Part

II

탐색기반 알고리즘의 설계

4. 탐색
5. 전체탐색법
6. 탐색공간의 배제

Part Ⅱ

탐색기반 알고리즘의 설계

이 교재에서는 알고리즘 설계 방법을 크게 탐색기반 설계방법과 관계기반 설계방법으로 나누어 다룬다. 이렇게 분류한 것은 학생들이 알고리즘을 보다 쉽고, 체계적으로 설계하기 위한 편의상의 분류이다.

탐색기반 설계방법은 문제를 탐색 가능한 형태로 구조화한 후 탐색해 나가는 알고리즘 설계방법이다.

하지만 대부분의 경우에는 탐색해야 할 공간이 너무 크기 때문에 문제에서 요구하는 시간 내에 해를 구하지 못하는 경우가 많다. 따라서 전체탐색법을 적절히 응용하는 알고리즘 설계법들을 익힐 필요가 있다.

이 단원에서는 전체탐색법을 기반으로 한 다양한 응용을 익히기 위하여 다양한 탐색방법을 먼저 학습하고, 학습한 탐색방법을 활용하여 주어진 문제를 해결할 수 있는 알고리즘 설계를 실습한다.

4 탐색

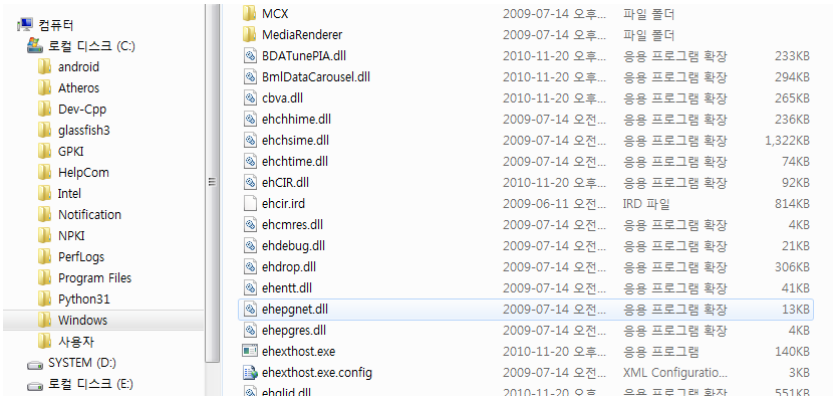
정보과학에서 탐색은 매우 중요하게 다루어지는 주제 중 하나이다. 일반적으로 탐색이란 같은 형태의 한 개 이상의 자료들이 적절한 형태로 구조화된 상태로 모여 있는 집합에서 특정 자료를 찾는 모든 작업을 말한다.

따라서 탐색은 탐색할 자료가 저장되어 있는 구조를 먼저 파악하는 것이 중요하다. 문제의 특성에 따라 구조가 명시적으로 드러나는 경우는 쉬운 문제에 속하고, 문제에서는 탐색구조가 직접적으로 드러나지는 않으나, 문제를 해결하는 과정에서 자체적으로 구조화하며 탐색하는 경우는 중급 이상의 문제라 할 수 있다.

이 교재에서는 탐색의 대상이 되는 구조를 선형구조와 비선형구조로 나누어 탐색을 실습한다. 일반적으로 배열이나 연결리스트로 표현될 수 있는 구조를 선형구조, 트리나 그래프의 형태로 표현되는 구조를 비선형구조로 구분한다.

탐색의 대상이 되는 구조화된 그룹의 예를 알아보자. 누구나 사용하고 있는 컴퓨터의 운영체제를 살펴보자. 운영체제에서 사용자의 자료를 파일의 형태로 보관하고, 이러한 파일들을 그룹화 한 것을 폴더라고 한다.

파일과 폴더를 선형으로 구성할 경우와 비선형으로 구성하는 경우 각각 어떤 장, 단점이 있는지 생각해보자.



Windows의 탐색기

위 그림은 일반적으로 운영체제에서 파일과 폴더를 구조화하는 방법이다. 위 구조는 자료를 트리형태로 구성하여 탐색하기 쉽도록 구성한다는 사실을 알 수 있다.

만약 파일과 폴더를 선형으로 구성한다면 어떤 단점이 있을지 생각해보자.

