
실전 알고리즘 0x01강

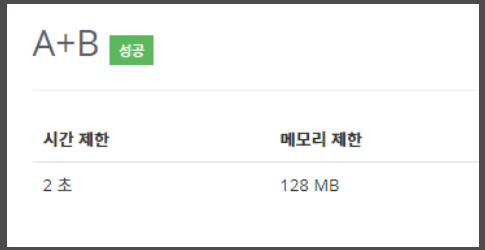
# 시간복잡도와 기초 코드 작성 요령

BaaaaaaaaaaaaaaaarkingDog

# 목차

# 0x00 시간, 공간복잡도


- 지난 강의에서 한 얘기 기억하나요?
- “코딩테스트는 PS, CP와 같이 주어진 문제를 정해진 시간 제한과 메모리 제한내로 해결할 수 있는 능력을 측정하는 테스트입니다.”
- 모든 문제에는 시간 제한과 메모리 제한이 있습니다.(설령 명시되어 있지 않더라도 상식적으로 생각해서 합리적인 수준의 제한이 있을 것입니다.)




- 문제에 대한 풀이를 생각한 이후 해당 풀이가 시간 제한/메모리 제한을 통과할 수 있는지를 파악할 수 있어야 합니다. 이를 위해 필요한 개념이 시간복잡도와 공간복잡도입니다.

# 0x00 시간, 공간복잡도


- 시간복잡도 = 입력의 크기와 문제를 해결하는데 걸리는 시간의 상관관계.
- 문제 : 대회장에 $N$명의 사람들이 일렬로 서있다. 거기서 당신은 이름이 '가나다' 인 사람을 찾기 위해 사람들에게 이름을 물어볼 것이다. 이름을 물어보고 대답을 듣는데까지 1초가 걸린다면 얼마만큼의 시간이 필요할까? (=시간복잡도가 얼마인가?)

# 0x00 시간, 공간복잡도


- 시간복잡도 = 입력의 크기와 문제를 해결하는데 걸리는 시간의 상관관계.
- 문제 : 대회장에 $N$명의 사람들이 일렬로 서있다. 거기서 당신은 이름이 '가나다' 인 사람을 찾기 위해 사람들에게 이름을 물어볼 것이다. 이름을 물어보고 대답을 듣는데까지 1초가 걸린다면 얼마만큼의 시간이 필요할까? (=시간복잡도가 얼마인가?)

- 답 : 앞에서부터 차례대로 물어보면 된다. 최악의 경우 $N$초, 최선의 경우 1초, 평균적으로 $N/2$초가 필요하다. 걸리는 시간은 $N$에 비례한다.

# 0x00 시간, 공간복잡도


- 시간복잡도 = 입력의 크기와 문제를 해결하는데 걸리는 시간의 상관관계.
- 문제 : 대회장에 $N$명의 사람들이 일렬로 서있다. 거기서 당신은 이름이 '가나다' 인 사람을 찾기 위해 사람들에게 이름을 물어볼 것이다. 이 때 사람들은 이름순으로 서있다. 이름을 물어보고 대답을 듣는데까지 1초가 걸린다면 얼마만큼의 시간이 필요할까? (=시간복잡도가 얼마인가?)

# 0x00 시간, 공간복잡도


- 시간복잡도 = 입력의 크기와 문제를 해결하는데 걸리는 시간의 상관관계.
- 문제 : 대회장에 $N$명의 사람들이 일렬로 서있다. 거기서 당신은 이름이 '가나다' 인 사람을 찾기 위해 사람들에게 이름을 물어볼 것이다. 이 때 사람들은 이름순으로 서있다. 이름을 물어보고 대답을 듣는데까지 1초가 걸린다면 얼마만큼의 시간이 필요할까? (=시간복잡도가 얼마인가?)

- 답 : 업다운게임을 하듯이 중간 사람에게 계속 물어보면 된다. 최악의 경우 $lg\,N$초, 최선의 경우 1초, 평균적으로 대략 $lg\,N$초가 필요하다. 걸리는 시간은 $lg\,N$에 비례한다.

# 0x00 시간, 공간복잡도


- 점근표기법(Big-O notation) = 함수의 결과값을 시간복잡도에서 가장 큰 지수승만 남겨서 나타내는 방법.
- $O(5N+7) = O(N)$, $O(3N^2+13N+4) = O(N^2)$, $O(2N^2lgN + N^2 + 3) = O(N^2lgN)$
- 엄밀한 정의는 수학의 극한을 이용하지만 넘어갑시다.
- 잘 와닿지 않더라도 앞으로 강의를 진행하면서 직접 예시를 통해 이 개념을 다시 설명할 일이 여러 번 있기 때문에 걱정하지 않으셔도 됩니다.
- 다시 앞의 문제들을 살펴보면…

# 0x00 시간, 공간복잡도


- 대회장에 $N$명의 사람들이 일렬로 서있다. 거기서 당신은 이름이 '가나다' 인 사람을 찾기 위해 사람들에게 이름을 물어볼 것이다. 이름을 물어보고 대답을 듣는데까지 1초가 걸린다면 얼마만큼의 시간이 필요할까? (=시간복잡도가 얼마인가?)

- $O(N)$

# 0x00 시간, 공간복잡도


- 대회장에 $N$명의 사람들이 일렬로 서있다. 거기서 당신은 이름이 '가나다' 인 사람을 찾기 위해 사람들에게 이름을 물어볼 것이다. 이 때 사람들은 이름순으로 서있다. 이름을 물어보고 대답을 듣는데까지 1초가 걸린다면 얼마만큼의 시간이 필요할까? (=시간복잡도가 얼마인가?)

