바킹독 0x01

태그	
공개여부	☑
날짜	
작성일자	

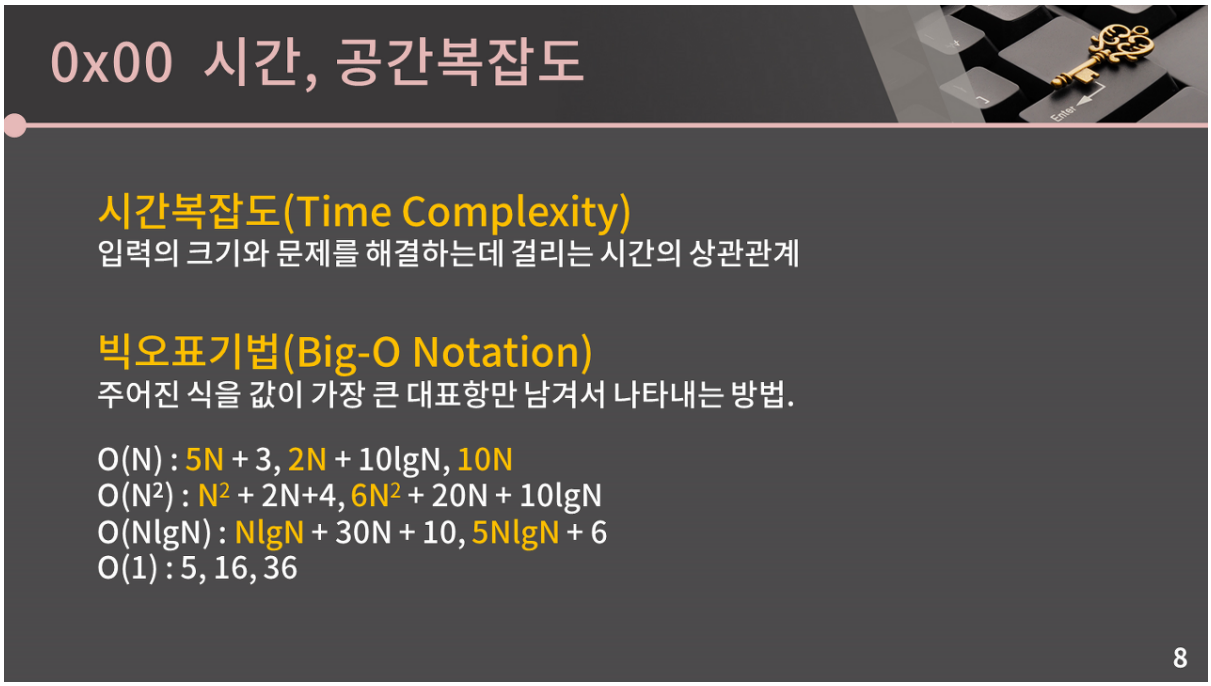
1. 시간/공간 복잡도

알고리즘 문제를 풀 때에는 시간제한과 메모리 제한이 주어집니다.

- 시간제한이 1초 = 내 프로그램은 3~5억 번의 연산 안에 답을 내고 종료되어야 한다는 것을 의미.

시간복잡도 결론 = 주어진 문제를 보고 풀이를 떠올린 후에 무턱대고 바로 그걸 짜는게 아니라 내 풀이가 이 문제를 제한 시간 내로 통과할 수 있는지, 즉 내 알고리즘의 시간복잡도가 올바른지를 꼭 생각해봐야 합니다. <내 코드의 시간복잡도, 주어진 문제의 입력 데이터 크기, 제한시간 확인!>

- 컴퓨터는 **1초에 대략 3~5억번의 정도의 연산을 처리할 수 있다.**



- ex) 입력 data의 크기가 N일 때, **최악의 경우** 연산을 lgN+4번 수행해야함.

  → **약 lgN번의 연산이 필요하다고(입력 데이터가 n개일 때) 퉁칠 수 있음.**

→ 시간복잡도(T(N)) = O(lgN)

=

입력 data의 크기가 N일 때, 연산을 약 lgN번 수행함.

=

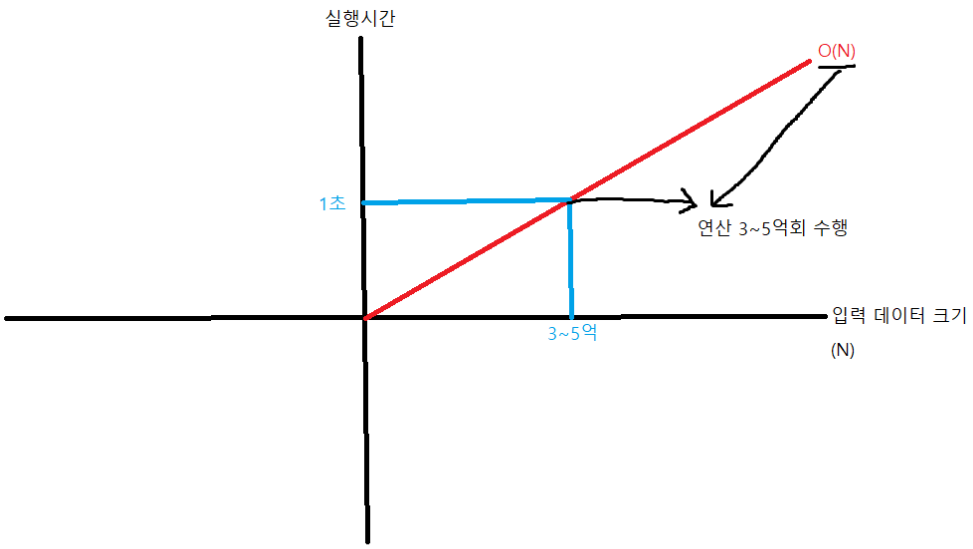
프로그램 실행시간이 lgN(대략적인 연산횟수)에 비례한다.