그리고 위 구조에서 C드라이브에서 내가 원하는 파일을 찾고자 한다면 어떤 탐색법이 필요할지 생각해 보는 것도 알고리즘 설계능력을 키우는데 많은 도움이 된다. 단순한 반복문으로 모든 폴더를 방문하는 알고리즘을 설계하기는 쉽지 않다.

탐색기반설계는 주어진 문제에서 주어진 데이터를 특성에 맞도록 구조화하고 이 자료를 적절한 방법으로 탐색해 나가면서 원하는 해를 찾는 알고리즘 설계법이며 탐색하는 범위

에 따라서 크게 전체탐색법이라 하고, 탐색할 영역을 적절한 방법으로 배제하여 탐색의 효율을 높인 방법을 부분탐색법이라 한다.

이 단원에서는 전체탐색법과 부분탐색법으로 주어진 문제의 해를 구하는 방법에 대해서 학습한다.

가. 선형구조의 탐색

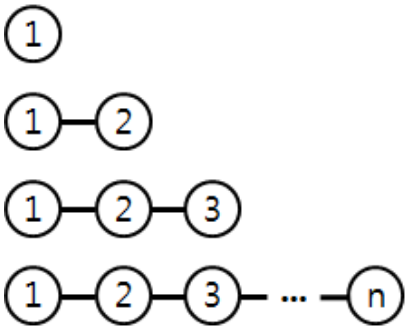
선형구조란 자료의 순서를 유일하게 결정할 수 있는 형태의 구조를 말한다. i번째 자료를 탐색한 다음, $i+1$번째로 탐색해야할 자료가 유일한 형태를 의미한다. 2차원, 3차원 구조라도 순서가 일정하게 정해져 있으면 이는 선형이라고 할 수 있다.

선형구조는 주로 배열과 리스트의 형태로 저장된다. 일반적으로 1차원 배열에 자료를 저장하는 1차원 선형구조와 2차원 이상의 배열에 자료가 저장이 되어있는 다차원 선형구조로 나눌 수 있다.

선형구조의 탐색은 선형구조로 저장된 자료들 중에서 원하는 것을 찾는 작업을 말한다. 선형구조를 탐색하는 방법은 기본적으로 순차탐색과 이분탐색이 있고, 이들을 적절히 응용한 탐색법도 만들어 사용할 수 있다. 이 단원에서는 순차탐색과 이분탐색을 익히고 이를 통하여 간단한 문제를 해결하는 실습을 한다.

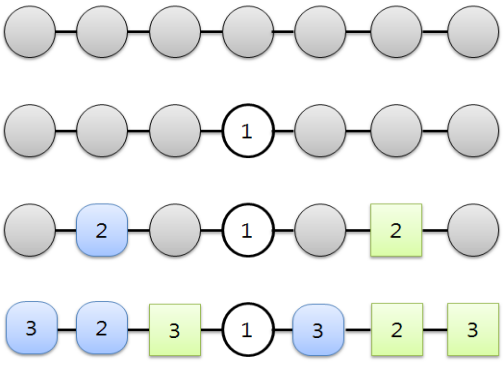
순차탐색은 자료의 특성에 관계없이 사용할 수 있는 일반적인 방법으로 전체탐색기법의 한 방법이다. 첫 번째 원소로부터 시작하여 한 원소씩 차례로 다음 원소를 탐색해 나가는 방법으로 자료가 n개 있을 때의 계산량은 $O(n)$이다.

탐색 순서를 그림으로 나타내면 다음과 같다.



순차탐색의 순서

다음 알고리즘은 배열 S에 n개의 원소를 입력받고, 그 중에 k가 있는지를 찾는 알고리즘이다. 이 알고리즘은 오름차순이나 내림차순으로 정렬된 선형구조에서 원하는 원소를 찾는 것으로 계산량은 $O(\log_2 n)$이다.



이분탐색의 탐색순서(○는 처음 접근하는 원소이고, 사각형은 찾은 곳의 값이 찾으려는 값보다 작으면 찾는 위치, 둥근 사각형은 그 값의 반대조건일 경우에 탐색하는 위치이다. 조건의 결과에 따라 왼쪽 또는 오른쪽 중 하나를 탐색하게 된다.)

다음은 이분탐색을 구현한 소스코드이다. 이를 재귀함수로도 구현할 수 있으므로 한 번 생각해보기 바란다.