- $O(lg\,N)$

# 0x00 시간, 공간복잡도


- 공간복잡도 = 입력의 크기와 문제를 해결하는데 걸리는 공간의 상관관계
- 예를 들어 입력의 크기 $N$에 대해 2차원 배열이 필요하면 $O(N^2)$ 의 공간복잡도, 따로 배열이 필요하지 않으면 O(1)의 공간복잡도입니다.
- 대부분의 경우 공간복잡도 문제보다는 시간복잡도로 인해 문제를 틀리게 되니 공간복잡도에 대해서는 크게 걱정을 안해도 됩니다.
- 공간복잡도라는 개념 보다는, 512MB가 `int` 변수를 대략 1.2억개 정도 담을 수 있다는 개념을 가지고 문제에 임하는 것이 좋겠습니다.

# 0x00 시간, 공간복잡도


- 아래에 주어진 코드에 대한 시간복잡도를 예측해보세요.

```
// arr[0]부터 arr[n-1]까지의 합을 구하는 함수
int func1(int* arr, int n){
  int tot = 0;
  for(int i = 0; i < n; i++) tot += arr[i];
  return tot;
}
// arr[0]부터 arr[n-1]까지의 최댓값을 구하는 함수
int func2(int* arr, int n){
  int mx = arr[0];
  for(int i = 0; i < n; i++){
    if(arr[i] > mx) mx = arr[i];
  }
  return mx;
}
```

```
// arr[0]부터 arr[n-1]까지에서
// 합이 k인 쌍의 갯수를 구하는 함수
int func3(int* arr, int n, int k){
  int cnt = 0;
  for(int i = 0; i < n; i++){
    for(int j = i+1; j < n; j++){
      if(arr[i]+arr[j] == k) cnt++;
    }
  }
  return cnt;
}
```

# 0x00 시간, 공간복잡도

- 답

```
// arr[0]부터 arr[n-1]까지의 합을 구하는 함수
int func1(int* arr, int n){  O(N)
  int tot = 0;
  for(int i = 0; i < n; i++) tot += arr[i];
  return tot;
}
// arr[0]부터 arr[n-1]까지의 최댓값을 구하는 함수
int func2(int* arr, int n){  O(N)
  int mx = arr[0];
  for(int i = 0; i < n; i++){
    if(arr[i] > mx) mx = arr[i];
  }
  return mx;
}
```

```
// arr[0]부터 arr[n-1]까지에서  O(N²)
// 합이 k인 쌍의 갯수를 구하는 함수
int func3(int* arr, int n, int k){
  int cnt = 0;
  for(int i = 0; i < n; i++){
    for(int j = i+1; j < n; j++){
      if(arr[i]+arr[j] == k) cnt++;
    }
  }
  return cnt;
}
```

# 0x00 시간, 공간복잡도


- 컴퓨터는 1초에 대략 1-3억개의 연산을 수행할 수 있습니다.(엄밀히 말해서 비트연산 혹은 비교와 같은 단순한 연산인지, 나눗셈/곱셈 등의 복잡한 연산인지에 따라 차이가 있지만 어림잡아 생각합시다.)
- 그렇기에 문제에서 주어진 Input의 범위를 보고 어느 정도의 시간복잡도까지 통과될 수 있는지를 잘 파악해야 합니다.

# 0x00 시간, 공간복잡도


- $N$의 크기에 따른 허용 시간복잡도는 대략 아래와 같습니다.(맹신하지는 마세요.)

| $N$의 크기 | 허용 시간복잡도 |
|---|---|
| $N \leq 11$ | $O(N!)$ |
| $N \leq 20$ | $O(2^N)$ |
| $N \leq 100$ | $O(N^4)$ |
| $N \leq 500$ | $O(N^3)$ |
| $N \leq 3{,}000$ | $O(N^2 lgN)$ |
| $N \leq 5{,}000$ | $O(N^2)$ |
| $N \leq 1{,}000{,}000$ | $O(NlgN)$ |
| $N \leq 10{,}000{,}000$ | $O(N)$ |
| 그 이상 | $O(lgN)$, $O(1)$ |

# 0x01 표준 입출력


- 코딩 테스트에서 입력과 출력은 표준 입출력을 사용합니다. 본인의 취향에 따라 `scanf/printf` 혹은 `cin/cout`을 사용하면 됩니다. (저는 `cin/cout`을 사용합니다.)
- `scanf/printf`와 `cin/cout`는 거의 동일한 기능을 수행하지만 단 한 가지 차이나는 것이 있는데, `cin/cout`는 char*와 C++ string 모두를 처리할 수 있는 반면 **`scanf/printf`로는 C++ string을 처리할 수 없습니다.**

```
int main(void){
  string s = "abcd";
  printf("s is %s", s);
}
```

result : `s is 0?`

# 0x01 표준 입출력


- C++ string이 char*보다 월등하게 편하기 때문에 **`scanf/printf`**를 사용하면서도 C++ string을 쓰고 싶은 상황이 있을 수 있는데, 아래와 같이 C++ string의 ***`c_str`*** 메소드를 사용하면 됩니다.

```
int main(void){
  char a[10];
  scanf("%s", a);
  string s(a); // 혹은 string s = a;
  printf("a is %s\n", a);
  printf("s is %s\n", s.c_str());
}
```

```
test
a is test
s is test
```

