〈자료구조 실습〉 - 알고리즘 분석

※ 입출력에 대한 안내

- 특별한 언급이 없으면 문제의 조건에 맞지 않는 입력은 입력되지 않는다고 가정하라.
- 특별한 언급이 없으면, 각 줄의 맨 앞과 맨 뒤에는 공백을 출력하지 않는다.
- 출력 예시에서 □는 각 줄의 맨 앞과 맨 뒤에 출력되는 공백을 의미한다.
- 입출력 예시에서 ↦ 이 후는 각 입력과 출력에 대한 설명이다.

[문제 1] 나머지 연산

'%'(modulo) 연산자는 나눗셈의 나머지를 반환한다. 덧셈과 뺄셈 연산자만을 사용하여 a를 b로 나눈 나머지를 구하는 프로그램을 작성하시오. 단, a ≥ 0, b > 0 인 정수다.

힌트: modulo(a, b) 함수는 O(a/b) 시간에 실행하도록 작성할 수 있다.

입력 예시 1	출력 예시 1
`14 3` ↦ a=14, b=3	`2` ↦ 14 % 3

입력 예시 2	출력 예시 2
`3 3` ↦ a=3, b=3	`0` ↦ 3 % 3

입력 예시 3	출력 예시 3
`0 4` ↦ a=0, b=4	`0` ↦ 0 % 4

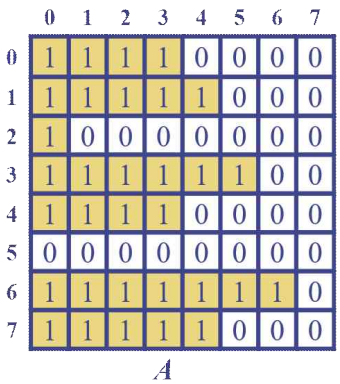
프로그램은 다음 함수를 작성하여 사용하시오.

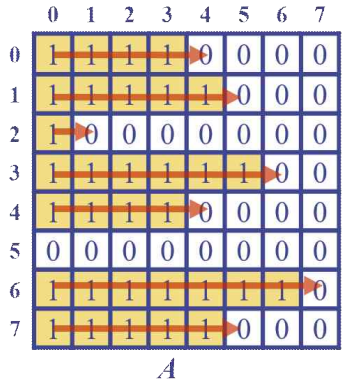
- modulo( ) 함수
  - 인자: 정수 a, b
  - 반환값: a % b 의 결과

[문제 2] 비트행렬에서 최대 1행 찾기

n × n 비트 행렬 A의 각 행은 1과 0으로만 구성되며, A의 어느 행에서나 1들은 해당 행의 0들보다 앞서 나온다고 가정하자. 행렬 A를 입력받아, 가장 많은 1을 포함하는 행을 찾는 프로그램을 작성하시오. 그러한 행이 여러 개 있을 경우 그 가운데 가장 작은 행 번호를 찾아야 한다.

- **예:** 8 × 8 비트 행렬 A : 6행이 가장 많은 1을 포함한다(아래 왼쪽 그림 참고).





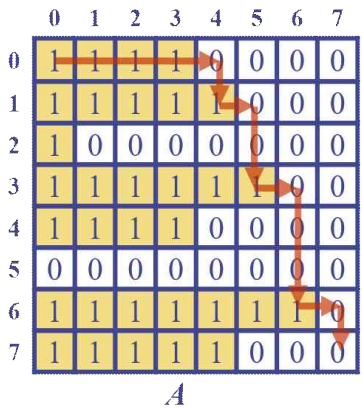
◦ 참고로, 아래 의사코드 함수 mostOnesSlowVersion은 1이 가장 많은 행을 찾기는 하지만, 실행시간이 O(n)이 아니라 $O(n^2)$, 즉 2차 시간(quadratic time)이다(위 오른쪽 그림 참고).

```
Alg mostOnes(A, n)              {slow version}
   input bit matrix A[n × n]
   output the row of A with most 1's

1. row ← jmax ← 0
2. for i ← 0 to n - 1
      j ← 0
      while ((j < n) & (A[i, j] = 1))
         j ← j + 1
      if (j > jmax)
         row ← i
         jmax ← j
3. return row                   {Total O(n^2)}
```

◦ 위 함수보다 빠른 해결은 다음과 같다.

