이 예습자료에서는 이번 시간 수업에 필요한 몇 가지 주제를 차례대로 보도록 하겠습니다. 이 자료에서 활용하는 코드는 이번 시간 수업 자료 중 Prime.py에 있습니다.

<소수 구하기 – prime sieve(소수 체, 에라토스테네스의 체)>

주어진 정수 범위 내에서 소수를 찾는 방법인 prime sieve(소수 체, 에라토스테네스의 체)에 대해 알아보겠습니다.

먼저 소수(prime number, prime)는 **1보다 큰 자연수** 중 **약수(factor, divisor)가 1과 자기 자신뿐인** 수입니다. 다시 말하면 소수 p는 $1\times p$ 형태로만 인수 분해할 수 있습니다. 5는 {1, 5}만으로 나누어지므로 소수이며 1×5로만 인수분해 가능합니다. 6은 {1, 2, 3, 6}으로 나누어지므로 소수가 아니며 1×6, 2×3 등 여러 방법으로 인수분해 가능합니다. 1보다 큰 자연수 중 소수가 아닌 수는 합성수(composite number)라 합니다. 따라서 6도 합성수입니다.

컴퓨터 공학에서 소수가 활용되는 가장 대표적인 분야는 암호 시스템 개발입니다. 데이터를 암호화(encryption)하고, 복호화(decryption)하고, 디지털 서명(digital signature)하는데 소수가 사용됩니다. 또한 소수는 해시(hash) 함수에서 충돌(collision)을 줄이기 위해서도 사용됩니다. 이 외에도 소수는 데이터베이스와 파일 압축 등 다양한 분야에 사용되고 있습니다.

[Q] 소수(prime number)의 정의에 따르면 20보다 작은(<20) 소수는 몇 개인가?

6개
7개
8개
9개

※ 이 자료에서 [Q]로 제시된 문제는 학습 후 풀이하는 온라인 퀴즈에 그대로 나오니 학습하면서 그때그때 문제를 풀어 두세요. 온라인 퀴즈에서 보기의 순서는 바뀔 수 있으니 유의하세요. (예: 보기 1이 보기 2가 되고, 보기 2가 보기 1이 될 수 있음)

Prime sieve는 자연수 입력 N이 주어졌을 때 $\leq N$ 범위에 속하는 모든 소수를 찾는 데 사용되는 간단하면서 효율적인 방법입니다. 'Sieve'는 '시-브'라 발음하며 음식 등의 물건을 거르는 '체'를 의미합니다. 이 방법이 **소수가 아닌 수를 체로 걸러내는 과정**과 유사하므로 이러한 이름이 지어졌습니다. Prime sieve는 개발자의 이름을 따서 Sieve of Eratosthenes(에라토스테네스의 체)라고도 합니다. Prime sieve는 [표 1]처럼 동작합니다.

[표 1] Prime Sieve 알고리즘

자연수 입력 N이 주어졌을 때
  2~N 까지 정수로 구성된 리스트 L 준비. 즉 L={2, 3, 4, … , N}
 	
for i = 2 to $\lfloor \sqrt{N} \rfloor$
  if $i \in L$, # i 가 이때까지 L에 남아 있으면 i는 소수
  L에서 i의 배수를 모두 제거 # i의 배수는 소수가 아니므로 제거
 	
위 과정 후 L에 남아 있는 수가 $\leq N$ 범위의 소수

[표 1]에서 $\sqrt{N}$은 N의 제곱근을 의미합니다. ⌊x⌋은 floor 함수로 ⌊x⌋ = max{ n ∈ 정수 | n ≤ x }를 의미하며 x보다 작거나 같은 가장 큰 정수를 나타냅니다. 예를 들어 x = 4.1 이라면 ⌊x⌋ = 4 입니다. 즉 ⌊x⌋는 '내림' 함수로 보면 됩니다. 따라서 ⌊ $\sqrt{N}$ ⌋는 $\sqrt{N}$보다 작거나 같은 가장 큰 정수 또는 $\sqrt{N}$을 내림한 값입니다.

