〈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.

※ 참고: 이번 주의 주요 실습 내용은 알고리즘의 성능을 비교하는 [문제 **3-2**]이다. 아래 **A**, **B**는 [문제 **3-2**]를 해결하기 위해 필요한 두 가지 참고 사항이다.

A. 난수발생 함수

○ 프로그램에 따라서는 사용자 개입없이 한 개 또는 여러 개의 난수를 공급받아야 제대로 작동하는 경우가 있다. 아주 간단한 예로 사람과 번갈아 가며 주사위를 던지는 프로그램의 경우를 들 수 있다. 컴퓨터가 주사위를 던질 차례에서 사람이 대신 던져줄 수는 없다. 사용자 개입없이 컴퓨터 스스로 무작위 수를 생성해야만 하는 것이다. 아래는 이런 상황에서 사용할 난수발생 함수에 대한 도움말이다.

○ 난수란 주사위 눈수처럼 특정한 나열 순서나 규칙이 없이 생성된 무작위 수를 말하며 영어로는 random number라고 한다. 그리고 난수발생 함수란 난수를 자동생성시켜 공급해주는 함수를 말한다.

○ C 언어에서는 시스템 라이브러리를 통해 난수발생 함수를 제공한다. 함수 **rand**()가 **0** ~ **RAND_MAX** 범위의 무작위 정수를 반환한다. 여기서 **RAND_MAX**는 **rand**()가 반환할 수 있는 최대수며 이 값은 시스템마다 다를 수 있다.

○ 난수발생 함수를 사용하기 위해서는 헤더파일 **stdlib.h**가 필요하다. 즉, 코드 상단에 **#include <stdlib.h>**를 써줘야 **rand**()를 사용할 수 있다. 이 헤더파일에 **rand**()의 원형이 포함되어 있으며 **rand**()가 발생시킬 수 있는 최대수인 **RAND_MAX**도 정의되어 있다.

○ 사용 예 **1**

```
#include<stdio.h>
#include<stdlib.h>

int main(void) {
    printf("%d₩n", rand());
    return 0;
}
```

○ 하지만 위 코드를 여러 번 실행시켜 보면 계속 같은 난수가 나오는 것을 볼 수 있다.

무작위 수의 연속이 아니라 같은 수의 연속이며, 이는 주사위를 아무리 던져도 같은 수가 나오는 상황이라 제대로 된 난수라고 할 수도 없다.

- 이 문제를 해결하기 위해, 즉 매번 다른 난수를 발생시키기 위해 시드(seed)값을 설정하는 방법이 있다. 시드 값을 달리 하면 함수 **rand**()에서 발생시키는 난수가 매번 달라진다.

- 시드 값을 설정하기 위해 사용하는 함수가 바로 **srand**()이다. **srand**()는 이를 호출할 때 전달하는 인자를 기반으로 하여 난수를 초기화시키는 역할을 한다.

- **srand**()의 인자로써 함수 **time**()을 권장한다. **time**()은 인자로 **NULL**을 전달하면 **1970**년 **1**월 **1**일 **0**시 (**UTC** 타임존) 이후 현재까지 흐른 초 수를 반환한다. 시간은 멈추지 않고 계속해서 흐르므로 **time**() 함수가 반환한 현재의 초 수를 인자로 하여 **srand**()를 호출하면 난수 기준 값이 (무작위라 할 수 있는) 현재 초 수로 초기화되는 것이다. 따라서 **srand**(**time**(**NULL**))과 같이 호출하면 된다. 잊지말 것은, **time**()을 사용하기 위해서는 상단에 **#include <time.h>**를 추가해야 하는 것이다. 정리하면, 최종적인 코드는 아래와 같다.

