정보개념 정리

```
#include <iostream>
#include <algorithm>
using namespace std;

int n;

//Print Array
//배열 left~right 인덱스 원소 출력

void print(int data[], int left, int right) {
    for(int i=left;i<=right;i++) cout << data[i] << " ";
    cout << endl;
    return;
}

//Bubble sort
//버블 정렬은 인접한 두 원소의 대소를 비교하여 정렬이 완료될 때까지 자리를 바꾸는 정렬이다.

void bubble_sort(int data[], int left, int right) {
    int temp;
    for(int i=left;i<=right-1;i++) {
        for(int j=left;j<=right-i-1;j++) {
            if(data[j]>data[j+1]) {
                temp = data[j];
                data[j] = data[j+1];
                data[j+1] = temp;
            }
        }
    }
    return;
}

//Insertion sort
//삽입 정렬은 그 원소가 들어가야 할 위치를 찾아 그 위치에 원소를 삽입하는 정렬이다.

void insertion_sort(int data[], int left, int right) {
    for(int i=left+1;i<=right;i++) {
        int key = data[i];
```

```
        int idx;
        for(idx=i-1;idx>=0&&data[idx]>=key;idx--) {
            data[idx+1] = data[idx];
        }
        data[idx+1] = key;
    }
    return;
}

//Selection sort
//선택 정렬은 그 인덱스에 들어가야할 원소를 찾아 그 위치에 원소를 넣는 방법이다.

void selection_sort(int data[], int left, int right) {
    int temp;
    for(int i=left;i<=right;i++) {
        int min_idx = i;
        for(int j=i; j<=right;j++) {
            if(data[min_idx] >= data[j]) min_idx = j;
        }
        temp = data[i];
        data[i] = data[min_idx];
        data[min_idx] = temp;
    }
    return;
}

//Quick sort
//0. pivot을 선정한다.(임의적, 보통 가장 왼쪽이나 가장 우측, 중앙, 랜덤으로 설정하는 편이다, 이 코드에서는 가장 왼쪽을 기준으로 선정)
//1. 왼쪽 인덱스 i, 오른쪽 인덱스 j에 대하여, 각각 배열에서 순, 역방향으로 진행하면서 i는 pivot 보다 큰 원소를, j는 pivot 보다 작은 원소를 찾아 i, j의 원소를 교환한다.
//2. 리스트가 pivot보다 작은 원소들과, pivot보다 큰 원소들로 나누어진다. 각각에 대해서 다시 피벗 선정하고 진행한다.
//3. 피벗을 어떻게 잡느냐에 따라서 정렬이 완료될 때의 idx 위치가 달라짐에 유의해야 한다.

void quick_sort(int data[], int left, int right) {
    int pivot = data[left];
    int i=left+1, j=right;
    int temp;
    while(i<=j) {
```

```
        cout << i << endl;
        /* 나누기(partition) */
        while(pivot>data[i]) i++; //left 탐색에서 걸리는 idx 찾기
        while(pivot<data[j]) j--; //right 탐색에서 걸리는 idx 찾기
        if(i<=j) {
            //교환
            temp = data[i];
            data[i] = data[j];
            data[j] = temp;
            i++;
            j--;
        }
    }
    //끝나고 피벗과 교차점의 원소를 교환하여 분할을 완료한다.
    temp = data[left];
    data[left] = data[j];
    data[j] = temp;

    /* 재귀 호출(recursion) */
    if(left < j) quick_sort(data, left, j);
    if(j+1 < right) quick_sort(data, j+1, right);
    return;
}void sort_algorithm(int data[], int left, int right) {
    //오름차순으로 정렬된다. 라이브러리를 이용하여 quick 정렬을 수행한다.
    sort(data+left, data+right);
    return;
}

int main() {
    int data[100];
    cin >> n;
    for(int i=0;i<n;i++) {
        cin >> data[i];
    }
    bubble_sort(data, 0, n-1);
    for(int i=0;i<n;i++) {
        cout << data[i] << " ";
    }
    return 0;
}
```

///정보개념_선형자료구조(리스트, 스택, 큐)
//배열도 선형자료구조에 들어가나, 너무 신물나게 배운 관계로 제낀다.
/*
> 선형 자료 구조(linear data structure)
- 자료를 저장하고 접근하는 관계가 선형적으로 구성되는 자료 구조로서(실제로 그렇게 되는 것은 아니나, 논리적으로 추상화한 것 - 추상화 자료형. abstract data type), 배열, 리스트, 스택, 큐가 있음.
- 배열은 신물나게 배웠으니, 설명을 제외한다. 리스트, 스택, 큐 등은 메모리의 물리적인 공간을 연속적으로 사용하는 배열과는 달리, 불연속적인 공간을 활용 가능함.