줄	코드	참고
1	#include <stdio.h>	
2		
3	int S[100], n, k;	
4		
5	int find(int s, int e)	
6	{	
7	while(s<=e)	
8	{	
9	int m=(s+e)/2;	
10	if(S[m]==k) return m;	
11	if(S[m]>k) e=m-1;	
12	else s=m+1;	
13	}	
14	return -1;	
15	}	
16		
17	int main()	
18	{	
19	scanf("%d%d", &n, &k);	
20	for(int i=0; i<n; i++)	
21	scanf("%d", &S[i]);	
22	printf("%d\n", find(0, n-1));	
23	return 0;	
24	}	

기본적인 탐색방법을 익힐 수 있는 다음 문제들을 해결해보자.

문제 1

최댓값(S)

9개의 서로 다른 자연수가 주어질 때, 이들 중 최댓값을 찾고 그 값이 몇 번째 수인지를 구하는 프로그램을 작성하시오.

예를 들어, 서로 다른 9개의 자연수가 각각

3, 29, 38, 12, 57, 74, 40, 85, 61

라면, 이 중 최댓값은 85이고, 이 값은 8번째 수이다.

입력

첫째 줄부터 아홉째 줄까지 한 줄에 하나의 자연수가 주어진다. 주어지는 자연수는 100보다 작다.

출력

첫째 줄에 최댓값을 출력하고, 둘째 줄에 최댓값이 몇 번째 수인지를 출력한다.

입력 예	출력 예
3 29 38 12 57 74 40 85 61	85 8

출처: 한국정보올림피아드(2007 지역본선 초등부)

풀이

이 문제는 자료를 1차원 배열에 저장한 후 반복문을 이용하여 전체탐색법을 구현하면 쉽게 구할 수 있다. 전체탐색을 하더라도 탐색해야할 자료의 수가 9개뿐이므로 충분히 빠른 시간 내에 해를 구할 수 있는 기본적인 문제이다.

따라서 반복문을 구현하는 연습을 할 수 있는 문제로 이 문제를 해결하는 방법이 다른 문제들을 해결하는 도구로 많이 활용될 수 있으므로 꼭 익혀둘 수 있도록 한다.

일단 먼저 문제해결 아이디어를 생각하자. 최종적으로 출력할 해를 변수 ans로 두고, 최댓값의 인덱스를 저장할 변수를 index로 설정한다.

먼저 모든 자료를 탐색하기 전에 ans를 모든 원소들 보다 작은 값으로 설정한다. 이 문제에서는 100 이하의 자연수가 데이터의 정의역이므로, 0으로 설정하면 된다. 다음으로 첫 번째 자료부터 마지막 자료까지 하나씩 검사해가며 현재까지 ans보다 더 큰 값이 나타나면 ans를 갱신하고, index값도 갱신한다.

마지막 자료까지 탐색을 마치면, ans와 index를 출력하면 된다. 이 과정을 입출력 예를 통해서 알아보자.

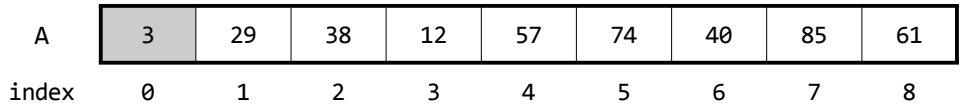
준비 단계

A	3	29	38	12	57	74	40	85	61
index	0	1	2	3	4	5	6	7	8

현재 탐색 중인 index = ?　　ans　0

〈입력받은 상태에서 탐색 준비를 한다. 탐색의 순서는 index가 0부터 8까지〉

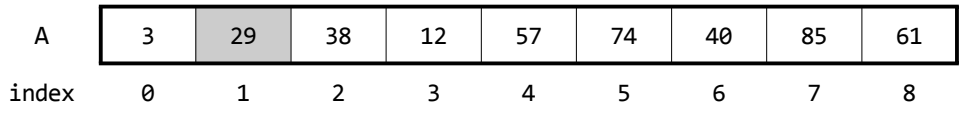
[1 단계]



현재 탐색 중인 index = 0 ans 3

〈A[0]의 값이 ans보다 크므로 ans의 값은 3이 된다.〉

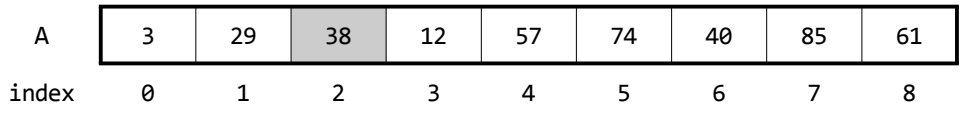
[2 단계]