[표 1]의 방법은 {2, 3, 4, ... , N}에 포함된 합성수를 하나씩 제거하며 동작합니다. 즉 체로 합성수는 차례로 걸러내고 소수만 남기는 것처럼 동작합니다.

N=50인 경우의 예를 들어 [표 1]의 방법을 실행해 보겠습니다. 먼저 아래와 같이 2~50까지의 모든 수를 포함한 리스트를 생성합니다.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

[표 1]의 방법에서 for 루프는 가장 작은 소수인 i=2에서 시작합니다. 2의 배수는 소수가 아닌 합성수이므로 이들을 모두 리스트에서 제거합니다. 최종적으로 리스트에는 소수만 남기는 것이 목표이기 때문입니다. 제거한 수는 회색으로 표기하겠습니다.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

이제 for 루프는 i=3으로 넘어갑니다. 3은 리스트에 남아 있는데 이는 3이 소수라는 의미입니다. 3보다 작은 2의 배수를 이전 단계에서 모두 제거했음에도 3이 리스트에 남아 있다는 것은 3이 1과 자신 외에는 나누어지지 않는다는 뜻이기 때문입니다.

3이 소수임을 확인했으므로 3의 배수를 모두 리스트에서 제거합니다. 3의 배수는 3으로 나누어지므로 합성수이기 때문입니다.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

이제 for 루프는 i=4로 넘어가는데, 4는 리스트에서 이미 제거되었으므로(즉 합성수이므로) 다음 수인 i=5로 넘어갑니다. 5는 리스트에 남아 있는데 이는 5가 소수라는 의미입니다. 5보다 작은 2~4의 배수를 이전 단계에서 모두 제거했음에도 5가 리스트에 남아 있다는 것은 5가 1과 자신 외에는 나누어지지 않는다는 뜻이기 때문입니다.

5가 소수임을 확인했으므로 5의 배수를 모두 리스트에서 제거합니다. 5의 배수는 5로 나누어지므로 합성수이기 때문입니다.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

이제 for 루프는 i=6으로 넘어가는데, 6은 리스트에서 이미 제거되었으므로(즉 합성수이므로) 다음 수인 i=7로 넘어갑니다. 7은 리스트에 남아 있는데 이는 7이 소수라는 의미입니다. 7이 소수임을 확인했으므로 7의 배수를 모두 리스트에서 제거합니다.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

이제 for 루프는 $\lfloor \sqrt{50} \rfloor = 7$ 까지 진행했기 때문에 반복을 종료합니다. 최종적으로 리스트에 남은 숫자는 {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47}로 ≤50인 소수입니다.

그렇다면 ≤N 범위에 속하는 모든 소수를 찾기 위해 for 루프는 왜 i = N까지 진행하지 않고 i= $\sqrt{N}$까지만 진행할까요? for 루프가 하는 일은 n≤N 인 각 정수 n이 합성수인지 확인하여 합성수면 제거하는 것입니다. **n이 합성수이기 위해서는 n**보다 작은 소수 중 하나로 나누어떨어져야 합니다. 이를 확인하기 위해 for 루프가 하는 일은 가장 작은 소수 i = 2에서부터 시작하여 i = 2, 3, 5, 7, ... 차례로 진행하며 n을 i로 나누어 보는 것이라고 볼 수 있습니다. 이 과정에서 n이 나누어떨어지는 가장 작은 약수 i를 발견한다면 **$n = i \times q$로 인수분해** 되는 합성수라는 뜻이며, 그보다 더 큰 수로는 나누어 보지 않아도 됩니다. 여기서 i는 가장 작은 약수이므로 $i \leq q$ 이어야 함에 유의하세요. **이러한 가장 작은 약수 i는 $i \leq \sqrt{n}$을 만족해야 합니다** - 만약 반대로 i > $\sqrt{n}$ 이라면 $n = i \times q$ 이기 위해 q < $\sqrt{n}$ 이어야 하는데, 이렇게 되면 i가 q보다 더 작은 약수라는 조건에 ($i \leq q$) 모순되기 때문입니다. 따라서 n이 합성수임을 확인하려면 i = 2~ $\sqrt{n}$ 까지 수로만 나누어 보면 됩니다. 정리하면, prime sieve 알고리즘에서는 n = 2~N 까지 모든 수가 합성수인지를 확인해 보아야 하므로 i = 2~ $\sqrt{N}$ 까지 수로만 나누어떨어짐을 확인하면 됩니다.