1. 행렬의 좌상 셀에서 출발한다
2. 0이 발견될 때까지 행렬을 가로질러 간다
3. 1이 발견될 때까지 행렬을 내려간다
4. 마지막 행 또는 열을 만날 때까지 위 2, 3 단계를 반복한다.
5. 1을 가장 많이 가진 행은 가로지른 마지막 행이다



◦ 빠른 버전 해결은 최대 2n 회의 비교를 수행하므로,
명백히 O(n)-시간, 즉 선형 시간(linear time) 알고리즘이다
(오른쪽 그림 참고).

◦ 위에서 설명한 **선형 시간에 실행하는 함수 mostOnes(A, n)를 작성**하여 사용하시오.
- 인자: 비트 행렬 A, 정수 n ≤ 100 (A의 크기)
- 반환값: 정수 (최대 1 행 번호)

입출력 형식:

1) **입력:** main 함수는 다음 값들을 표준입력 받는다.
- 첫 번째 라인: 정수 n (n × n 행렬에서 n 값, 단 n ≤ 100으로 전제함))
- 두 번째 이후 라인: n × n 비트 행렬 원소들 (행우선 순서)

2) **출력:** main 함수는 1이 가장 많은 행 번호를 출력한다. 단, 첫 번째 행 번호는 0이다.

입력 예시 1

```
8                   ↦ n = 8  (8 × 8 행렬)
1 1 1 1 0 0 0 0     ↦ 각 비트는 공백으로 구분
1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 0
1 1 1 1 1 1 0 0
1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0
1 1 1 1 1 0 0 0
```

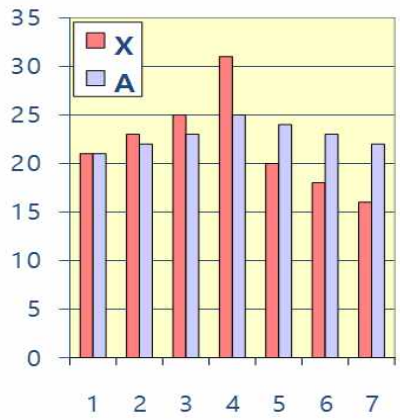
출력 예시 2

```
6      ↦ 1이 가장 많은 행 번호: 6
```

[문제 3] 원시 데이터값들로 구성된 배열 X의 i-번째 누적평균(prefix average)이란 X의 i-번째에 이르기까지의 (i + 1)개 원소들의 평균이다. 즉,

A[i] = (X[0] + X[1] + … + X[i])/(i + 1)

누적평균은 경제, 통계 분야에서 오르내림 변동을 순화시킴으로써 대략적 추세를 얻어내기 위해 사용된다. 일례로 부동산, 주식, 펀드 등에도 자주 활용된다.. 이 문제는 배열 X의 누적평균(prefix average) 배열 A를 구하는 프로그램을 구현하고 테스트하는데 관한 것이다.



◦ 아래 의사코드 함수 **prefixAverages1**은 위 정의를 있는 그대로 이용하여 누적평균값들을 2차 시간에 구한다.

참고: 각 명령문 오른 편 중괄호 내의 수식은 실행 시간 분석을 위한 근거로서, 해당 명령문이 수행하는 치환, 반환, 산술 및 비교 연산 등 기본 명령들의 수행 횟수를 나타낸다.

```
Alg prefixAverages1(X, n)      {ver.1}
   input array X of n integers
   output array A of prefix averages of X

1. for i ← 0 to n - 1          {n}
      sum ← 0                  {n}
      for j ← 0 to i           {1 + 2 +…+ n}
         sum ← sum + X[j]      {1 + 2 +…+ n}
      A[i] ← sum / (i + 1)     {n}
2. return A                    {1}              {Total O(n^2)}
```

◦ 위의 의사코드 함수 prefixAverages1의 내용을 살펴보면, i 번째 외부 반복에서는, 바로 전 i – 1 번째 반복에서 구했던 [0 ~ i - 1]의 합에, i + 1 번째 원소 값 한 개만을 더해 현재 합을 얻어 평균을 구한다. 따라서 이를 수정하여 이전의 i – 1번째까지의 합을 보관하여 다음 반복으로 전달하는 방식으로 반복한다면 현재 합을 구하는데 필요한 시간을 단축할 수 있다는 것을 알 수 있다. 이렇게 중간 합을 보관하는 방식으로 알고리즘을 개선한 함수 **prefixAverage2**는 누적평균값들을 선형 시간에 구할 수 있게 된다.

문제 3-1> 함수 prefixAverages1과 prefixAverages2, 그리고 이들을 테스트할 수 있는 main 함수를 구현하여 아래 테스트를 수행하라.