- 사용 예 **2**

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int main(void) {
    srand(time(NULL));
    printf("%d₩n", rand());
    return 0;
}
```

- Tip

rand() % n

- 0 ~ (n – 1) 범위의 난수를 생성한다.
- **예 1:** 0 ~ 1 사이의 정수 난수를 발생시키고 싶다면, rand() % 2로 호출.
- **예 2:** 0 ~ 10000 사이의 정수 난수를 발생시키고 싶다면, rand() % 10001로 호출.

rand() % n + m

- m ~ (n – 1 + m) 범위의 난수를 생성한다.
- **예 1:** 1 ~ 6 사이의 정수 난수(**예:** 주사위 눈수)를 발생시키고 싶다면, rand() % 6 + 1로 호출.
- **예 2:** 1000 ~ 9999 사이의 정수 난수를 발생시키고 싶다면, rand() % 9000 + 1000으로 호출.

((((long) rand() << 15) | rand()) % 1000000) + 1

- 1 ~ 1,000,000 범위의 난수를 생성한다.
- **RAND_MAX** = **32,767**인 경우 이 수보다 큰 범위의 난수를 발생시키기 위한 방법: **32,767**의 16진수 표현은 0X7FFF, 2진수 표현은 0111 1111 1111 1111이므로 **rand()** 호출은 15비트로 표현할 수 있는 난수를 반환한다. 이 값을 왼쪽으로 15비트만큼 shift해준 뒤 두번째 **rand()** 호출로부터 얻은 난수와 **OR** 연산을 해주면 30비트로 표현할 수 있는 난수를 얻을 수 있다. 이 난수의 범위는 0 ~ 1,073,741,823이며 이 값을 1,000,000으로 나머지 연산을 한 뒤 1을 더해주면 1 ~ 1,000,000 범위의 난수를 얻게 된다.

B. 시간측정 함수

- 알고리즘의 실행시간을 측정하는 데는 점근적 분석 방법과 실제 시간 측정, 두 가지가 있다.

- 점근적 분석에 의한 시간 측정은 big-Oh 값을 구하는 이론적 방식을 말하며, 실제 시간 측정은 알고리즘 실행에 소요되는 cputime을 측정하는 것을 말한다.

- 점근적 방식에 의한 측정은 <자료구조 및 실습> 교재 **1**장에서 배운대로 이론적 방식에 의해 측정할 수 있으며, 실제 시간 측정은 라이브러리 함수를 이용하여 어떤 알고리즘 실행에 소요되는 실제 cputime을 측정한다. 아래는 라이브러리 함수를 이용한 실제 시간 측정에 관한 도움말이다.

- 라이브러리 함수 가운데 일반적인 시간측정 함수인 **clock**()을 사용하면 시간이 정밀하게 나오지 않는 문제가 발생한다. 대안으로 **QueryPerformanceCounter**() 함수를 사용하면 정밀한 시간을 출력할 수 있다. 구체적인 사용 방법은 다음과 같다.
  - 헤더파일로 **windows.h**를 추가한 후,
  - **LARGE_INTEGER** 변수 선언하고,
  - **QueryPerformanceFrequency**() 함수를 통해 타이머의 주파수를 변수에 저장한 후,
  - 시간을 측정하고 싶은 작업의 전후에 **QueryPerformanceCounter**()를 호출하고 그 반환값들을 이용하여 계산, 출력하면 된다.

- 사용 예

```
#include <stdio.h>
#include <Windows.h>

int main(void){
    LARGE_INTEGER ticksPerSec;
    LARGE_INTEGER start, end, diff;

    QueryPerformanceFrequency(&ticksPerSec);
    QueryPerformanceCounter(&start);
    // 시간을 측정하고 싶은 작업(예: 함수 호출)을 이곳에 삽입
    QueryPerformanceCounter(&end);