아래 코드는 Python 언어로 구현한 prime sieve 알고리즘을 보여줍니다. 이 함수는 이번 시간 첨부파일에서도 볼 수 있습니다. **[표 1]**의 pseudo code와 비교하며 이해해 보세요.

```
00 def findPrimes(maxN):
01     prime = [True for _ in range(maxN+1)]
02     prime[0] = prime[1] = False
03     i = 2
04     while i*i <= maxN:
05         if prime[i]:
06             prime[i*i::i] = [False] * ((maxN - i*i) // i + 1)
07         i += 1
08
09     result = []
```

```
10      for i in range(len(prime)):
11          if prime[i]: result.append(i)
12
13      return result
```

위 함수는 입력으로 양의 정수 maxN을 받아 ≤maxN 조건을 만족하는 모든 소수를 리스트에 담아 반환합니다. 먼저 리스트 prime[0..maxN]을 생성하고 모든 원소를 True로 초기화하는데 (라인 01), 알고리즘 수행 후에 prime[i] = True라면 i가 소수임을 나타내고, False라면 i가 소수가 아님을 나타냅니다. 0과 1은 소수가 아니므로 prime[0]과 prime[1]은 False로 설정하고 (라인 02), i = 2부터 시작해 1씩 증가시켜 가며 prime sieve 알고리즘에 따라 i가 소수인지 확인해 갑니다 (라인 03~07). i가 소수라면 (라인 05) i의 배수를 모두 False로 표기합니다 (라인 06). 라인 06에서 등호(=) 왼쪽의 prime[i*i::i]는 'prime[i*i]에서 시작해서 리스트의 마지막 원소까지 i 간격으로' 라는 뜻입니다. 등호(=) 오른쪽의 [False] * ((maxN - i*i) // i + 1)은 등호 왼쪽에 대응되는 모든 원소에 False를 저장하겠다는 뜻으로, ((maxN - i*i) // i + 1)이 그러한 원소의 갯수를 나타냅니다. 이러한 Python 문법이 익숙하지 않다면 'list slicing (슬라이싱)'을 검색해 보세요. 이렇게 i가 $\sqrt{\max N}$일 때까지 진행하면 (라인 04) maxN 이하의 모든 소수를 분류해 낼 수 있습니다. 소수와 합성수의 분류가 끝났다면 지금까지 찾은 모든 소수를 리스트 result에 담아 (라인 09~11) 반환합니다 (라인 13).

위 코드의 prime sieve를 구현 방식은 [표 1]의 수도코드를 보다 효율적으로 구현한 것입니다. [표 1]처럼 모든 숫자를 담은 리스트 L을 준비한 후 합성수를 하나씩 제거하는 방법을 사용하면 **원소를 하나 제거할 때마다** 제거한 원소 뒤의 모든 원소를 한 칸씩 앞으로 옮겨 담아야 해서 평균적으로 **리스트의 길이에 비례한 시간**이 걸립니다. 하지만 위 코드처럼 고정된 길이의 리스트 prime[]을 두고 True와 False 값 중 적절한 값을 저장하는 방식으로 구현하면 원소를 하나 제거하는데 상수 시간만이 걸립니다.