# 0x01 표준 입출력


- **scanf**의 **%s**, 그리고 **cin** 은 공백을 포함한 문자열을 제대로 입력받지 못합니다.

```
int main(void) {
  char a[20];
  scanf("%s", a);
  printf("%s", a);
}
```

result : `cc dd` / `cc`

```
int main(void) {
  string a;
  cin >> a;
  cout << a;
}
```

result : `aa bb` / `aa`

- C stream에서 공백을 포함한 문자열을 받으려면 C++11 이전 버전에서만 쓸 수 있는 **gets** 함수를 쓰거나, **fgets** 혹은 **scanf**의 **%[^\n]** 형식 지정자를 이용해야 하는데 맨 앞에 공백이 무시될 수 있다던가, 문자열 끝의 **\n**을 별도로 제거해야 한다던가 하는 불편함이 있어서 굉장히 껄끄럽습니다.

# 0x01 표준 입출력


- 반면 C++ stream에서는 `getline(cin, s)` 라는 명령 하나로 깔끔하게 처리가 가능하기 때문에 공백이 포함된 한 줄의 문자열을 입력받고 싶을 때는 그냥 C++ stream에서 입력받는 것을 추천드립니다.

```
int main(void) {
  string s;
  getline(cin, s);
  cout << s << '\n';
  const char* t = s.c_str(); // char* 로 문자열을 다루고 싶다면
  printf("%s", t);
}
```

result :

```
aa bb cc
aa bb cc
aa bb cc
```

# 0x01 표준 입출력


- `scanf/printf`에는 그다지 신경쓸 것이 없지만 `cin/cout`을 사용할 때에는 시간초과를 방지하기 위해서 반드시 아래의 두 명령을 실행시켜야 합니다.

```
int main(void){
  ios::sync_with_stdio(0);
  cin.tie(0);
```

- 해당 명령을 실행시키지 않은 경우, 입출력이 작을 때는 상관없지만 양이 많을 경우(예를 들어 출력 100만번, 입력 100만번) 입출력이 지연되어 시간 초과가 발생할 수 있습니다.

# 0x01 표준 입출력


- **`ios::sync_with_stdio(0)`** 는 C++ stream과 C stream 사이의 sync를 끄는 명령입니다. 아래와 같이 `cout`과 `printf`를 섞어서 사용했을 경우 기본적으로 출력 순서를 유지할 수 있도록 C++, C stream 사이의 sync를 유지하고 있으나,`cout`만 사용할 경우 sync를 유지할 필요가 없는데 의미없이 시간을 낭비하므로 sync를 꺼야 합니다. 반대로 말하면, **해당 명령을 실행한 이후에는 반드시 `printf`를 쓰지 말고 `cout`만 사용**해야 합니다.

```
int main(void){
//  ios::sync_with_stdio(0);
  printf("C stream 1\n");
  cout << "C++ stream 1\n";
  printf("C stream 2\n");
  cout << "C++ stream 2\n";
}
```

```
C stream 1
C++ stream 1
C stream 2
C++ stream 2
```

```
int main(void){
  ios::sync_with_stdio(0);
  printf("C stream 1\n");
  cout << "C++ stream 1\n";
  printf("C stream 2\n");
  cout << "C++ stream 2\n";
}
```

```
C stream 1
C stream 2
C++ stream 1
C++ stream 2
```

# 0x01 표준 입출력


- `cin.tie(0)` 는 `cin`과 `cout`이 번갈아 나올 때마다 flush 하지 않도록 하는 명령입니다. 기본적으로는 flush를 하도록 설정이 되어있고, flush가 되지 않으면 콘솔에 출력이 되지 않기 때문에 프로그램의 흐름과 콘솔에 보이는 결과가 다를 수 있지만, 채점 환경에서는 input buffer와 output buffer가 분리되어 있기 때문에 flush를 해줄 필요가 없습니다.
- 마찬가지 이유로 줄바꿈을 `endl`하는 경우가 있는데 `endl`은 줄을 바꾸고 flush를 하라는 명령이므로 줄바꿈은 `endl` 대신 `'\n'`으로 해야 합니다.

# 0x02 기초 지식 – 정수 자료형의 범위


- `char` 자료형이 1 byte라는 얘기를 들어보셨나요? 그 의미도 정확하게 알고 있나요?
- 1 byte = 8 bit, 즉 8개의 0 혹은 1이 들어가는 칸을 이용해 `char` 자료형의 값을 표현한다는 의미입니다.
- 10진수 표기에서 각 자리의 값이 $10^0$, $10^1$, $10^2$, … 를 의미하는 것 처럼 8 bit의 각 칸 또한 비슷하게 $2^0$, $2^1$, $2^2$, … 를 의미합니다. 그러나 `unsigned char`에서는 최상위 비트가 $2^7$ 이지만 `char`에서는 최상위 비트가 $2^7$이 아닌 $-2^7$ 을 의미한다는 점에 유의해야 합니다. 이는 연산을 편하게 처리하기 위함입니다.