    // 측정값으로부터 실행시간 계산
    diff.QuadPart = end.QuadPart - start.QuadPart;
    printf("time: %.12f sec₩n₩n", ((double)diff.QuadPart/(double)ticksPerSec.QuadPart));
    return 0;
}
```

- 작동 원리: 메인보드에 고해상도의 타이머가 존재하는데 이를 이용하여 특정 실행 시점들의 CPU 클럭수들을 얻어온 후 그 차이를 이용하여 작업 시간을 구한다. **clock**() 함수와 달리 **1us** 이하의 시간까지 측정한다.

  QueryPerformanceFrequency() : 타이머의 주파수(초당 진동수)를 얻는 함수

  QueryPerformanceCounter() : 타이머의 CPU 클럭수를 얻는 함수

  작업 전후의 클럭수 차를 주파수로 나누면 작업 시간(초, **sec**)을 구할 수 있고, **ms**단위로 출력하기 위해선 결과 값에 **1,000**을 곱해주면 된다.

- **clock**() 함수와 비교: **clock**()은 초당 **1,000**번의 측정을 통해 **1ms**의 시간을 측정할 수 있는 데 비해, **QueryPerformanceCounter**()는 초당 **10,000,000**번의 측정으로 **0.1us**의 시간까지 측정할 수 있다. 초당 클럭수는 **time.h**를 헤더로 추가한 후 **CLOCKS_PER_SEC**을 출력하여 알 수 있고, 타이머의 주파수는 **QueryPerformanceFrequency**()를 통해 알 수 있다.

[문제 1] 나머지 연산

'%'(modulo) 연산자는 나눗셈의 나머지를 반환한다. 덧셈과 뺄셈 연산자만을 사용하여 **a**를 **b**로 나눈 나머지를 반환하는 **modulo**(a, b) 함수와 이를 테스트할 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

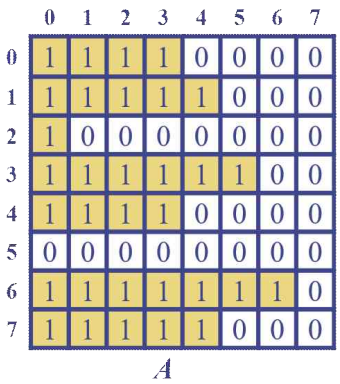
입력 예시 3	출력 예시 3
0 4 ↦ a=0, b=4	0 ↦ 0 % 4

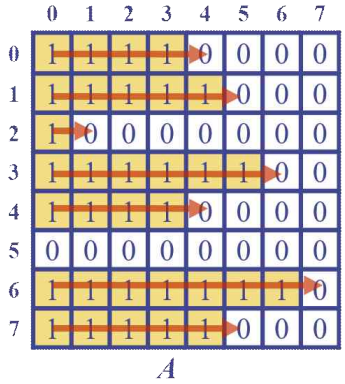
필요 함수:

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

[문제 2] 비트행렬에서 최대 1행 찾기

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





한다.

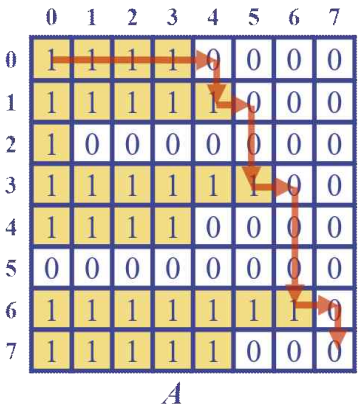
- **예: 8** × **8** 비트 행렬 **A** : **6**행이 가장 많은 **1**을 포함한다(아래 왼쪽 그림 참고).

- 참고로, 아래 의사코드 함수 **mostOnesSlowVersion**은 **1**이 가장 많은 행을 찾기는 하지만, 실행시간이 **O**(**n**)이 아니라 **O**(n^2), 즉 **2**차 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(오른쪽 그림 참고).
  1. 행렬의 좌상 셀에서 출발한다.
  2. **0**이 발견될 때까지 행렬을 가로질러 간다.
  3. **1**이 발견될 때까지 행렬을 내려간다.
  4. 마지막 행 또는 열을 만날 때까지 위 **2**, **3** 단계를 반복한다.
  5. **1**을 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.

입출력 형식:

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

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

입력 예시 1	출력 예시 2
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	6 ↦ 1이 가장 많은 행 번호: 6

필요 함수:

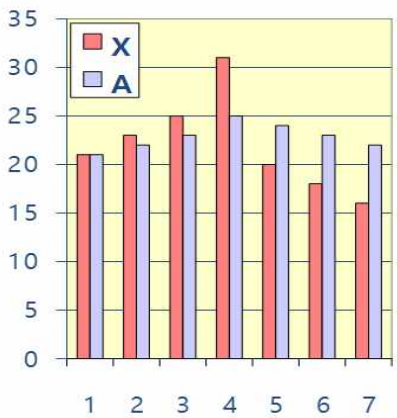
- **mostOnes**(**A**, **n**) 함수
  - 인자: 비트 행렬 **A**, 정수 **n** ≤ **100** (**A**의 크기)
  - 반환값: 정수 (최대 **1** 행 번호)
  - 시간 성능: **O**(**n**)

[문제 3] 누적 평균