[Q] [표 1]에서 prime sieve 알고리즘을 보았다. 이 알고리즘의 입력 N = 100이라면 for 루프에서 사용하는 변수 i 값이 가지게 되는 최댓값은?

1
10
20
100

Prime sieve 알고리즘은 기존 시간에 배운 것처럼 나눗셈보다는 곱셈을 사용한 방법이라고도 볼 수 있습니다. 즉 i = 2 ~ N까지의 각 수를 그보다 작은 수로 나누어서 소수임을 판명하는 대신 반대로 소수 2, 3, 5, 7, ... 에서 시작해 그들의 배수를 합성수로 보고 제거하는 방식이기 때문입니다.

<정렬된 값에 대한 검색 – 이진 탐색(binary search)>

리스트 형태로 저장된 값이 **정렬되어 있을 때** 원하는 값 I를 찾는 가장 기본적인 방법은 I를 찾을 때까지 리스트의 값을 하나씩 순차적으로 탐색(sequential search)하는 것입니다. 순차 탐색 방법은 리스트의 크기 N이 작을 때는 잘 동작하지만, N이 커질수록 비용이 많이 들며 느려지기 시작합니다. 예를 들어 $N = 10^{10}$개의 값이 리스트에 저장되어 있고 I가 리스트의 3/4 위치에 있다고 하겠습니다. 탐색하기 전에는 원하는 값이 어디에 있는지 모르므로 임의의 지점을 탐색 시작점으로 정해야 하는데, 리스트의 첫 번째 값을 탐색 시작점으로 정했다고 가정합시다. 이렇게 첫 번째 값부터 순차 검색을 해간다면 I를 찾을 때까지 총 $3 / 4 \times 10^{10}$ 개의 값을 탐색해야 하므로 상당히 많은 시간이 걸립니다.

순차 탐색보다 효율적인 방법은 이진 탐색(binary search) 입니다. 이진 탐색 방법은 먼저 리스트 전체를 탐색 범위로 설정한 후 매 단계 진행할 때마다 탐색하는 범위를 1/2배로 줄여나가며, 최종적으로 원하는 값 I 하나만이 탐색 범위에 남으며 탐색이 종료합니다. 따라서 리스트의 크기가 N일 때는 $\log_2 N$ 단계를 진행한 후에 탐색을 종료합니다. 이진 탐색은 아래 [표 2]와 같이 동작합니다.

[표 2] 이진 탐색 알고리즘 (원하는 값 I와 같은 값 검색)

```
30  L: N 개의 오름차순으로 정렬된 값을 가진 리스트 {i0, i1, ... , iN-1}
31  I: L에서 찾고자 하는 값
32
33  // BinarySearch() 함수를 호출하여 L에서 I를 찾고 반환된 I의 index 출력
34  print BinarySearch(0, N-1, I);
35
36  // BinarySearch() 함수 정의
37  function BinarySearch(int from, int to, int I)  // 탐색 범위: ifrom~ito
38  {
39   if (from>to) return -1;          // I 가 L에 없는 경우 -1 반환
40
41   int mid = (from + to) / 2;    // 탐색 범위에서 가운데 값의 index
42   if (imid==I) return mid;        // 가운데 값이 I와 같다면 그 index 반환
43   else if (imid<I)                          // (imid<I) 이면
44      return BinarySearch(mid+1, to, I);  // 탐색 범위를 imid+1~ito로 조정
45   else                                        // (I<imid) 이면
46      return BinarySearch(from, mid-1, I);  // 탐색 범위를 ifrom~imid-1로 조정
47  }
```

[표 2]의 알고리즘은 리스트에 저장된 값이 숫자라고 가정합니다 (실제로 컴퓨터 시스템의 모든 정보는 숫자로 대응되어 저장됩니다). 이진 탐색 방법은 특히 이러한 값들이 **정렬되어 있을 때** 적용할 수 있으며, [표 2]의 방법은 오름차순으로 정렬되어 있다고 가정합니다.

