때 0 을 리턴 하는 것 이므로 !을 취해주어 양수일 때 0, 음수일때 1 을 리턴 하게끔 해준다. 문제는 !((x>>63)&0x01) 까지만 해 주었을 때 0은 1을 리턴 한다는 것이다. 따라서 0일 때 0을 리턴 하게끔 하기위해 !!x를 하면 0은 0을 리턴하고 0이외의 것은 1을 리턴 한다. 이 둘을 & 연산 하게 되면 (!!x&!((x>>63)&0x01) 앞에 조건이 0이기 때문에 추가되어도 뒤의 연산에 영향을 주지 않게 되어 원하는 결과를 얻을 수 있다.
Answer	**return** !!x&!((x>>63)&0x01);

ㅌ--

Problem	**getByte** (long x, long n)
Description	주어진 x에서 n번째 byte를 추출하는 함수로 n은 LSB부터 0으로 센다. 1바이트씩 이동하기 위해서는 총 8 비트를 left shift 해야 한다. n은 16진수가 아닌 10진수기 때문에 3만큼 left shift 했을 때 8비트를 이동하게 되고, n을 8비트만큼 leftshift 한 것 만큼을 x에서 right shift 하게되면 우리가 찾는 n번째 바이트가 LSB가 되고 나머지는 0으로 채워지게 된다. 이를 0xff와 &연산하게 되면 1과 1의 연산일때만 1이되는 &의 특징 때문에 마지막 LSB 만 살아남고 나머지는 다 0으로 바뀌어 원하는 값을 얻을 수 있다.
Answer	**return** 0xff & (x >> (n << 3))

Problem	**isNotEqual** (long x, long y)
Description	x와 y가 같으면 0을 리턴하고 다르면 1을 리턴하는 함수이다. x^y연산을 하면 x와 y가 같으면 0을 리턴, 다르면 -1을 리턴한다. 따라서 !!을 취해주면 x와 y가 같을 때 0을 리턴하고 다르면 1을 리턴하게 된다.
Answer	**return** !!(x^y)

Problem	**evenBits** (void)
Description	뒤에서부터 0으로 셌을 때 모든 짝수비트를 1로 바꾸어 0101... 즉 0x5555555555555555를 만드는 함수이다. left shift 와 +연산만을 이용할 경우 max operation을 지킬 수 없으므로 변수를 이용한다. long x = 0x55(=0x0000000000000055) 라고 둔 후 8비트 left shift 후 저장하면

	0x0000000000005555가 되어 다시 x에 저장되고, 이를 다시 16비트 left shift 후 저장하면 0x0000000055555555가 되어 다시 x에 저장되고, 이를 다시 32비트 left shift 후 저장하면 0x5555555555555555가 되어 원하는 값을 얻을 수 있다.

Answer	long x = 0x55;
	x = x + (x<<8);
	x = x + (x<<16);
	x = x + (x<<32);
	return x;

Problem	**reverseBytes** (long x)
Description	끝(LSB)에서부터 바이트 단위로 순서를 바꿔주는 함수이다. 바이트 단위를 살려주기 위해 0xff와의 &연산을 바이트 단위로 left shift해준 것과 한 후 각각을 바꾸는 위치만큼 right shift 한 후 더해주면 원하는 값을 얻을 수 있다.
Answer	**return** ((x&0xff)<<56) + (((x>>8)&0xff)<<48) + (((x>>16)&0xff)<<40) + (((x>>24)&0xff)<<32) + (((x>>32)&0xff)<<24) + (((x>>40)&0xff)<<16) + (((x>>48)&0xff)<<8) + ((x>>56)&0xff)