(실행시간과 연산횟수가 비례함)

=

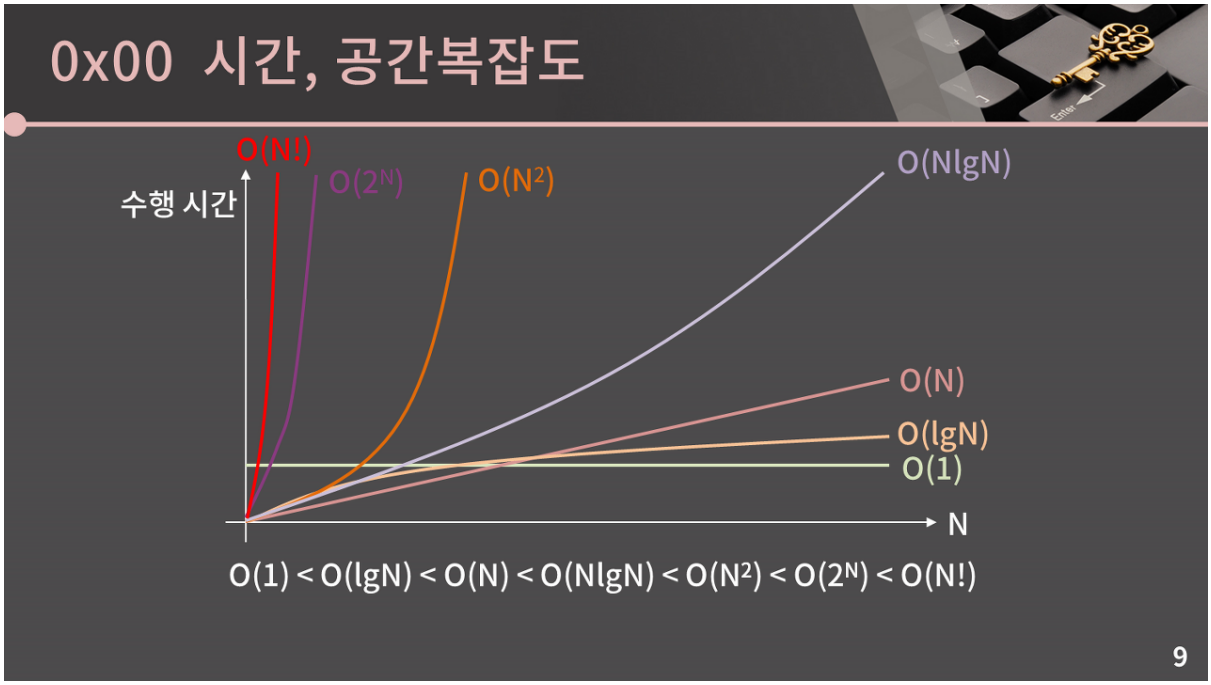
입력데이터가 늘어남에 따라 프로그램의 실행시간이 로그함수의 증가속도만큼 증가한다.

약 lgN번의 연산이 필요하다.