원시 데이터값들로 구성된 배열 **X**의 **i**-번째 **누적평균**(prefix average)이란 **X**의 **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 **X**의 누적평균(prefix average) 배열 **A**를 구하는 프로그램을 구현하고 테스트하는데 관한 것이다.



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

※ **참고:** 각 명령문 오른 편 중괄호 내의 수식은 실행시간 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 **prefixAverages1**의 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** – **1**번째까지의 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 **prefixAverages1**과 **prefixAverages2**, 그리고 이들을 테스트할 수 있는 **main** 함수를 구현하여 아래 테스트를 수행하라.

입출력 형식:

1) **main** 함수는 아래 형식의 표준입력을 사용하여 배열 **X**를 초기화한 후 각 함수를 호출한다.

입력 : **main** 함수는 다음 값들을 표준입력 받는다.

첫 번째 라인: 정수 **n** (배열 **X**의 크기)

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

※ **힌트: n**의 크기에는 제한이 없다. 따라서 동적 할당을 사용하여야 함)

2) **main** 함수는 아래 형식의 표준출력을 사용하여 각 함수로부터 반환된 배열 **A**를 출력한다.

출력 : **A**[**0**] **A**[**1**] **A**[**2**] ...

(배열 **X**와 같은 크기의 배열 **A**의 원소들을 나타내는 한 라인 상의 양의 정수 수열로서 첫 번째 라인은 **prefixAverages1**의 출력을, 두 번째 라인은 **prefixAverages2**의 출력을 나타낸다)

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

입력 예시 1		출력 예시 1	
3 5 1 9	↦ 배열 X 크기 ↦ 배열 X	5 3 5 5 3 5	↦ prefixAverages1의 출력 ↦ prefixAverages2의 출력

입력 예시 2		출력 예시 2	
6 1 3 2 10 6 8	↦ 배열 X 크기 ↦ 배열 X	1 2 2 4 4 5 1 2 2 4 4 5	↦ prefixAverages1의 출력 ↦ prefixAverages2의 출력

문제 3-2> 위 **main** 함수를 수정하여 아래 절차로 두 함수 **prefixAverage1**과 **prefixAverage2** 각각의 실행시간을 측정 비교하라.

※ 주의:

1) **main** 함수는 배열 **X**의 크기 **n**을 표준입력받는다.
2) **main** 함수는 **난수발생 함수**를 사용하여 크기 **n**의 **1~10,000** 사이의 정수 배열 **X**를 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 이 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 0.054142322ms	↦ prefixAverages1의 cpu time ↦ prefixAverages2의 cpu time

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

입력 예시 1		출력 예시 1	
500000	↦ 배열 X 크기	0.421289721ms 0.054142322ms	↦ prefixAverages1의 cpu time ↦ prefixAverages2의 cpu time

필요 함수:

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

※ 참고: 문제 3-2의 실습 목적

- 본 문제는 실행시간(cpu time)을 측정하여 비교분석하는 것이 목적인 실습으로, 출력결과는 컴퓨터마다, 실행할 때마다 다르게 나올 수 있다.
- 코딩평가시스템(OJ 시스템)으로 자동 채점되지 않는 문제로, 출력 결과(실행시간)를 통해 두 함수의 실행시간이 차이가 나는지, **X**가 증가함에 따라 실행시간이 어떤 비율로 증가하는지를 확인해보자.
- 위의 입력 예시 **1**, **2**, **3**을 그대로 사용한 결과를 출력하라.