Problem	**conditional** (long x, long y, long z)
Description	x가 참일 경우(x가 0이 아닐 경우) y가 리턴 되어야 하므로, x가 0이 아닐 경우 x를 모든 비트가 1인 형태(-1, **1을 2의보수 취한 형태**)로 만들어 이를 y와 &연산 해야 하고, x가 0인 경우 모든 비트가 0인 형태(0, **0을 2의보수 취한 형태**)으로 만들어 이를 y와 &연산 해야 한다. 한편, !!x 연산을 했을 때 x가 0이 아닌 경우 1이 나오고, x가 0인 경우에는 0이 나오므로 !!x 연산을 한 결과를 각각 2의보수 취하여 모든 비트를 1 혹은 0으로 바꿔 y와 &연산을 해주면 x가 0이 아닐 때 y의 값을 그대로 유지하고, x가 0일 때 y값을 없앨 수 있다 (*((~(!!x)+1)&y)*). 반대로, x가 참이 아닐경우 (x가 0일 경우) z가 리턴되어야 하므로 x가 0일 때 z와 &연산하는 식이 1이 되어야 한다. 따라서 !x만 해주어도 현재 상태를 만족하지만, x가 참일 경우(x가 0이 아닐경우) 0이 되어야 하기 때문에 0의 2의 보수 취한 것이 0이 된다는 사실을 이용하여 ((~(!x)+1)을 z와 &연산 해주면 된다. 따라서 ((~(!!x)+1)&y) + ((~(!x)+1)&z) 을 하면 원하는 결과를 얻을 수 있다.
Answer	**return** ((~(!!x)+1)&y) + ((~(!x)+1)&z);

Problem	**isGreater** (long x, long y)
Description	일반적으로 크기 비교는 두 수의 차이의 부호를 이용할 수 있다. 본 문제에서는 x의 값이 더 큰 경우 1을 반환한다. 크기 비교를 할 수 있는 첫 번째 방법

	은 두 수의 부호를 비교하는 것이다. 만약, x의 부호가 양수이고 y의 부호가 음수라면 (x의 최상위비트가 0이고 y의 최상위비트가 1), x가 항상 y보다 크게 되므로 1을 리턴 할 수 있다. (*(!(x>>63))&(!!(y>>63))*). 한편, x와 y의 부호가 같은 경우(*(!((x>>63)^(y>>63)))*) 두 수의 정밀한 크기 비교가 필요하다. x가 y보다 크다고 가정하면, x에서 y를 뺀 수는 0보다 커야 한다. 그런데, 0보다 크다는 조건은 0이 포함되지 않으므로 최상위비트가 0이다는 조건을 이용할 수 없다. 그래서, x에서 y를 빼지 않고 y에서 x를 빼는 방법을 이용한다. y에서 x를 뺀 경우 무조건 음수가 나와야 하므로 (최상위 비트가 1이 나와야 하므로) y에서 x를 뺀 수를 63비트 Right Shift한 후 !연산을 두 번 해서 원하는 결과를 얻도록 하였다. (*(!!((y+(~x+1))>>63))*). 위의 두 case를 각각 **더하여** 문제에서 원하는 결과를 얻도록 하였고, x의 부호가 음수이고, y의 부호가 양수인 경우는 항상 x가 y보다 작아 항상 0을 리턴 해야 하므로 **위 연산에서** 고려되지 않았다.
Answer	**return** ((!(x>>63))&(!!(y>>63))) + ((!((x>>63)^(y>>63)))&(!!((y+(~x+1))>>63))))

Problem	**bang** (long x)
Description	x가 0과 0x8000000000000000 아닌 경우, x와 이의 2의 보수의 부호가 다르다 (최상위 비트의 값이 다르다). 이를 이용해서 x와 x의 2의 보수를 63비트 Right Shift하여 XOR 연산하여 둘을 구분할 수 있다. 0과 0x800000000000의 경우에 0이나오고, 0이 아닌 경우에는 -1이 나온다. 이 값에 각각 1을 더하면 x가 0x8000000000000000일 때를 제외하고 문제에서 요구한 값을 얻을 수 있다(*((x>>63)^((~x+1)>>63)+0x01)*). x가 0x8000000000000000인 경우, 다행히 0과의 큰 차이가 있는데 최상위 비트가 1이라는 점이다. 이를 이용해 x 자체를 63비트 Right Shift하여 1을 더하면 0이 되어 이 방법으로 해당 예외를 걸러줄 수 있다(*((x>>63)+0x01) &*). 물론 0을 이 연산에 대입하였을 때 결과가 1이 나오므로 결과에는 전혀 지장이 없다.
Answer	**return** ((x>>63)+0x01) & ((x>>63)^((~x+1)>>63)+0x01);