# 0x02 기초 지식 – 정수 자료형의 범위


- `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 | → $-2^7+2^1+2^0 = -125$ |

- `char` 자료형의 범위 = 최소 $-2^7$(= -128) ~ $2^0+2^1 \cdots +2^6$ (= $2^7$ -1 = 127)
- `unsigned char` 자료형의 범위 = 최소 0 ~ $2^0+2^1 \cdots +2^7$ (= $2^8$ -1 = 255)

# 0x02 기초 지식 – 정수 자료형의 범위


- `short` 자료형은 2 byte, `int` 자료형은 4 byte, `long long` 자료형은 8 byte 이므로 `char`과 유사하게 계산해보면 각각 최대 $2^{15}$-1(=32767), $2^{31}$-1(≒$2.1\times10^{9}$) , $2^{63}$-1 (≒$9.2\times10^{18}$)까지 표현할 수 있습니다.
- 정수를 표현하기 위해서 보통 `int`, `long long` 자료형을 주로 사용합니다. 그다지 크지 않은 수를 저장할 경우에는 `int` 자료형을 사용하면 속도, 성능 모두 `long long` 보다 우수하나 80번째 피보나치 수를 구하는 문제와 같이 `int` 자료형이 표현할 수 있는 범위를 넘어서는 수를 저장해야 하면 반드시 `long long` 자료형을 사용해야 합니다.

# 0x02 기초 지식 – Integer overflow


- 이런 현상 겪어본 적 있으신가요?

```
int main(void){
  int a = 2'000'000'000 * 2;
  cout << a;
}
```

result : `-294967296`

- `int`가 최대 21억 정도까지 표현 가능하므로 의도대로 40억이 담기는건 불가능하겠지만 왜 오류가 나는게 아니라 뜬금없이 -2949672960이 나왔는지는 잘 모르겠는데…
- 이러한 현상을 integer overflow라고 합니다. 이 부분을 명확하게 이해하지 않고 그냥 각 자료형의 범위 안에서만 써야겠다고 생각하고 넘어가도 무방하지만, 원리를 알고 나면 코드의 오류를 조금 더 빠르게 찾을 수 있고 또 그다지 어려운 내용이 아니므로 짚고 넘어가겠습니다.

# 0x02 기초 지식 – Integer overflow


- Integer overflow의 원리 - 컴퓨터는 멍청해서 시키는대로 할 뿐이다!

| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

$2^3+2^0 = 9$

+

| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

$2^4+2^2+2^0 = 21$

=

| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|

$2^5+2^4+2^3+2^2+2^1 = 30$

# 0x02 기초 지식 – Integer overflow


- 음수의 덧셈도 그냥 이진수의 덧셈과 똑같이 생각하면 문제없이 잘 동작한다.

| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

$-2^7+2^3+2^0 = -119$

+

| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

$2^2+2^0 = 5$

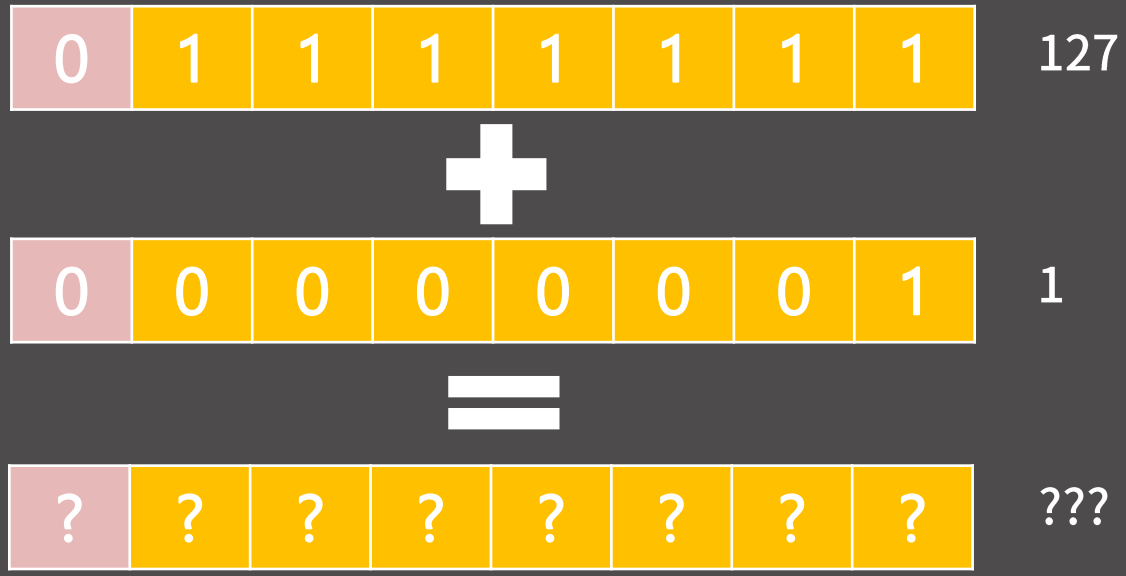
=

| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|

$-2^7+2^3+2^2+2^1 = -114$

# 0x02 기초 지식 – Integer overflow


- 그런데 127에 1을 더하면 어떤 일이 발생할까?