<u>알고리즘 a의 시간복잡도(T(n))가 O(n)일 때</u>

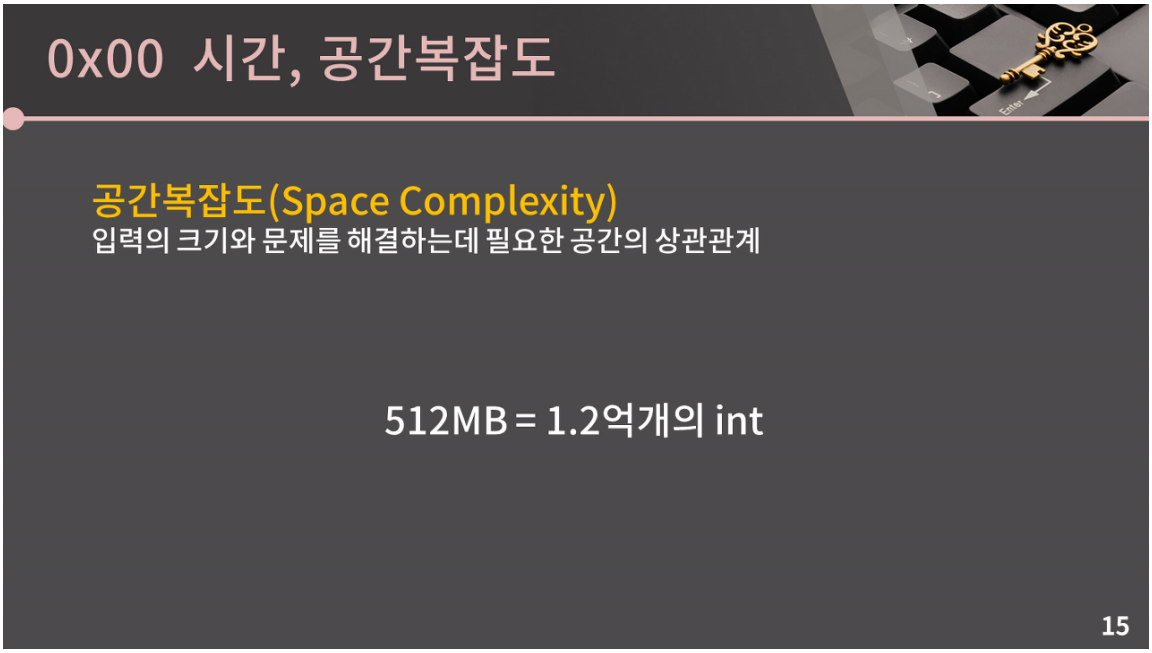
1. T(n) = O(n) 이라고 표기하며, '='는 'equal'의 의미보다는 '대략 이 정도 된다'라는 의미를 갖고있다고 보면 된다.
2. 알고리즘 a의 수행시간은 n에 비례한다. (수행시간이 입력크기에 비례한다, 입력크기가 2배되면 수행시간도 2배, 입력데이터가 늘어나면서 프로그램의 실행시간은 n의(선형함수의) 증가추세 정도로 증가한다)



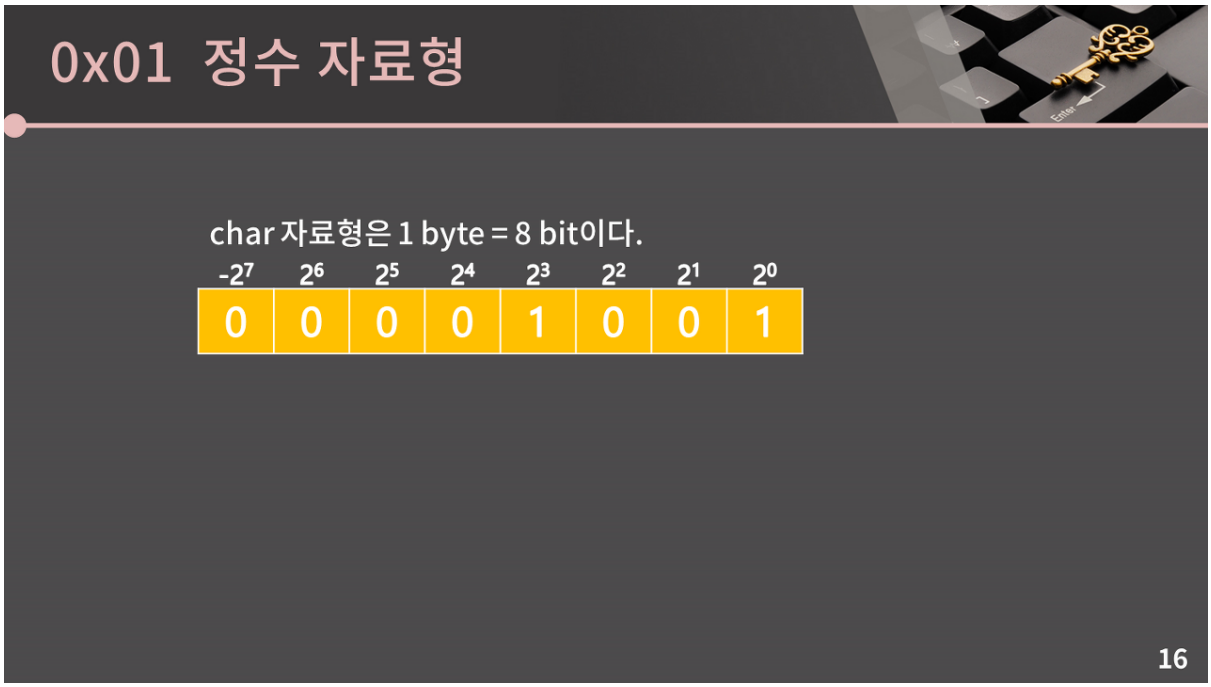
N 의 크기	허용 시간복잡도
N ≤ 11	O(N!)
N ≤ 25	$O(2^N)$
N ≤ 100	$O(N^4)$
N ≤ 500	$O(N^3)$
N ≤ 3,000	$O(N^2 lgN)$
N ≤ 5,000	$O(N^2)$
N ≤ 1,000,000	O(NlgN)
N ≤ 10,000,000	O(N)
그 이상	O(lgN), O(1)

- ex) 제한시간이 1초일 때, 내 프로그램의 시간복잡도는 O(N)이다.

  문제에서 주어진 N(입력 데이터의 크기)이 100만 정도라면 무난히 시간내에 돌아가지만, 10억 정도라면(최소 2초) 시간초과가 된다.