현재 탐색 중인 index = 1 ans 29

〈A[1]의 값이 현재 ans의 값 3보다 크므로 ans값은 29로 갱신〉

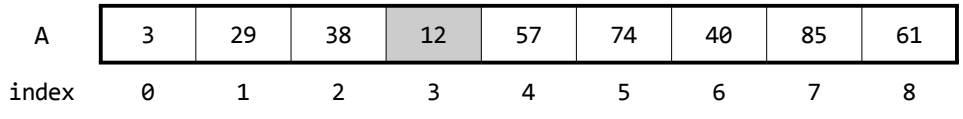
[3 단계]



현재 탐색 중인 index = 2 ans 38

〈A[2]의 값이 현재 ans의 값 29보다 크므로 ans값은 38로 갱신〉

[4 단계]

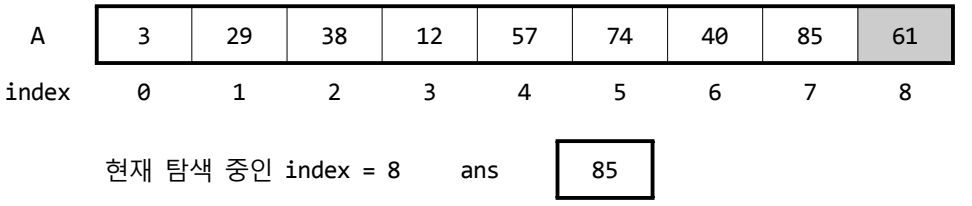


현재 탐색 중인 index = 3 ans 38

〈A[3]의 값이 현재 ans보다 작으므로 ans는 38을 유지〉

⋮

[9 단계]



〈A[8]의 값이 현재 ans보다 작으므로 ans는 85를 유지〉

이와 같이 배열을 선형으로 전체탐색을 하면서 최댓값을 구할 수 있다. 이 방법은 가장 기본적인 방법 중 하나로 다른 알고리즘에 많이 응용되는 방법이므로 잘 익혀둘 수 있도록 한다. 이를 프로그램으로 구현하면 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`#define MAXN 9`	
3		
4	`int main()`	
5	`{`	
6	`  int i, ans, index, A[MAXN+1];`	
7	`  ans = 0;`	
8	`  index = 0;`	
9	`  for( i=1; i<MAXN+1; i++)`	
10	`    scanf("%d", &A[i]);`	
11	`  for( i=1; i<MAXN+1; i++)`	
12	`    if( ans<A[i] )`	
13	`    {`	
14	`      ans = A[i];`	
15	`      index = i;`	
16	`    }`	
17	`  printf("%d\n%d\n", ans, index);`	
18	`  return 0;`	
19	`}`	

위 프로그램은 문제 1을 푼 가장 표준적인 코드라고 볼 수 있다.

자세한 내용을 설명하기에는 너무 길 수 있으므로 간단하게 말하자면, main 함수를 비롯한 모든 함수들은 함수 내에서 사용되는 모든 변수를 지역변수로 쓰는 것이 좋다. 즉 함수를 완전히 독립 모듈로 보더라도 모든 데이터는 모듈 내에서 정의되어야 한다는 의미이다. 만약 어떤 함수가 전역변수를 참조한다면, 이 함수를 다른 프로그램에 가져가면 전역변수를 더 이상 참조할 수 없으므로, 이는 좋은 모듈이라 할 수 없다.

하지만 정보올림피아드와 같은 대회에서는 주어진 짧은 시간에 알고리즘을 구현하는 것이 중요하다.

일반적으로 대회에 경험이 있는 학생들이 문제 1을 해결하는 데 구현하는 코드는 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`#define MAXN 9`	
3	`int ans, index, A[MAXN+1];`	
4		
5	`void solve(void)`	
6	`{`	
7	`  for(int i=1; i<10; i++)`	
8	`  {`	
9	`    scanf("%d", &A[i]);`	
10	`    if(ans<A[i]) ans=A[i], index=i;`	
11	`  }`	
12	`}`	
13		
14	`int main()`	
15	`{`	
16	`  solve();`	
17	`  printf("%d\n%d\n", ans, index);`	
18	`  return 0;`	
19	`}`	

이 풀이에서는 일반적으로 좋지 않은 방법으로 보이는 코드들이 많이 보인다. ans, index를 모두 전역으로 선언하고 있으며, 초기화도 하지 않고 있다.