# 0x02 기초 지식 – Integer overflow


- 컴퓨터는 마찬가지로 2진수 연산을 한다. 이런 현상이 Integer overflow이다.

| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 127 |
|---|---|---|---|---|---|---|---|---|
| + | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| = | | | | | | | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | $-2^7 = -128$ |

# 0x02 기초 지식 – Integer overflow


- Integer overflow는 생각보다 빈번하게 일어나고, 또 찾기 힘든 버그입니다. 아래의 코드들에서 Integer overflow가 발생할 수 있는 함수들이 무엇인지, 어느 부분에서 발생하는지를 찾아보세요.

```
// 0부터 127까지의 수를 순회하는 함수
void func1(){
  for(char s = 0; s < 128; s++){
    // do something
  }
}
// 50!을 61로 나눈 나머지를 반환하는 함수
int func2(){
  int r = 1;
  for(int i = 1; i <= 50; i++){
    r = r * i % 61;
  }
  return r;
}
```

```
// 10**10을 1'000'000'007로 나눈
// 나머지를 반환하는 함수
int func3(){
  int a = 10;
  int mod = 1e9+7;
  for(int i = 0; i < 10; i++){
    a = 10 * a % mod;
  }
  return a;
}
```

# 0x02 기초 지식 – Integer overflow

- **func1** : **char** 자료형의 경우 127에서 1이 더해지면 -128이 되므로 의도한대로 동작하지 않고 무한루프에 빠짐
- **func2** : 정상적으로 동작
- **func3** : **int** 자료형의 경우 $10^9$에서 10이 곱해지면 integer overflow가 발생하므로 올바르지 않은 값을 반환

```
// 0부터 127까지의 수를 순회하는 함수
void func1(){
  for(char s = 0; s < 128; s++){
    // do something
  }
}
// 50!을 61로 나눈 나머지를 반환하는 함수
int func2(){
  int r = 1;
  for(int i = 1; i <= 50; i++){
    r = r * i % 61;
  }
  return r;
}
```

```
// 10**10을 1'000'000'007로 나눈
// 나머지를 반환하는 함수
int func3(){
  int a = 10;
  int mod = 1e9+7;
  for(int i = 0; i < 10; i++){
    a = 10 * a % mod;
  }
  return a;
}
```

# 0x02 기초 지식 – Integer overflow


- 버그를 어떻게 고칠 수 있을까?

```
// 0부터 127까지의 수를 순회하는 함수
void func1(){
  for(char s = 0; s < 128; s++){
    // do something
  }
}
// 50!을 61로 나눈 나머지를 반환하는 함수
int func2(){
  int r = 1;
  for(int i = 1; i <= 50; i++){
    r = r * i % 61;
  }
  return r;
}
```

```
// 10**10을 1'000'000'007로 나눈
// 나머지를 반환하는 함수
int func3(){
  int a = 10;
  int mod = 1e9+7;
  for(int i = 0; i < 10; i++){
    a = 10 * a % mod;
  }
  return a;
}
```

# 0x02 기초 지식 – Integer overflow

- 정답(func3은 2가지 해결 방법이 존재)

```
// 0부터 127까지의 수를 순회하는 함수
void func1(){
  for(char s = 0; s < 128; s++){
    // do something
  }
}
// 50!을 61로 나눈 나머지를 반환하는 함수
int func2(){
  int r = 1;
  for(int i = 1; i <= 50; i++){
    r = r * i % 61;
  }
  return r;
}
```

int

```
// 10**10을 1'000'000'007로 나눈
// 나머지를 반환하는 함수
int func3(){
  int a = 10;
  int mod = 1e9+7;
  for(int i = 0; i < 10; i++){
    a = 10 * a % mod;
  }
  return a;
}
```

long long

1011 혹은 (long long)10

# 0x02 기초 지식 – Integer overflow


- 이론적으로 Integer overflow를 익혔다고 하더라도 실제로 문제를 풀 때 이 실수를 분명 여러 번 저지르게 될 것입니다. 그렇지만 Integer overflow로 인해 여러 번 맞왜틀을 반복하다보면 이후엔 실수를 하지 않게 될 것입니다.
- 좋은 코딩 습관은 아니지만, 마음 편하게 아예 `int`를 쓰지 않고 모두 `long long`으로 대체하는 것도 하나의 방법입니다.
- `unsigned long long` 자료형을 넘는 수를 저장해야하면 string을 활용해야 합니다. 그 수로 연산을 해야하는 문제는 코딩테스트에서 나오지 않는게 정상이지만, 만약 그런 나쁜 문제가 나온다면 C++대신 Python으로 풉시다!
- 굳이 코딩테스트에서 C++로 `unsigned long long` 자료형을 넘는 수의 연산을 해야한다면 GCC의 `__int128` 자료형을 이용하거나 직접 큰 수의 연산 클래스를 만들어야 합니다.

# 0x02 기초 지식 – 실수 자료형


- 실수가 어떻게 자료형에 저장이 되는지 알고 있나요?
- 정수와 마찬가지로, 실수에서도 원리를 알고 있어야 어떨 때 실수를 사용해야 하고 어떨 때 실수를 사용하면 안되는지를 잘 구분할 수 있습니다.
- `float` 자료형은 4 byte(=32 bit), `double` 자료형은 8 byte(=64 bit) 입니다. 이제 0과 1이 들어갈 수 있는 이 32칸 혹은 64칸을 가지고 어떻게 실수를 표현하는지 알아봅시다.

# 0x02 기초 지식 – 실수 자료형


- 우선 2진수를 실수로 확장하는 것을 이해해야 합니다.
- $3 = 2^1+2^0$ 이고 이를 2진수로 나타내면 $11_{(2)}$ 입니다.
- 이를 실수로 확장하면 $3.75 = 2^1+2^0+2^{-1}+2^{-2}$ 이므로 $11.11_{(2)}$ 이라고 써도 되겠네요. 이와 같이 2의 음수 승을 이용해 임의의 실수를 2진수 상에서 표현할 수 있습니다.
- 한편, 10진수의 무한소수와 같이 2진수에서도 무한소수가 나타납니다.
- $1/3 = 2^{-2}+2^{-4}+2^{-6}+2^{-8}\cdots$ 이기 때문에 1/3을 2진수로 나타내면 $0.010101\cdots_{(2)}$입니다.