0x00 시간, 공간복잡도
공간복잡도(Space Complexity)
입력의 크기와 문제를 해결하는데 필요한 공간의 상관관계
512MB = 1.2억개의 int
15

2. 정수 자료형


0x01 정수 자료형
char 자료형은 1 byte = 8 bit이다.
-2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
0 0 0 0 1 0 0 1
16

0x01 정수 자료형

unsigned char

0	0	0	0	1	0	0	1	→ $2^3+2^0=9$
1	0	0	0	0	0	1	1	→ $2^7+2^1+2^0=131$

char

0	0	0	0	1	0	0	1	→ $2^3+2^0=9$
1	0	0	0	0	0	1	1	→ $-2^7+2^1+2^0=-125$

unsigned char 자료형의 범위 = 0 ~ $2^0+2^1 \cdots +2^7$ (= 2^8-1 = 255)

char 자료형의 범위 = -2^7(= -128) ~ $2^0+2^1 \cdots +2^6$ (= 2^7-1 = 127)

17

0x01 정수 자료형

short (2 byte)	$2^{15}-1$(=32767)
int (4 byte)	$2^{31}-1$(≒2.1×10^9)
long long(8 byte)	$2^{63}-1$ (≒9.2×10^{18})

18

만약 문제에서 unsigned long long 범위를 넘어서는 수를 저장할 것을 요구한다면 string을 활용해서 저장해야 합니다. 그리고 그 수로 연산을 해야 하는 문제는 코딩테스트에 안나오는 게 정상이긴 한데, 만에 하나 나왔다 치면 그냥 Python을 쓰는게 C++로 꾸역꾸역 구현하는 것 보다 훨씬 편합니다.