실제 대회에서는 이와 같이 전역변수를 활용하는 경우가 많다. 특히 배열을 선언하는 경우는 대부분 전역변수를 활용한다. 가장 큰 이유는 지역변수보다 더 많은 공간을 확보

할 수 있다는 장점이 있으며, 0으로 초기화되는 점도 무시할 수 없기 때문이다.

그리고 solve라는 모듈로 풀이를 분리하는 것도 자주 볼 수 있는데, 각종 대회에서는 디버깅을 빨리할 수 있는 능력이 매우 중요하다. 이와 같이 모듈을 입력, 처리, 출력으로 나눠두면 오류가 발생했을 때, 이를 처리하기 용이하다.

마지막으로 for문 내부에 i와 같은 반복문의 인덱스 변수를 선언한다는 점이다. 프로그램 전체적으로 i와 같은 변수를 자주 활용하게 되는데 이로 인한 오류가 의외로 많이 발생한다. 따라서 각 반복문별로 영역을 제한하여 사용하는 것도 일종의 오류를 줄이기 위한 팁이라고 할 수 있다.

중요한 것은 자기만의 코딩스타일을 구축하여 실수를 최소화하는 것이 매우 중요한 점이기 때문에 쉬운 문제들을 다룰 때부터 자신만의 습관을 기르는 방법을 익힐 수 있도록 하자.

마지막으로 다음과 같이 더 효율적으로 작성할 수도 있으니 참고하기 바란다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`#define MAXN 9`	
3	`int ans, A[MAXN+1];`	
4		
5	`void solve(void)`	
6	`{`	
7	`  for(int i=1; i<10; i++)`	
8	`  {`	
9	`    scanf("%d", A+i);`	
10	`    if(A[ans]<A[i]) ans=i;`	
11	`  }`	
12	`}`	
13		
14	`int main()`	
15	`{`	
16	`  solve();`	
17	`  printf("%d\n%d\n", A[ans], ans);`	
18	`  return 0;`	
19	`}`	

이번 풀이에서는 ans, index를 하나의 변수 ans로 처리하고 있다. 그리고 9행에서 입력받을 때 “&A[i]” 대신 “A+i”를 활용하고 있다. 이러한 코딩 스타일도 자주 활용되는 방법으로 배열과 포인터를 이해하면 위와 같이 사용할 수 있음을 알 수 있다. 이와 같을 때에는 특수문자로 인한 오타의 확률도 줄일 수 있으므로 다양한 방법을 익힐 수 있도록 하자.

그리고 이 문제에서는 배열을 쓰지 않고도 해결 가능하므로 다양한 방법으로 코딩 연습을 해두는 연습을 하자.

문제 2

3의 배수 게임

3의 배수 게임을 하던 정올이는 3의 배수 게임에서 잦은 실수로 계속해서 벌칙을 받게 되었다.

3의 배수 게임의 왕이 되기 위한 마스터 프로그램을 작성해 보자.

** 3의 배수 게임이란?

여러 사람이 순서를 정해 순서대로 수를 부르는 게임이다.
만약 3의 배수를 불러야 하는 상황이라면, 그 수 대신 "박수" 를 친다.

입력

첫 째 줄에 하나의 정수 n이 입력된다(n은 10미만의 자연수이다.).

출력

1부터 그 수까지 순서대로 공백을 두고 수를 출력하는데, 3 또는 6 또는 9인 경우 그 수 대신 영문 대문자 X를 출력한다.

입력 예	출력 예
7	1 2 X 4 5 X 7

풀이

이 문제도 [문제 1]과 마찬가지로 단순히 반복문을 이용하여 전체탐색법으로 해결할 수 있다. 단지 이 문제는 특정 값을 찾거나 하는 것이 아니라 전체 데이터를 읽으면서 특정 자료가 있으면 변경한다는 점은 다르나 전반적으로 같은 방법으로 해결할 수 있다.