# 0x02 기초 지식 – 실수 자료형


- 그 다음엔 2진수에서의 과학적 표기법을 이해해야 합니다.
- 10진수에서 편의를 위해 $3561.234 = 3.561234 \times 10^3$ 으로 표현하는 것 처럼
- $11101.001_{(2)} = 1.1101001_{(2)} \times 2^4$ 로 표현할 수 있습니다.

- `float` 자료형 32 bit 는 sign(1 bit) / exponent(8 bit) / fraction field(23 bit)으로 구분됩니다. `double` 자료형 64 bit 는 sign(1 bit) / exponent(11 bit) / fraction field(52 bit)으로 구분됩니다.

# 0x02 기초 지식 – 실수 자료형


- -6.75 = $-1.1011_{(2)} \times 2^2$ 를 `float`에 저장하는 예를 들어보면 아래와 같습니다.


- 음수이기 때문에 sign은 1이고, exponent field에는 음의 지수승까지 원만하게 표현하기 위해 원래의 지수승인 2에 127을 더한 129를 8 bit 공간에 2진수로 적고, fraction field에는 11011을 적습니다.
- 지수승에 더해지는 127을 bias라 부르고, bias 덕분에 지수 부분에 $2^{-127}$부터 $2^{127}$까지 표현할 수 있습니다.
- `double`에 저장할 때에도 마찬가지 방법이나, exponent field가 11 bit이므로 bias가 1023입니다.

# 0x02 기초 지식 – 실수 자료형


- 제 설명이 부족해 이해가 잘 가지 않는다면 IEEE 754 format으로 검색을 해서 자료를 많이 찾을 수 있을 것입니다.
- 도저히 원리를 이해하기가 힘들면 나중으로 미루셔도 되지만 다음 장에서부터 설명할, 해당 원리로부터 나오는 성질들은 꼭 기억을 하셔야 합니다.
- 기본적으로 알고리즘 문제를 풀 때 메모리 문제로 인해 `double`을 쓰면 안되고 `float`을  써야만 하는 일은 지금까지 단 한번도 겪어보지 못했습니다. 그렇기에 모든 실수 관련 문제에 더 정확한 `double`을 쓰시면 됩니다.

# 0x02 기초 지식 – 실수 자료형


1. 실수의 저장/연산 과정에서 반드시 오차가 발생할 수 밖에 없다.

- fraction field는 유한한데 1/3과 같은 무한소수를 저장하려고 하면 앞의 일부분만 잘라서 저장을 하기 때문에 `float a`에 1/3을 저장하는 즉시 오차가 발생합니다.

```
int main(void){
  if(0.1+0.1+0.1 == 0.3)
    cout << "true";
  else
    cout << "no no..";
}
```

result : `no no..`

# 0x02 기초 지식 – 실수 자료형


1. 실수의 저장/연산 과정에서 반드시 오차가 발생할 수 밖에 없다.

- `float`는 $2^{-23}$(≒$1.19\times10^{-7}$), `double`은 $2^{-52}$ (≒ $2.22\times10^{-16}$)까지 정확하게 표현할 수 있기 때문에 보통 `float`은 대략 상대 오차 $10^{-6}$ 까지 안전하고 `double`은 $10^{-15}$ 까지 안전합니다.
- 문제에서 오차를 허용한다고 주어지는 경우에는 실수를 써도 되지만 그렇지 않다면 모든 연산을 정수에서 해결할 수 있도록 해야 실수오차로 인한 오답을 방지할 수 있습니다.

출력

첫째 줄에 놀이에 성공할 확률을 출력한다. 절대/상대 오차는 $10^{-6}$까지 허용한다.

# 0x02 기초 지식 – 실수 자료형


2. double에 long long 범위의 정수를 함부로 담으면 안된다.

- 앞서 말한 것 처럼 double은 상대오차 $10^{-15}$ 정도까지만 안전합니다. int 범위의 정수는 $10^{15}$보다 작으므로 괜찮지만 double에 long long 범위의 정수(예를 들어 $10^{18}+1$)을 담을 경우 오차가 섞인 값이 저장됩니다.
-

```
int main(void){
  double a = 1'000'000'000'000'000'001;
  cout << fixed; // double을 올바르게 출력하기 위해 지정하는 옵션
  cout << a;
}
```

result : 1000000000000000000.000000

# 0x02 기초 지식 – 실수 자료형


3. 실수를 비교할 때는 등호를 사용하면 안된다.

- 상대오차로 인해 수학적으로는 값이 일치하지만 실제 변수에 저장된 값은 다를 수가 있습니다. 그러므로 아래의 코드와 같이 두 수의 차이가 대략 $10^{-12}$ 아래라면 동일하다고 처리를 하는 것이 안전합니다.

```
int main(void){
  double a = 0.1+0.1+0.1;
  double b = 0.3;
  if(a==b) cout << "same 1\n";
  if(abs(a-b) < 1e-12) cout << "same 2\n";
}
```

result : `same 2`

# 0x02 기초 지식 – 실수 자료형


- 이외에도 실수의 뺄셈은 상대오차가 보장되지 않는다, 여러 개의 수를 더할 때 작은 수끼리 먼저 더하고 이후 큰 수를 더해야 한다는 주의사항 등이 있으나 너무 지엽적인 부분이기 때문에 넘어가겠습니다. 개인적으로 더 알아보고 싶다면 zlzmsrhak님이 작성하신 글(https://www.acmicpc.net/blog/view/37)을 참고하세요.