- 또한 배열은 크기가 유동적이지 않다는 단점이 있음. - 리스트는 이의 해결이 가능함.(다음 주소만 추가시켜줌으로서 공간을 append하면 되므로)
- 배열을 제외한 추상적 선형 자료 구조들은, std 클래스 선 지정자가 앞에 붙음. using namespace std;로 제거.
> 리스트(list) - 탐색은 -, 자료 재구성은 +
- 하나의 연결 관계에 따라 자료들을 한 줄로 연결시킨 형태로, 추상화된 자료구조
- 배열과는 달리, 연결관계의 재구성만으로 중간에 자료를 삽입, 삭제 가능 - 삽입 정렬 등에서 데이터 이동에 따른 비용 절감의 효과
- 메모리상의 연속적이지 않은 공간을 연속적인 것처럼 사용 가능
- 그 인덱스의 원소와 다름 리스트 주소 저장(리스트에서 원소 접근 위해서는 순차적으로 접근해야 하는 단점)
- 라이브러리: <list>
- 생성자: std::list<자료형>
- 주요 메서드
-> 삽입: push_front(x): 첫번째 원소 앞에 x 삽입(연결), push_back(x): 마지막 원소 다음에 자료 x 삽입(연결) insert(k, x): k 위치 앞에 자료 x 삽입
-> 삭제: pop_front(x): 첫번째 원소 삭제, pop_back(x): 마지막 원소 삭제, erase(k): k 위치의 자료를 삭제.
-> 기타: begin(): 첫번째 원소 위치 반환, end(): 마지막 원소의 다음 위치(종결자) 반환(리스트 구조의 형태를 생각해보라!, 맨 마지막 다음에 종결 인자가 들어올 수 밖에 없다), size() : 리스트에 연결된 자료의 개수 반환
- 이터레이터(iterator, 반복자): 리스트에 들어 있는 자료들의 위치를 선형적 접근이 가능하도록 해주는 자료(pointer, 주소를 16진수 형태로 출력해줌. *iterator는 그 주소에 저장된 값을 돌려준다)
-> 생성자: std::list<자료형>::iterator -> 마치 for 문에서 int i 사용하듯이 사용 가능, iterator++로 다음 주소로 접근 가능함(iterator+=1 등은 사용할 수 없음. 전/후치 증/감 연산자만 사용 가능)
-> 리스트 중간에 삽입하려면, 바로 k번째 위치에 k에 int 쓰면 안되고, 주소가 와야 하므로, iterator를 이동시켜서 삽입해야 한다.
> 스택(stack) - 쌓는다
- LIFO(Last In, First Out) 형태. 쌓아 올리는 자료 구조.
- 이전에 저장된 자료에 접근하기 위해서는, 그 위에 쌓인 자료들 제거 필요. 즉, 삽입과

삭제는 스택의 가장 위에서만 가능.
  - 라이브러리: <stack>
  - 생성자: std::stack<자료형>
  - 주요 메서드
   -> 삽입: push(x): 스택 가장 위에 자료 x 추가.
   -> 삭제: pop(): 스택 가장 위에 있는 자료의 삭제.
   -> 기타: empty(): 스택이 비어 있는지 확인(bool return), size(): 스택의 자료 개수 return. top(): 스택의 최고 위의 값 return
   -> empty 여부를 먼저 검사하지 않고 pop을 실행하면 error raise.
 > 큐(queue) - 통과한다
  - FIFO(First In, First Out) 형태, 관을 통과하는 자료 구조.
  - 순서대로 자료를 넣으며, 넣은 순서대로 자료 접근 가능(끝에서 넣고, 앞에서 확인 - 확인은 앞과 뒤에서 가능하기는 함) - 즉, 뚫린 부분에만 접근 가능함.
  - 라이브러리: <queue>
  - 생성자: std::queue<자료형>
  - 주요 메서드
   -> 삽입: push(x): 큐의 마지막에 자료 x 추가
   -> 삭제: pop(): 큐의 처음의 자료 삭제
   -> 기타: empty(): 큐의 empty 검사(bool return), size(): 큐의 size(자료 개수) return, front(): 큐의 처음 자료 값 return, back(): 큐의 마지막 자료 값 return
   -> empty 여부를 먼저 검사하지 않고 pop을 실행하면 error raise.