이 문제에서 특정 자료란 입력된 숫자가 3의 배수일 경우를 말한다. 임의의 변수 n이 3의 배수인지 판정하는 가장 일반적인 방법은 다음과 같은 방법을 이용한다.

```
n % 3 == 0
```

또는 정수의 나눗셈의 특성을 이용한 다음과 같은 방법도 있다.

```
n / 3 * 3 == n
```

이 방법의 경우는 3의 배수가 아니면 3으로 나누어서 곱할 때 원래의 수가 되지 않는다. 다음 표는 1부터 10까지의 자연수를 3으로 나눈 후 곱할 때의 값을 나타낸다.

n	n/3	n/3*3==n
1	0	False
2	0	False
3	**1**	**True**
4	1	False
5	1	False
6	**2**	**True**
7	2	False
8	2	False
9	**3**	**True**
10	3	False

위의 방법들 이외에도 다양한 방법들이 있으므로 다양하게 생각해보기 바란다. 문제를 해결할 때, 1부터 n까지 1씩 증가하여 탐색하면서 각 수가 3의 배수인지 판정하여 3의 배

수이면 "X"를 아니면 그 수를 출력하도록 작성하면 쉽게 해결할 수 있다.

이 문제를 해결한 예시는 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	
2		
3	`int n;`	
4		
5	`void solve(void)`	
6	`{`	
7	`  for(int i=1; i<=n; i++)`	
8	`  {`	
9	`    if(i%3==0) printf("X ");`	
10	`    else printf("%d ", i);`	
11	`  }`	
12	`}`	
13		
14	`int main()`	
15	`{`	
16	`  scanf("%d", &n);`	
17	`  solve();`	
18	`  return 0;`	
19	`}`	

이 문제에서도 입력값 n을 전역변수로 두고 있다. 이 방법에 대해서는 각자 자신의 방법을 정하기 바라며, solve 함수의 for문에서 i를 1부터 시작해야 한다는 점도 유의할 필요가 있다. 일반적으로 반복문의 0부터 출발하거나 1부터 출발하는데, 문제의 특성과 개인의 성향에 따라서 차이가 있다. 이 부분 또한 자신의 스타일을 정확하게 정해두면 실수를 줄일 수 있다.

앞으로 입출력에 특별한 사항이 없을 때에는 풀이는 다음과 같이 solve 함수 부분만 제시하는 경우도 있을 것이다. 이때는 전역변수와 main 함수는 solve 함수로 충분히 유추할 수 있을 것이다.

줄	코드	참고
1	void solve(void)	5: 두 번째 방법을 이용
2	{	
3	for(int i=1; i<=n; i++)	
4	{	
5	if(i/3*3==i) printf("X ");	
6	else printf("%d ", i);	
7	}	
8	}	

문제 3

linear structure search

n개로 이루어진 정수 집합에서 원하는 수의 위치를 찾으시오.

단, 입력되는 집합은 오름차순으로 정렬되어 있으며, 같은 수는 없다.

입력

첫 줄에 한 정수 n이 입력된다.
둘째 줄에 n개의 정수가 공백으로 구분되어 입력된다.
셋째 줄에는 찾고자 하는 수가 입력된다.
(단, 2 <= n <= 1,000,000 , 각 원소의 크기는 100,000,000을 넘지 않는다.)

출력

찾고자 하는 원소의 위치를 출력한다. 없으면 -1을 출력한다.

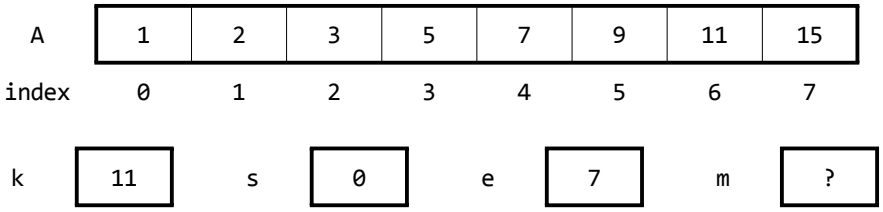
입력 예	출력 예
8 1 2 3 5 7 9 11 15 11	7
3 2 5 7 3	-1

풀이

이 문제는 앞에서 다룬 이분탐색의 예제 프로그램을 거의 그대로 활용할 수 있는 문제이다. 일단 탐색 범위를 [s, e]로 정한 다음 범위의 가운데 위치의 값을 m으로 설정하고 탐색을 시작한다.