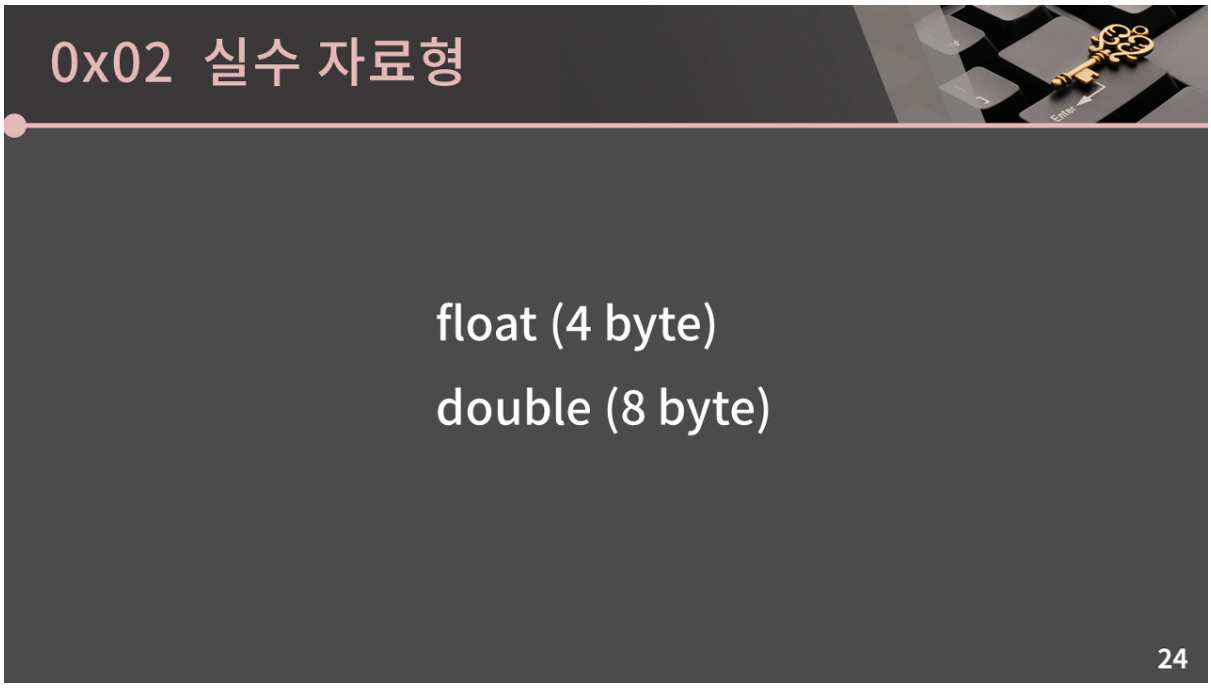
- Integer Overflow 주의

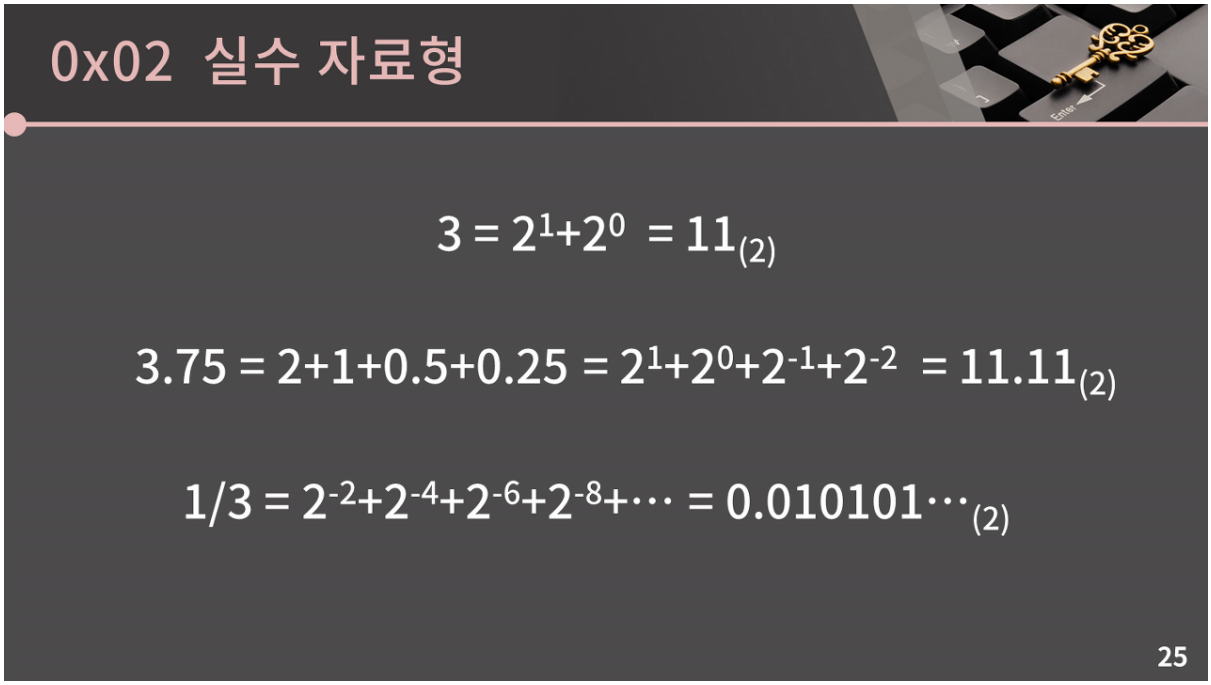
01111111 + 1 = 10000000

127 + 1 = -128이 되는 상황

Integer Overflow를 막는 방법은 아주 쉽습니다. 각 자료형의 범위에 맞는 값을 가지게끔 연산을 시키면 됩니다. 하지만 실제 코드를 짜다보면 Integer Overflow는 아주 빈번하게 일어나고, 또 찾기도 정말 힘듭니다.

3. 실수 자료





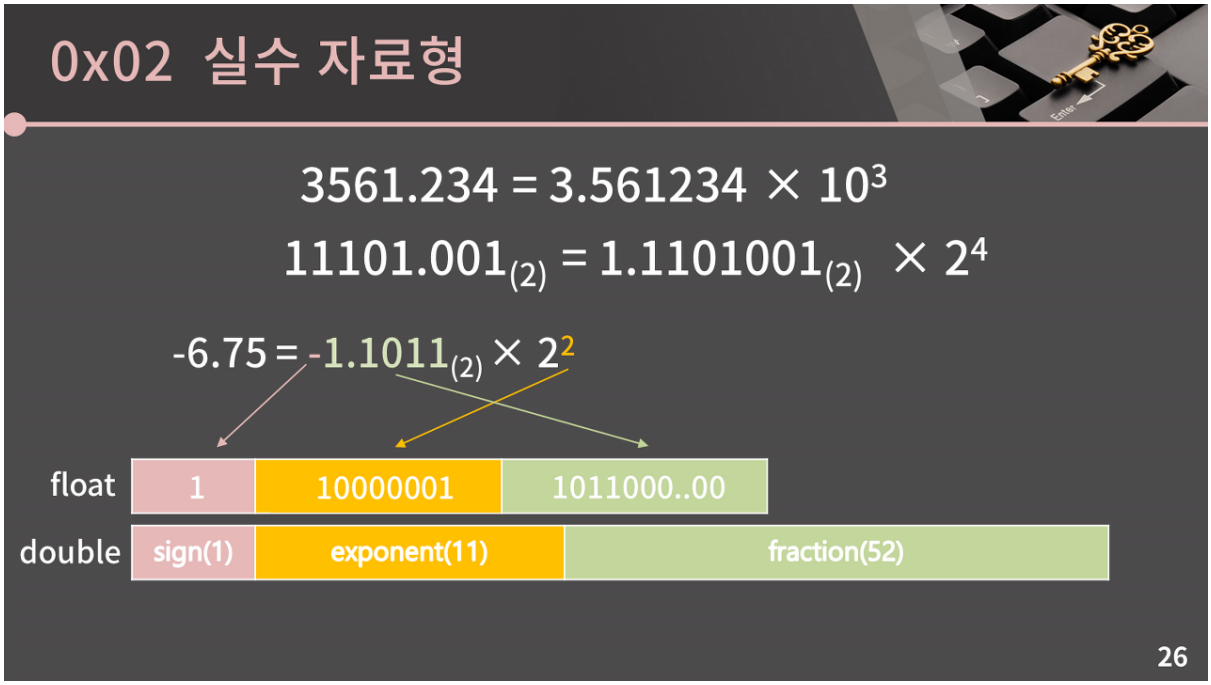
2진수를 실수로 확장해보자.