입출력 형식:

1) main 함수는 아래 형식의 표준입력을 사용하여 배열 X를 초기화한 후 각 함수를 호출한다.

입력 : main 함수는 다음 값들을 표준입력 받는다.

첫 번째 라인: 정수 n (배열 X의 크기)

두 번째 이후 라인: X[0] X[1] X[2] ... (배열 X, 한 라인 상의 양의 정수 수열)

(도움말: n의 크기에는 제한이 없다. 따라서 동적할당을 사용하여야 함)

2) main 함수는 아래 형식의 표준출력을 사용하여 각 함수로부터 반환된 배열 A를 출력한다.

출력 : A[0] A[1] A[2] ...

(배열 X와 같은 크기의 배열 A의 원소들을 나타내는 한 라인 상의 양의 정수 수열로서 첫 번째 라인은 prefixAverages1의 출력을, 두 번째 라인은 prefixAverages2의 출력을 나타낸다)

3) 평균 계산 시 소수점 이하를 반올림하여 정수로 구한다. 정확한 반올림을 위해, %.f를 쓰지 말고 int 성질을 이용해서 반올림하라.

입력 예시 1

```
3                 ↦ 배열 X 크기
5 1 9             ↦ 배열 X
```

출력 예시 1

```
5 3 5             ↦ prefixAverages1의 출력
5 3 5             ↦ prefixAverages2의 출력
```

입력 예시 2

```
6                 ↦ 배열 X 크기
1 3 2 10 6 8      ↦ 배열 X
```

출력 예시 2

```
1 2 2 4 4 5       ↦ prefixAverages1의 출력
1 2 2 4 4 5       ↦ prefixAverages2의 출력
```

문제 3-2> 위 main 함수를 수정하여 아래 절차로 두 함수 prefixAverage1과 prefixAverage2 각각의 실행시간을 측정 비교하라.

주의:

1) main 함수는 배열 X의 크기 n을 표준입력 받는다.
2) main 함수는 난수발생 함수를 사용하여 크기 n의 1~10,000 사이의 정수 배열 X를 초기화한 후 각 함수를 한 번씩 호출한다.
3) main 함수는 각 함수로부터 반환 직후 해당 함수의 실행시간 데이터를 표준출력한다. (배열 초기화 시간을 포함하지 않도록 주의)
4) 예를 들어 실행시간 = 0.000 ms가 되지 않도록, 그리고 두 함수의 성능 비교가 가능하도록 소수점 정밀도를 높여야 한다. 사용 컴퓨터의 성능 문제 때문에 피치 못할 경우에만 아래 입력 예시와는 다른 배열 크기 값들을 사용하여 실행시간 데이터를 얻어도 좋다.

입력 예시 1	출력 예시 1
100000 ↦ 배열 X 크기	0.421289721ms ↦ prefixAverages1의 cpu time 0.054142322ms ↦ prefixAverages2의 cpu time

입력 예시 2	출력 예시 1
200000 ↦ 배열 X 크기	0.852323142ms ↦ prefixAverages1의 cpu time 0.054142322ms ↦ prefixAverages2의 cpu time

입력 예시 1	출력 예시 1
300000 ↦ 배열 X 크기	0.421289721ms ↦ prefixAverages1의 cpu time 0.054142322ms ↦ prefixAverages2의 cpu time

- 함수 prefixAverages1, prefixAverages2
  - prefixAverage1(X, n): 느린 버전
  - prefixAverage2(X, n): 빠른 버전
  - 인자: 정수 배열 X (원시 데이터값들), 정수 n (배열 X의 크기)
  - 반환값: 정수 배열 A (누적평균값들)

※ 참고사항: 문제 3-2의 실습 목적

- 본 문제는 실행시간을 측정하여 비교분석하는 것이 목적인 실습으로, 출력결과는 컴퓨터마다 실행할 때 마다 다르게 나올 수 있다.
- 코딩평가시스템(oj 시스템)으로 채점되지 않는 문제로, 출력 결과(실행 시간)를 통해 두 함수의 실행 시간이 차이가 나는지, X가 증가함에 따라 실행 시간이 어떤 비율로 증가하는 지를 확인해보자.

1,2, 3-1 은 OJ에서 처리하고 3-2는 별도로 작성하여 sucomclass2@gmail.com

제목 : 2주실습 학번 이름 을 지켜

소스코드를 보내고, 실행화면을 캡쳐하여 (위의 입력 3번 실행 결과) 첨부하고, 분석내용을 써볼 것.