# 쉬어갑시다

- 이런 개념들을 처음 접해보면 굉장히 난해하고 어려울 것 같아요ㅠㅠ 오늘 강의에서 힘든 내용은 거의 다 끝났으니 조금만 더 힘을 내서 화이팅!

# 0x02 기초 지식 – 전역 변수와 지역 변수


- 변수의 지역성은 이미 C/C++ 문법을 공부하면서 익숙해졌을테니 굳이 여기서 설명하지는 않겠습니다.
- 일반적인 개발에서는 전역변수의 사용을 자제하고 포인터 혹은 참조자를 이용해 함수간 데이터를 넘겨주는 것을 권장하지만 코딩테스트에서는 빠르게 코딩하는 것이 관건이니 전역변수를 마음껏 써서 함수 인자를 적는데 시간을 낭비하지 않도록 합시다.
- 지역 변수는 0으로 초기화되지 않지만 전역 변수는 0으로 초기화됩니다.

```
int a[22];
int main(void){
  int b[22];
  cout << a[20] << ' ' << b[20];
}
```

result : 0 1618482848

# 0x02 기초 지식 – 채점 결과


- 코딩테스트에서는 공개된 예제 말고도 비공개된 다양한 예제들로 주어진 코드를 채점합니다.
- 하나의 TC(=Test Case)라도 틀리면 오답으로 처리되는게 일반적이나, 맞은 TC의 갯수만큼 부분점수를 주는 경우도 있습니다.
- 코딩테스트에 따라 채점 결과를 매번 알려주는 경우도 있고 그렇지 않은 경우도 있습니다.
- BOJ에서의 채점 결과에 대해 알려드리겠습니다.

# 0x02 기초 지식 – 채점 결과


- 맞았습니다!! : 모든 데이터에 대해 올바른 결과를 출력합니다.
- 출력 형식이 잘못되었습니다 : 답은 올바른데 공백이나 줄바꿈을 잘못 썼습니다.
- 틀렸습니다 : 특정 데이터에 대해 올바르지 않은 결과를 출력합니다.
- 시간 초과 : 프로그램이 제한된 시간 내에 종료되지 않았습니다. 초과되는 그 즉시 실행이 중단되기 때문에 정답이 맞았는지 틀렸는지는 알 수 없습니다.
- 메모리 초과 : 프로그램이 제한된 메모리보다 더 많이 사용했습니다. 마찬가지로 초과되는 그 즉시 실행이 중단되기 때문에 정답이 맞았는지 틀렸는지는 알 수 없습니다.
- 출력 초과 : 프로그램이 너무 많은 출력을 합니다. 보통 출력 부분에 무한 루프가 걸려서 발생하는 경우가 많습니다.

# 0x02 기초 지식 – 채점 결과


- 런타임 에러 : 프로그램이 정상적으로 실행되어 동작하던 도중 비정상적으로 종료되었습니다. main 함수가 0이 아닌 다른 값을 반환하는 경우, 0으로 나눈 경우, 배열에서 잘못된 인덱스 값에 접근한 경우 등의 다양한 이유로 발생할 수 있는 에러입니다.
- 컴파일 에러 : 컴파일에 실패했습니다. 클릭하면 컴파일 에러 메시지를 볼 수 있습니다.

출처 : BOJ 채점 도움말(*https://www.acmicpc.net/help/judge*)