3.75 = 2 + 1 + 0.5 + 0.25 = 21 + 20 + 2-1 + 2-2 == 11.11(2)

2진수에서도 10진수와 마찬가지로 무한소수가 나타날 수 있다.

1/3 = 2-2 + 2-4 + 26 + ... = 0.010101...(2)

2의 음수 거듭제곱을 이용해 임의의 실수를 2진수로 나타낼 수 있다.
그 다음으로는 2진수에서의 과학적 표기법을 이해해야 합니다. 저희가 10진수에서 편의를 위해 3561.234 = 3.561234 × 10^3 으로 나타내는 것 처럼 2진수에서도 11101.001(2) = 1.1101001(2) × 2^4 으로 나타낼 수 있습니다.



그 다음으로는 2진수에서의 과학적 표기법을 이해해야 합니다. 저희가 10진수에서 편의를 위해 $3561.234 = 3.561234 \times 10^3$ 으로 나타내는 것 처럼 2진수에서도 $11101.001_{(2)} = 1.1101001_{(2)} \times 2^4$ 으로 나타낼 수 있습니다.

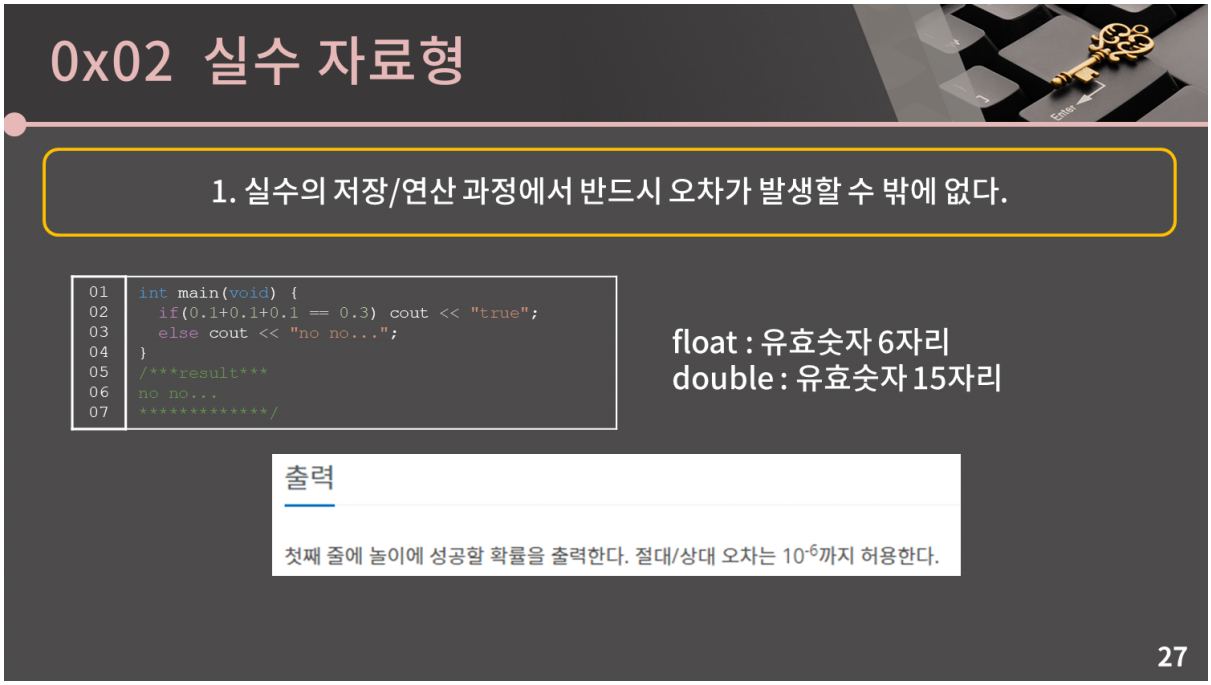
드디어 실수를 32칸 혹은 64칸에 저장하기 위한 모든 준비가 끝났습니다. 실수를 나타낼 때 칸은 sign, exponent, fraction field로 나누어집니다. sign field는 해당 수가 음수인지 양수인지 저장하는 필드이고, exponent field는 과학적 표기법에서의 지수를 저장하는 필드이고, fraction field는 유효숫자 부분을 저장하는 필드입니다. 각 필드의 크기가 float은 1, 8, 23 bit이고 double은 1, 11, 52 bit입니다.

-6.75로 예를 들어 설명드리면, 일단 -6.75는 $-1.1011_{(2)} \times 2^2$입니다. -6.75의 부호가 음수여서 sign은 1이고 또 2의 2승이 곱해지니 지수는 2인데, float에서는 여기에 127을 더한 129를 exponent field에 기록합니다. 참고로 129는 2진수로 10000001이니까 10000001이 기록됩니다. 왜 127을 더하냐면 이렇게 해줘야 음수 지수승도 exponent field 안에 잘 넣을 수 있기 때문입니다. double에서는 exponent field가 11칸이기 때문에 1023을 더합니다.