이진탐색 함수 BinarySearch()는 리스트 L에서 원하는 값 I를 찾아 그 값의 리스트 내에서의 index를 반환합니다. 매개변수 from과 to는 리스트 내에서 index로 I를 찾는 탐색 범위가 $i_{from} \sim i_{to}$ (from번째 값 ~ to번째 값) 임을 의미합니다.

$N = 10$인 경우의 예를 들어 [표 2]의 방법을 실행해 보겠습니다. 아래와 같이 오름차순으로 정렬된 리스트가 주어졌으며, 이로부터 $I = 25$를 찾는다고 하겠습니다.

index	0	1	2	3	4	5	6	7	8	9
item	1	5	7	11	12	20	25	30	32	36

(단계 #1) 처음에는 탐색 범위가 리스트 전체인 i_0 ~ i_9 으로 설정됩니다. 이들의 가운데 값인 i_4 = 12가 검색 대상인 I = 25 와 같은지 비교합니다. 그 결과 i_4 < I 이며 리스트는 오름차순으로 정렬되어 있으므로 (I가 리스트에 존재한다면) i_4의 오른쪽에 I가 있을 것입니다. 따라서 다음 단계에서의 탐색 범위는 현재 탐색 범위 중에서 i_4의 오른쪽인 i_5 ~ i_9 가 됩니다. i_4의 왼쪽인 i_0 ~ i_3에는 I가 존재하지 않으므로 탐색 범위에서 제외되며, 제외된 범위는 회색으로 표기하였습니다.

index	0	1	2	3	4	5	6	7	8	9
item	1	5	7	11	12	20	25	30	32	36

(단계 #2) 새로운 탐색 범위 i_5 ~ i_9에서 가운데 값은 i_7 = 30이므로 이 값을 검색 대상 I = 25와 비교합니다. 그 결과 I < i_7 이므로 i_7의 왼쪽에 I가 있을 것입니다. 따라서 다음 단계에서의 탐색 범위는 현재 탐색 범위 i_5 ~ i_9 중에서 i_7의 왼쪽인 i_5 ~ i_6 이 됩니다.

index	0	1	2	3	4	5	6	7	8	9
item	1	5	7	11	12	20	25	30	32	36

(단계 #3) 새로운 탐색 범위 i_5 ~ i_6에서 가운데 값은 i_5 = 20 이므로 이 값을 검색 대상 I = 25와 비교합니다. 그 결과 i_5 < I 이므로 i_5의 오른쪽에 I가 있을 것입니다. 따라서 다음 단계에서의 탐색 범위는 현재 탐색 범위 i_5 ~ i_6 중에서 i_5의 오른쪽인 i_6 ~ i_6 이 됩니다.

index	0	1	2	3	4	5	6	7	8	9
item	1	5	7	11	12	20	25	30	32	36

(단계 #4) 새로운 탐색 범위 i_6 ~ i_6에서 가운데 값은 i_6 = 25 이므로 이 값을 검색 대상 I = 25와 비교합니다. 이번에는 i_6 == I 이므로 이 값의 index인 6을 반환하며 종료합니다.

지금까지 4단계에 걸쳐 정숫값을 비교하며 I의 index를 찾았습니다. 만약 이진탐색을 사용하지 않고 i_0 에서 시작하여 순차적으로 탐색하였다면 i_0 ~ i_6 까지 7단계의 비교가 필요했을 것입니다.

이제부터는 **리스트에서 찾고자 하는 값 I가 리스트에 없는 경우**에 대해 생각해 보겠습니다. 지금까지 본 것과 같은 리스트에서 검색하되, 다른 값인 I = 27을 찾는다고 가정하겠습니다. 이러면 단계 #1~3까지는 앞에서 본 I = 25를 찾는 경우와 같게 진행되며, 단계 #4만이 달라집니다.

