<알고리즘 실습> - 우선순위 큐(선택 & 삽입 정렬)

※ 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**]이다. 아래 **A**, **B**는 [문제 **3**]을 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] (선택 정렬) **n**개의 양의 정수(중복 가능)를 입력받아, 아래에서 설명하는 선택 정렬을 이용하여 정렬하는 프로그램을 작성하시오.

○ 구현해야 할 선택 정렬 알고리즘 (가장 큰 값을 찾는 버전):
- 크기가 **n**인 배열을 동적 할당하여, 입력된 양의 정수 저장(입력 정수는 중복 가능)
- 제자리(in place) 정렬 사용.
  즉, 입력 값 저장을 위한 배열 이외에 **O**(**1**)의 추가 공간만 사용
- 배열의 뒷 부분을 정렬 상태로 유지하고, 매 반복마다 최대 한 번의 교환 연산만 사용
  (매 반복마다 가장 큰 값을 찾아, 오른쪽부터 채우는 방식으로 정렬)
- 가능하면 교재의 의사코드를 보지 말고 구현해볼 것을 권장
- **참고:** 아래 그림에 예시된 버전은 매 반복마다 가장 큰 값을 찾아 배열의 오른쪽부터 채워 나가는, 즉 교재의 알고리즘과는 정반대 방향으로 작동하는 버전이다. 어느 방향으로 구현하더라도 무방하다.

○ 알고리즘 동작 과정 예시 (**n** = **8**)

초기 상태:	8	31	48	73	3	65	20	29	
1번째 반복 후:	8	31	48	29	3	65	20	**73**	(73과 29 교환)
2번째 반복 후:	8	31	48	29	3	20	**65**	73	(65와 20 교환)
3번째 반복 후:	8	31	20	29	3	**48**	65	73	(48과 20 교환)
4번째 반복 후:	8	3	20	29	**31**	48	65	73	(31과 3 교환)
5번째 반복 후:	8	3	20	**29**	31	48	65	73	(29 제자리 교환)
6번째 반복 후:	8	3	**20**	29	31	48	65	73	(20 제자리 교환)
7번째 반복 후:	3	**8**	20	29	31	48	65	73	(8과 3 교환)

입력 예시 1

```
8                ↦ n
8 31 48 73 3 65 20 29
```

출력 예시 1

```
□3 8 20 29 31 48 65 73          ↦ 정렬 결과
```

입력 예시 2

```
8                ↦ n
73 65 48 31 29 20 8 3
```

출력 예시 2

```
□3 8 20 29 31 48 65 73          ↦ 정렬 결과
```

[문제 2] (삽입 정렬) **n**개의 양의 정수를 입력(중복 가능)받아, 아래에서 설명하는 삽입 정렬을 이용하여 정렬하는 프로그램을 작성하시오.

○ 구현해야 할 삽입 정렬 알고리즘:
- 크기가 **n**인 배열을 동적 할당하여, 입력된 양의 정수 저장(입력 정수는 중복 가능)
- 제자리(in-place) 정렬 사용.
  즉, 입력 값 저장을 위한 배열 이외에 **O**(**1**)의 추가 공간만 사용
- 배열의 앞부분을 정렬 상태로 유지
- 가능하면 교재의 의사코드를 보지 말고 구현해볼 것을 권장

○ 알고리즘 동작 과정 예시 (**n** = **7**)

초기 상태 :	3	73	48	31	8	11	20	
1번째 반복 후:	3	**73**	48	31	8	11	20	73 삽입
2번째 반복 후:	3	**48**	73	31	8	11	20	48 삽입
3번째 반복 후:	3	**31**	48	73	8	11	20	31 삽입
4번째 반복 후:	3	**8**	31	48	73	11	20	8 삽입
5번째 반복 후:	3	8	**11**	31	48	73	20	11 삽입
6번째 반복 후:	3	8	11	**20**	31	48	73	20 삽입

입력 예시 1

```
7          ↦ n
3 73 48 31 8 11 20
```

출력 예시 1

```
□3 8 11 20 31 48 73      ↦ 정렬 결과
```

입력 예시 2

```
8          ↦ n
73 65 48 31 29 20 8 3
```

출력 예시 2

```
□3 8 20 29 31 48 65 73   ↦ 정렬 결과
```

[문제 3] **(주 실습 내용 - OJ 제출하지만 수동채점)** 아래 절차로 여러가지 다양한 입력에 대해 선택 정렬과 삽입 정렬의 실행시간을 측정 비교하라.

○ 작성해야 할 프로그램

① 정렬할 원소의 개수 **n**을 표준입력 받고, 크기가 **n**인 정수 배열 **A**와 **B**를 동적할당 받는다.

② 난수발생 함수(**srand**, **rand** 등)를 사용하여 **n**개의 정수 난수로 배열 **A**와 **B**를 동일하게 초기화한다.

③ 배열 **A**에 대해서는 선택 정렬을, 배열 **B**에 대해서는 삽입 정렬을 수행하고, 시간측정 함수(**clock** 등)를 이용하여 각 정렬에 소요된 시간을 표준출력한다.

입력 예시 1		출력 예시 1	
100000	↦ n	0.051289721ms 0.054142322ms	↦ 선택 정렬 수행시간 ↦ 삽입 정렬 수행시간

○ 실행시간 비교 분석

다음과 같이 다양한 입력 데이터에 대해 두 정렬 알고리즘의 시간을 측정하여, 두 정렬의 성능을 비교 분석해보자.

A. 각 정렬의 입력으로 정렬이 안 된 데이터가 주어지는 경우

a) 동일한 **n**으로 여러 번 실험하여, 어느 정렬이 더 빠른지 비교해보자.

b) **n**이 증가함에 따라 두 정렬의 실행시간이 어떤 비율로 증가하는지 확인해보자.

B. 각 정렬의 입력으로 정렬된 데이터가 주어지는 경우.

※ **참고:** 정렬된 데이터로 실험하기 위해서는 ②번과 ③번 사이에, **A**와 **B**를 아무 정렬 알고리즘이나 사용해서 정렬시키는 과정을 추가하면 된다(시간은 ③에서 측정).

a) 동일한 **n**으로 여러번 실험하여, 어느 정렬이 더 빠른지 비교해보자.

b) **n**이 증가함에 따라 두 정렬의 실행시간이 어떤 비율로 증가하는지 확인해보자.

C. 각 정렬의 입력으로 역순으로 정렬된 데이터가 주어지는 경우.

※ **참고:** 정렬된 데이터로 실험하기 위해서는 ②번과 ③번 사이에, **A**와 **B**를 아무 정렬 알고리즘이나 사용해서 역순으로 정렬시키는 과정을 추가하면 된다(시간은 ③에서 측정).

a) 동일한 **n**으로 여러 번 실험하여, 어느 정렬이 더 빠른지 비교해보자.

b) **n**이 증가함에 따라 두 정렬의 실행시간이 어떤 비율로 증가하는지 확인해보자.