마지막으로 fraction field에는 현재 1.1011인데 여기서 맨 앞의 1은 뺀 1011이 적힙니다. 맨 왼쪽부터 2^{-1}, 2^{-2}자리인 셈이라 필드에 1011000…00이 적히게 됩니다. 그러니까 제일 왼쪽부터 채우는거죠.

이렇게 실수를 저장하는 방식을 IEEE-754 format이라고 부르니까 지금 제 설명이 잘 이해가 가지 않는다면 추가적으로 검색을 해보셔도 좋습니다. 도저히 이해가 안가면 지금 당장 이해를 못해도 상관없지만, 지금부터 설명할 실수의 성질들은 꼭 기억을 하셔야 합니다.

실수의 성질



첫 번째로, 실수의 저장과 연산 과정에서 반드시 오차가 발생할 수 밖에 없습니다. 아주 대표적인 예가 이 코드인데, 충격적이게도 0.1+0.1+0.1이 0.3과 다르다고 합니다.

이게 왜 그런거냐면 유효숫자가 들어가는 fraction field가 유한하기 때문에
2진수 기준으로 무한소수인걸 저장하려고 할 때에는 어쩔 수 없이
float은 앞 23 bit, double은 앞 52 bit까지만 잘라서 저장할 수 밖에 없습니다.

0.1은 이진수로 나타내면 무한소수여서 애초에 오차가 있는 채로 저장이 됐고 그걸 3번 더하다보니 오차가 더 커져서 위의 코드처럼 말도 안되는 일이 벌어졌습니다.

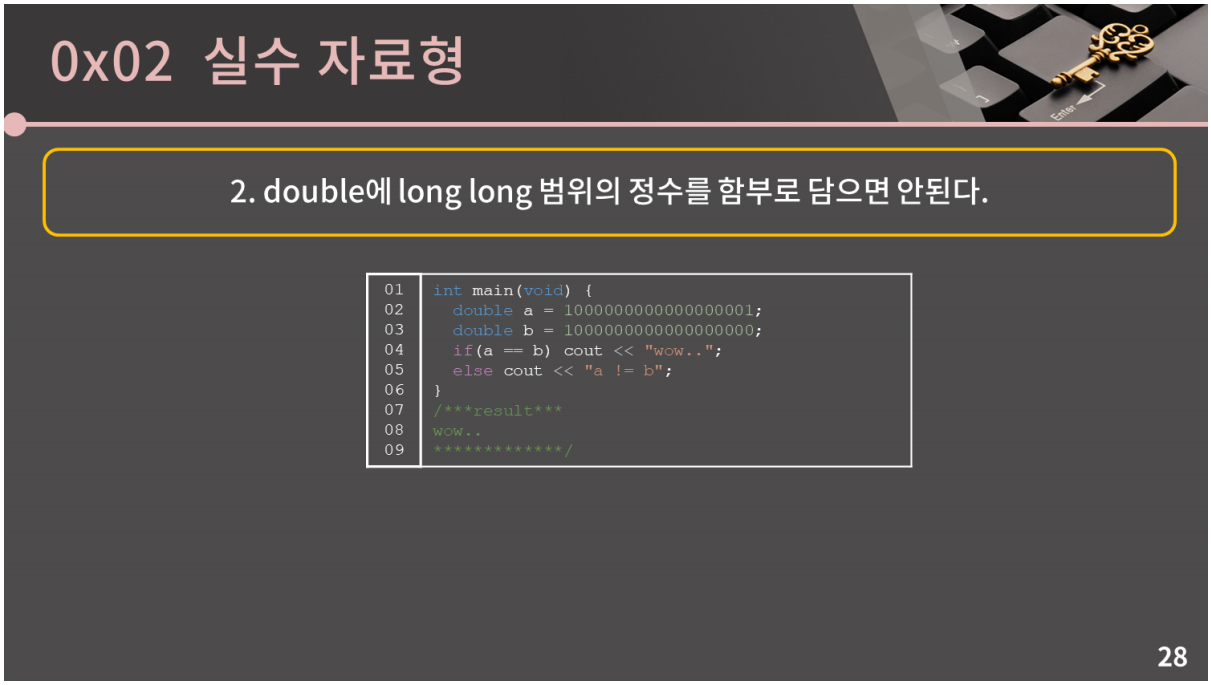
fraction field를 가지고 각 자료형이 어디까지 정확하게 표현할 수 있는지 보면 float은 유효숫자가 6자리이고 double은 유효숫자가 15자리입니다. 이 말은 곧 float은 상대 오차 10^-6까지 안전하고 double은 10^-15까지 안전하다는 소리입니다.

상대 오차가 10^-15까지 안전하다는 표현을 정확하게 이해할 필요가 있는데, 원래 참값이 1이라고 할 때, 1-10^-15 에서 1+10^-15 사이의 값을 가진다는게 보장된다는 의미입니다.
즉 오차가 생기는 것 자체는 막을 수가 없지만 오차가 어느 정도인지는 알 수 있습니다.