(단계 #4, I = 27을 찾는 경우) 탐색 범위 i_6 ~ i_6에서 가운데 값은 i_6 = 25 이므로 이 값을 검색 대상 I = 27과 비교합니다. 이번에는 i_6 < I 이므로 i_6의 오른쪽에 I가 있을 것이라고 보고 탐색 범위를 변경합니다. 특히 [표 2]의 라인 #43~44와 같이 다음 단계에서의 탐색 범위를 i_7 ~ i_6 으로 설정합니다. 하지만 이처럼 from > to (7 > 6) 인 탐색 범위는 라인 #39와 같이 유효하지 않은 범위로 판단되며 I가 리스트에 존재하지 않음을 의미하는 -1을 반환하게 됩니다.

이처럼 원하는 값을 찾지 못하는 경우 -1을 반환하는 대신에 I에 가까운 다른 값의 index를 반환하도록 할 수도 있습니다. 예를 들어 **I보다 작거나 같은 가장 큰 값**의 index를 반환하도록(즉 $\leq I$ 인 가장 큰 값 반환) [표 2]를 수정하면 아래의 [표 3]과 같습니다. 표 2의 라인 #39만 표 3의 라인 #59와 같이 수정하였습니다. 즉 "if (from>to) return -1" → "if (from>to) return to" 로 변경하였습니다.

[표 3] 이진 탐색 알고리즘 (원하는 값 I보다 작거나 같은 가장 큰 값 검색)

```
50  L: N 개의 오름차순으로 정렬된 값을 가진 리스트 {i0, i1, ... , iN-1}
51  I: L에서 찾고자 하는 값
52
53  // BinarySearch() 함수를 호출하여 L에서 I를 찾고 반환된 I의 index 출력
54  print BinarySearch(0, N-1, I);
55
56  // BinarySearch() 함수 정의
57  function BinarySearch(int from, int to, int I)  // 탐색 범위: ifrom~ito
58  {
59   if (from>to) return to;  // I 가 없는 경우 ≤I 인 가장 큰 값 index 반환
60
61   int mid = (from + to) / 2;   // 탐색 범위에서 가운데 값 검사
62   if (imid==I) return mid;       // 가운데 값이 I와 같다면 그 index 반환
63   else if (imid<I)                    // (imid<I) 이면
64       return BinarySearch(mid+1, to, I);  // 탐색 범위를 imid+1~ito로 조정
65   else                                    // (I<imid) 이면
66       return BinarySearch(from, mid-1, I);  // 탐색 범위를 ifrom~imid-1로 조정
67  }
```

[표 3]의 방법을 두 가지 경우에 대해 실행해 보겠습니다.

먼저 지금까지 본 것과 같은 리스트에서 I = 27**보다 작거나 같은 가장 큰 값**을 찾는다고 가정하겠습니다. 단계 #1~3까지는 앞에서 본 경우와 같게 진행되며, 단계 #4만이 달라집니다.

index	0	1	2	3	4	5	6	7	8	9
item	1	5	7	11	12	20	25	30	32	36

(단계 #4, I = 27을 찾는 경우) 탐색 범위 i_6 ~ i_6에서 가운데 값은 i_6 = 25 이므로 이 값을 검색 대상 I = 27과 비교합니다. $i_6 < I$ 이므로 i_6의 오른쪽에 I가 있을 것이라고 보고 다음 단계에서의 탐색 범위를 i_7 ~ i_6 으로 설정합니다. 이 범위는 라인 #59의 from > to (7 > 6) 조건을 만족하게 되고 따라서 to = 6이 반환됩니다. 이는 I = 27보다 작거나 같은 가장 큰 값은 i_6=25 임을 의미합니다.

다음으로 같은 리스트에서 I = 23**보다 작거나 같은 가장 큰 값**을 찾는다고 가정하겠습니다. 이번에도 단계 #1~3까지는 앞에서와 같게 진행되며, 단계 #4만이 달라집니다.