# 0x03 코드 작성 요령


- 알고리즘 능력을 평가하는 코딩테스트는 남과 협업을 하는 프로젝트가 아닙니다. 코드가 한눈에 알아보기 쉽도록 예쁘게 짜기 위해 노력을 기울이는 것 보다는 내가 알아볼 수 있는 선에서 최대한 짧고 군더더기 없는 코드를 만들어내는 것이 중요합니다.
- 참고로 저는 아래와 같은 대회용 템플릿을 만들어놓고 씁니다.(절대 좋은 코딩 습관은 아닙니다ㅎㅎ..)

```
# pragma GCC optimize ("O3")
# pragma GCC optimize ("Ofast")
# pragma GCC optimize ("unroll-loops")

#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/rope>

using namespace std;
using namespace __gnu_pbds;
using namespace __gnu_cxx;

typedef unsigned int ui;
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int, int> pii;
typedef vector<int> vi;
typedef tuple<int, int, int> ti3;
typedef tuple<int, int, int, int> ti4;
typedef stack<int> si;
typedef queue<int> qi;
typedef priority_queue<int> pqi;
typedef pair<ll, ll> pll;
typedef vector<ll> vl;
typedef tuple<ll, ll, ll> tl3;
typedef tuple<ll, ll, ll, ll> tl4;
typedef stack<ll> sl;
typedef queue<ll> ql;
typedef priority_queue<ll> pql;
typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>ordered_set;
```

```
const int dx[4] = { 1,0,-1,0 };
const int dy[4] = { 0,1,0,-1 };
const int ddx[8] = { 0,0,1,1,1,-1,-1,-1 }, ddy[8] = { 1,-1,1,0
ll POW(ll a, ll b, ll MMM) { ll ret = 1; for (; b; b >>= 1, a
ll GCD(ll a, ll b) { return b ? GCD(b, a%b) : a; }
ll LCM(ll a, ll b) { if (a == 0 || b == 0)return a + b; return
ll INV(ll a, ll m) {
  ll m0 = m, y = 0, x = 1;
  if (m == 1) return 0;
  while (a > 1) {
    ll q = a / m;
    ll t = m;
    m = a % m, a = t;
    t = y;
    y = x - q * y;
    x = t;
  }
  if (x < 0) x += m0;
  return x;
}
pll EXGCD(ll a, ll b) {
  if (b == 0) return { 1,0 };
  auto t = EXGCD(b, a%b);
  return { t.second,t.first - t.second*(a / b) };
}
bool OOB(ll x, ll y, ll N, ll M) { return 0 > x || x >= N || 0
#define X first
#define Y second
#define rep(i,a,b) for(int i = a; i < b; i++)
#define pb push_back
```

```
#define pb push_back
#define all(x) (x).begin(), (x).end()
#define sz(a) ((int)(a.size()))
#define sf1(a) cin >> a
#define sf2(a,b) cin >> a >> b
#define sf3(a,b,c) cin >> a >> b >> c
#define sf4(a,b,c,d) cin >> a >> b >> c >> d
#define sf5(a,b,c,d,e) cin >> a >> b >> c >> d >> e
#define sf6(a,b,c,d,e,f) cin >> a >> b >> c >> d >> e >> f
#define pf1(a) cout << (a) << ' '
#define pf2(a,b) cout << (a) << ' ' << (b) << ' '
#define pf3(a,b,c) cout << (a) << ' ' << (b) << ' '<< (c) << ' '
#define pf4(a,b,c,d) cout << (a) << ' ' << (b) << ' '<< (c) << ' '<< (d) << ' '
#define pf5(a,b,c,d,e) cout << (a) << ' ' << (b) << ' '<< (c) << ' '<< (d) << ' '<< (e) << ' '
#define pf6(a,b,c,d,e,f) cout << (a) << ' ' << (b) << ' '<< (c) << ' '<< (d) << ' '<< (e) << ' ' << (f) << ' '
#define pf0l() cout << '\n';
#define pf1l(a) cout << (a) << '\n'
#define pf2l(a,b) cout << (a) << ' ' << (b) << '\n'
#define pf3l(a,b,c) cout << (a) << ' ' << (b) << ' '<< (c) << '\n'
#define pf4l(a,b,c,d) cout << (a) << ' ' << (b) << ' '<< (c) << ' '<< (d) << '\n'
#define pf5l(a,b,c,d,e) cout << (a) << ' ' << (b) << ' '<< (c) << ' '<< (d) << ' '<< (e) << '\n'
#define pf6l(a,b,c,d,e,f) cout << (a) << ' ' << (b) << ' '<< (c) << ' '<< (d) << ' '<< (e) << ' ' << (f) << '\n'
#define pfvec(V) for(auto const &t : V) pf1(t)
#define pfvecl(V) for(auto const &t : V) pf1(t); pf0l()
```

# 0x03 코드 작성 요령


- 지금 제 템플릿은 너무 과하지만, 미리 어느 정도 틀을 만들어두는게 도움이 되긴 합니다.
- 헤더는 `bits/stdc++.h`를 사용합시다. 해당 헤더 안에 모든 표준 라이브러리가 포함되어 있습니다. 단, MSVC에는 해당 헤더가 존재하지 않습니다. 직접 경로에 헤더파일을 추가해주면 MSVC에서도 `bits/stdc++.h` 하나로 깔끔하게 여러 라이브러리를 가져다 쓸 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
int main(void){
  ios::sync_with_stdio(0);
  cin.tie(0);


}
```

# 0x03 코드 작성 요령


- 문제를 풀고난 뒤 다른 사람의 코드를 보며 다른 사람은 어떻게 접근했는지 이해해보고 좋은 코딩 스킬들을 습득하는 것도 좋은 습관입니다.
- 단, 코드를 볼 때 너무 길이가 짧은 코드는 가독성을 포기하고 숏코딩을 한 코드일 가능성이 높으니 적당한 길이의 코드를 보세요.
- 아직 이론을 많이 모르고 구현력을 늘리는 것이 시급한 초보 단계에서는 한 문제를 잡고 너무 오랜 시간 고민하는 것 보다는 길어야 30분-1시간 정도 고민한 후에도 전혀 실마리가 보이지 않으면 빠르게 풀이를 참고하는 것이 좋습니다. 또 풀이를 봤는데도 이해가 가지 않으면 과감하게 다음으로 미루고 다른 문제로 넘어가는 것도 괜찮습니다.

# 강의 정리


- 시간/공간 복잡도가 무엇인지를 공부했습니다.
- `cin/cout, printf/scanf` 를 사용할 때 주의해야할 점에 대해 공부했습니다.
- 정수 자료형의 저장 방식과 Integer overflow에 대해 공부했습니다.
- 실수 자료형의 저장 방식과 실수 오차에 대해 공부했습니다.
- 전역 변수와 지역 변수에 대해 공부했습니다.
- 간단한 코드 작성 요령에 대해 공부했습니다.
- 특별한 알고리즘이 쓰이지 않고 입출력+식+조건문+반복문을 이용해 풀 수 있는 문제들로 구성된 문제집을 만들어두었으니 풀어봅시다.