*/
```
//list
#include <iostream>
#include <list>
using namespace std;
list<int> mylist;
//list 모두 출력
void view_list() {
    list<int>::iterator it;
    for(it=mylist.begin(); it!=mylist.end(); ++it)  {
        printf("%d", *it); //*it: it위치에 있는 리스트 값(pointer인 it)
    cout<<endl;
    }
}int main() {
    //n개의 자료를 순서대로 list에 넣고 모두 출력
    int n, input;
    cin >> n;
    for(int i=0;i<n;i++) {
        cin >> input;
        mylist.push_back(input);
```

```
    }
    view_list();
    //다음 입력된 자료를 list의 맨 앞에 넣음
    cin >> input;
    mylist.push_front(input);
    //list의 다음 입력된 위치에 그 다음 입력된 원소를 넣음
    cin >> input;
    list<int>::iterator i;
    i = mylist.begin();
    for(int j=0;j<input;j++) i++;
    cin >> input;
    mylist.insert(i, input);
    //전체 리스트를 확인
    view_list();
    return 0;
}/*
//stack
#include <iostream>
#include <stack>
using namespace std;
stack<int> mystack;
int main() {
    //n개의 원소를 stack에 넣음
    int n, input;
    cin >> n;
    for(int i=0;i<n;i++) {
        cin >> input;
        mystack.push(input);
    }
    //stack의 모든 원소를 출력
    while(!mystack.empty()) {
        cout << mystack.top() << endl;
        mystack.pop();
    }
    return 0;
}*/
/*
//queue
#include <iostream>
#include <queue>
using namespace std;
```

```
queue<int> myqueue;
int main() {
    //n개의 원소를 queue에 넣음
    int n, input;
    cin >> n;
    for(int i=0;i<n;i++) {
        cin >> input;
        myqueue.push(input);
    }
    //queue의 front 자료값과 back 자료값 출력
    cout << myqueue.front() << endl << myqueue.back() << endl;
    //queue의 모든 자료값을 넣은 순서대로 출력
    while(!myqueue.empty()) {
        cout << myqueue.front() << endl;
        myqueue.pop();
    }
    return 0;
}*/
```

/*
> 순차 탐색(Squential Search)
 - 리스트 처음부터 끝까지 차례대로 모든 요소를 비교해서 자료를 찾는 탐색 알고리즘
 - 한 쪽 방향으로만 탐색을 수행한다고 해서 선형 탐색(Linear Search)이라고 부르기도 한다.
> 자기 구성 순차 탐색
 - 전진 이동법(한번 검색된 데이터는 데이터집합의 가장 앞으로 이동)
 - 전위법(탐색된 항목을 바로 이전 항목과 교환)
 - 빈도 계수법(탐색된 횟수 정보를 저장하고, 탐색된 횟수가 높은 순으로 재구성)
> 이진 탐색(Binary Search)
 - 정렬된 데이터 집합에서 사용할 수 있는 고속 탐색 방법
 - 탐색 범위를 1/2씩 줄여나가는 방식
 - 이진 탐색의 시간 복잡도는 O(log2 n)이다.
> 이진 탐색 알고리즘(Binary Search Algorithm)
 0. 데이터를 정렬해놓고 탐색을 수행한다.
 1. 데이터 집합의 중앙에 위치한 요소를 고른다.(인덱스가 중앙이라는 것이다)
 2. 중앙 요소의 값과 찾고자 하는 목표값을 비교한다.
 3. 목표 값이 중앙 요소의 값보다 작다면 중앙을 기준으로 데이터 집합의 왼편에 대해, 크다면 오른편에 대해 이진 탐색을 수행
 4. 데이터 나올 때까지 1~3을 수행
> 이진 탐색 성능 분석
 - 탐색 대상의 범위가 1/2, 1/4, 1/8, 1/16 ...
 - 데이터 집합의 크기를 n, 탐색 반복 횟수를 x라 한다면,
 - 1 = n X (1/2)^x
 - x = log_2 n
*/
//아래 Binary Search Algorithm은 1430 Problem C의 Solution에서 들고 왔다.
#include <iostream>
using namespace std;
int data[10000];
int find_linear(int target, int start, int end) {
    for(int i=0;i<n;i++) {
        if(data[i] == target) return i;
    }
    return -1;
}//Binary Search의 재귀함수형
//아래 함수는 찾아진 원소의 index를 return 한다.
int find_binary(int target, int start, int end) {
    int mid = (start+end)/2
    if (target==data[mid]) return mid;
    if ((start-end)/2==0) return -1;

```
    else if (target > data[mid]) return find_binary(target, mid, end);
    else if(target < data[mid]) return find_binary(target, 0, mid);
}//이건 KYH 선생님의 while문으로 Binary Search 구현한 코드
int BinarySearch(int data[], int n, int key) {
    int left=0, right=n-1, mid;
    while(left<=right) {
        mid=(left+right)/2;
        if(key==data[mid])
            return mid;
        else if(key<data[mid])
            right=mid-1;
        else
            left=mid+1;
    }
    return -1;
}//n개의 data를 입력해서 그 중에 찾고자 하는 m개의 input이 몇 번째 원소인지 출력한다.(index는 0부터 시작)
int main() {
    int n, m, input;
    cin >> n;
    for(int i=0;i<n;i++) {
        cin >> data[i];
    }
    cin >> m;
    for(int i=0;i<m;i++) {
        cin >> input;
        cout << find_binary(input, 0, n-1) << " ";
    }
    return 0;
}
```