(단계 #4, I = 23을 찾는 경우) 탐색 범위 i_6 ~ i_6에서 가운데 값은 i_6 = 25 이므로 이 값을 검색 대상 I = 23과 비교합니다. $I < i_6$ 이므로 I가 i_6의 왼쪽에 있을 것이라고 보고 탐색 범위를 변경합니다. 특히 [표 3]의 라인 #66과 같이 다음 단계에서의 탐색 범위를 i_6 ~ i_5 으로 설정합니다. 이 범위는 라인 #59의 from > to (6 > 5) 조건을 만족하게 되고 따라서 to=5가 반환됩니다. 이는 I = 23보다 작거나 같은 가장 큰 값은 i_5 = 20 임을 의미합니다.

지금까지는 ***I*보다 작거나 같은 가장 큰 값**을 반환하도록(즉 $\leq I$ 인 가장 큰 값 반환) 이진 탐색 방법을 수정해 보았습니다. 정리하면 이진 탐색을 사용하면 from과 to 값이 점차 서로 가까워지며 거리를 좁혀가게 되는데, 만약 찾고자 하는 원소 *I*가 존재하지 않는다면 두 값은 결국 서로를 지나쳐 to < from 상태가 됩니다. 이때 to에 있는 원소는 찾고자 하는 원소 *I*보다 작은 값 중 가장 큰 값이며, from에 있는 원소는 *I*보다 큰 값 중 가장 작은 값입니다. 이에 따르면 [표 3]의 조건을 바꾸어 ***I*보다 크거나 같은 가장 작은 값**을 반환하도록(즉 $\geq I$ 인 가장 작은 값 반환) 하는 것도 크게 다르지 않습니다. [표 3]의 방법을 어떻게 수정해야 할지 생각해 보세요.

[Q] 아래 표와 같이 오름차순으로 정렬된 10개의 값이 리스트에 저장되어 있다. 이진 탐색을 사용하여 30을 찾는다면 몇 개의 원소와 비교를 진행한 후에 탐색을 종료하는가?

index	0	1	2	3	4	5	6	7	8	9
item	1	5	7	11	12	20	25	30	32	36

1개
2개
3개
4개

[Q] 아래 표와 같이 오름차순으로 정렬된 10개의 값이 리스트에 저장되어 있으며, 여기서 30을 찾고자 한다. i_0에서 시작하여 순차적으로 탐색한다면 몇 개의 값과 비교를 진행한 후에 탐색을 종료하는가?

index	0	1	2	3	4	5	6	7	8	9
item	1	5	7	11	12	20	25	30	32	36

6개
7개
8개
9개

[Q] 원소 *I*보다 크거나 같은 가장 작은 값($\geq$I인 가장 작은 값)을 찾고자 한다. [표 2]의 라인 #39 "if (from>to)" 조건에서 어떤 값을 반환해야 하는가?

return -1;
return to;
return from;
return to-1;

아래 코드는 Python 언어로 구현한 이진 탐색 알고리즘을 보여줍니다. 이 함수는 이번 시간 첨부파일에서도 볼 수 있습니다. [표 2]의 pseudo code와 비교하며 이해해 보세요.

```
70 def binarySearchEQ(numbers, target):
71     def recur(fromIndex, toIndex):
72         if fromIndex > toIndex: return -1
73 
74         mid = int((fromIndex+toIndex)/2)
75         if numbers[mid] < target: return recur(mid+1, toIndex)
76         elif numbers[mid] > target: return recur(fromIndex, mid-1)
77         else: return mid
78 
79     return recur(0, len(numbers)-1)
```