상대 오차의 허용 범위에서 볼 수 있듯 두 자료형끼리 차이가 굉장히 크기 때문에 **실수 자료형이 필요하면 꼭 float 대신 double을 써야합니다.** float이 메모리를 적게 쓴다는 장점이 있으니까 메모리가 정말
소중하면 필요할 수도 있긴 하지만 적어도 저는 지금까지 알고리즘 문제를 풀면서 double 대신 float을
써야 하는 상황을 경험해본 적이 없습니다.

또 실수 자료형은 필연적으로 오차가 있으니까 실수 자료형이 필요한 문제면 보통 문제에서 절대/상대 오차를 허용한다는 단서를 줍니다. 그런데 만약 이런 표현이 없다면 열

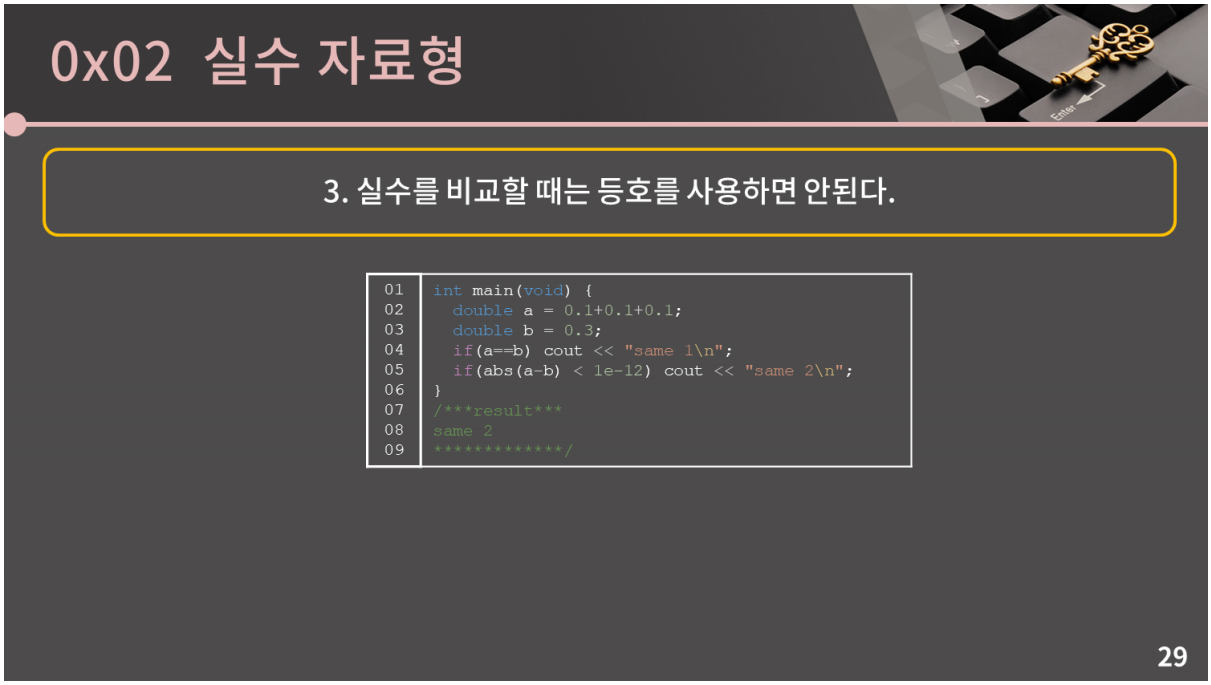
에 아홉은 실수를 안쓰고 모든 연산을
정수에서 해결할 수 있는 문제일 것입니다.



두 번째로, double에 long long 범위의 정수를 함부로 담으면 안됩니다. double은 유효숫자가 15자리인데 long long은 최대 19자리니까 1018+1과 1018을 구분할 수가 없고 그냥 같은 값이 저장됩니다. 즉, double에 long long 범위의 정수를 담을 경우 오차가 섞인 값이 저장될 수 있습니다.

다만 int는 최대 21억이기 때문에 double에 담아도 오차가 생기지 않습니다.

= 작은 곳에 큰 것을 넣을 수 없다



마지막으로 **실수를 비교할 때는 등호를 사용하면 안됩니다**. 이건 0.1+0.1+0.1과 0.3이 일치하지 않았던걸로 이미 보셨을텐데, 오차 때문에 두 실수가 같은지 알고 싶을 때에는 **둘의 차이가 아주 작은 값, 대략 10-12 이하면 동일하다고 처리를 하는게 안전합니다.**

5번째 줄의 1e-12가 처음 보는 표현일 수 있을 것 같은데 저게 바로 10-12입니다. 비슷하게 만약 109 가 필요하면 1000000000이라고 써도 되긴 한데 아무래도 이렇게 쓰면 0 갯수를 세는 것도 힘들고 하니까 대신에 1e9라고 써도 됩니다.

거듭 말하지만 sign field니 exponent field니 하는 IEEE-754
소수 표현법은 많이 헷갈리면 이해를 미뤄도 됩니다. 하지만 지금 이 실수 자료형의 3가지 성질은
꼭 알고 가셔야 나중에 "컴퓨터가 미쳤어!"라고 하며 샷건을 칠 일이 없어집니다.