위 함수 binarySearchEQ()는 입력으로 숫자의 리스트 numbers와 찾고자 하는 값 target을 받아 리스트 내에서 target의 index를 반환합니다. 이 코드가 [표 2]의 수도코드와 다른 점은 크게 두 가지입니다. 첫 번째는 binarySearchEQ()함수 내부적으로 재귀호출에 사용하는 recur() 함수를 정의해 사용했습니다 (라인 71~77). 이 함수는 이진 탐색에서 현재 탐색하는 범위인 fromIndex와 toIndex를 입력으로 받습니다. 이러한 함수를 내부적으로 정의하고 재귀호출함으로써 이진 탐색을 잘 모르는 사용자도 binarySearchEQ() 함수를 편리하게 호출할 수 있습니다. 즉 탐색 대상인 numbers와 target만 입력으로 전달하면 되고 이진 탐색에 필요한 from/to index의 초깃값은 설정해 줄 필요가 없습니다. 위 코드가 [표 2]의 수도코드와 다른 두 번째 차이점은 현재 탐색하는 가운데 원소와 찾고자 하는 원소 간의 대소 비교를 하는 순서에 있습니다. 즉 위 코드는 <, >, == 순서로 비교를 하며 (라인 75~77), [표 2]는 ==, <, > 순서로 대소 비교를 합니다 (라인 42~46). 이진 탐색 과정을 보면 두 원소가 다른 경우가 더 많으므로 <와 >를 먼저 확인하는 편이 더 빠르고, 따라서 라인 75~77과 같이 구현하였습니다. 정리하면 Python 코드는 사용자 편의성과 속도를 향상시키기 위해 [표 2]의 수도코드를 약간 변형하였습니다.

<Python 함수에서 임의 갯수의 매개변수를 입력으로 받는 방법>

Python 함수는 임의 갯수의 매개변수를 입력으로 받도록 구현할 수 있습니다. 임의 갯수의 매개변수를 받는다는 것은 같은 함수 function을 function(1)로 호출할 수도, function(1,2)로 호출할 수도 있다는 뜻입니다. 이렇게 임의 갯수의 매개변수가 필요한 예를 들면 최댓값을 구하는 경우를 들 수 있습니다. 여러 숫자 중 최댓값을 반환하는 함수가 필요하되, 숫자의 갯수가 변할 수 있다면 이러한 방식으로 함수를 구현하면 사용하기 편리할 것입니다.

이렇게 매개 변수의 갯수가 변동 가능하도록 하려면 매개 변수 이름 앞에 **별표(*)**를 붙이면 됩니다. 아래 코드에 그러한 예가 있습니다. 매개변수 이름 args 앞에 *를 붙였음을 유의해 보세요.

```
80 def variableLengthArguments(*args):
81     print(f"args: {args}")
82
83     print(f"each item: ", end='')
84     for arg in args: print(arg, end=' ')
85     print()
86
87     print(f"max(args): {max(args)}")
```

이렇게 *를 사용하게 되면, * 뒤에 지정한 매개 변수는 임의 갯수의 매개 변수를 tuple로 저장하는 변수가 됩니다. 예를 들어 위 함수에 다섯 개의 매개 변수를 주어

variableLengthArguments(1,2,3,4,5)

와 같이 호출하면

```
args: (1, 2, 3, 4, 5)
each item: 1 2 3 4 5
max(args): 5
```

와 같은 출력을 볼 수 있습니다. 먼저 매개변수 args를 출력한 결과 (라인 81) 다섯 개의 매개변수가 5-tuple (1,2,3,4,5)로 저장되었음을 볼 수 있습니다. 이러한 tuple에 저장된 원소는 for loop을 사용해 하나씩 사용할 수도 있습니다 (라인 84). 또한 리스트나 tuple에 적용할 수 있는 함수(예: max, min, sorted 등)도 그대로 사용할 수 있습니다 (라인 87).

[Q] variableLengthArguments(300,100,200,500)을 실행했을 때 출력 마지막 라인에서 볼 수 있는 값은?

```
max(args): 300
max(args): 100
max(args): 200
max(args): 500
```