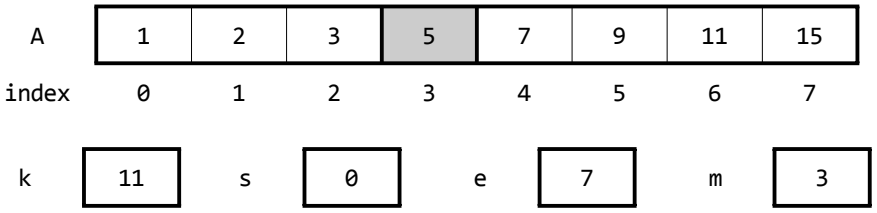
배열을 A라고 할 때, A[m] == k 인 경우와 A[m] 〉 k, A[m] 〈 k인 경우로 나누어 처리하는 방법으로 문제를 해결할 수 있다. 자세한 과정은 다음과 같다.

준비 단계



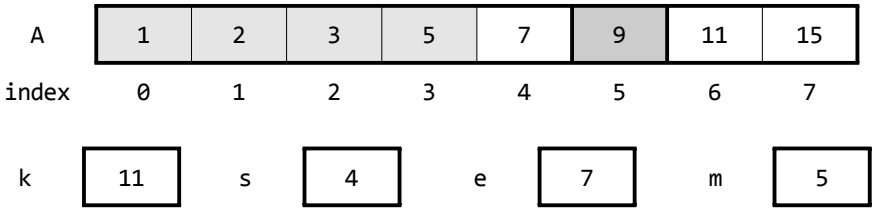
〈입력받은 상태에서 탐색 준비를 한다. 탐색 범위는 0~7〉

[1 단계]



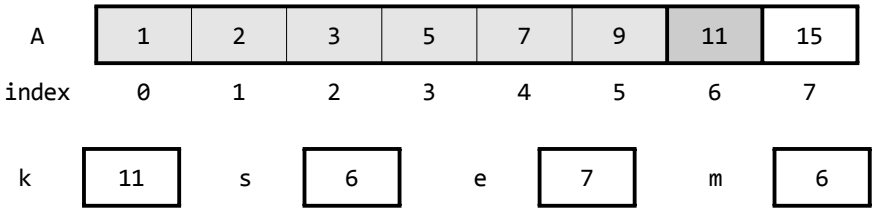
〈A[3]의 값이 k보다 작으므로, 영역을 4~7로 변경〉

[2 단계]



〈A[5]의 값이 k보다 작으므로, 영역을 6~7로 변경〉

[3 단계]



〈A[6]의 값과 k가 같으므로 탐색종료, 찾은 인덱스는 6〉

위 방법을 소스코드로 작성하면 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`int n, k, A[1000001];`	
3	`int solve(int s, int e)`	
4	`{`	
5	`  int m;`	
6	`  while(e-s>=0)`	
7	`  {`	
8	`    m=(s+e)/2;`	
9	`    if(A[m]==k)`	
10	`      return m+1;`	
11	`    if(A[m]<k) s=m+1;`	
12	`    else e=m-1;`	
13	`  }`	
14	`  return -1;`	
15	`}`	
16	`int main()`	
17	`{`	
18	`  scanf("%d",&n);`	
19	`  for(int i=0; i<n; i++ )`	
20	`    scanf("%d", A+i);`	
21	`  scanf("%d",&k);`	
22	`  printf("%d\n", solve(0, n-1));`	
23	`  return 0;`	
24	`}`	

앞의 예제에서는 5행을 e 〉= s로 작성했으나 일반화를 위하여 e−s 〉= 0 으로 활용했으며, 나머지 이분탐색 부분들도 자세히 분석하여 익힐 수 있도록 한다.

위 소스코드를 다음과 같은 재귀함수로도 만들 수 있다. 재귀함수는 매우 다양한 응용이 가능하므로 이해해두면 많은 도움이 된다.

줄	코드	참고
1	#include <stdio.h>	
2	int n, k, A[1000001];	
3	int solve(int s, int e)	
4	{	
5	if(s>e)	
6	return -1;	
7	int m=(s+e)/2;	
8	if(A[m]==k)	
9	return m+1;	
10	if(A[m]<k)	
11	return solve(m+1, e);	
12	else	
13	return solve(s, m-1);	
14	}	
15	int main()	
16	{	
17	scanf("%d",&n);	
18	for(int i=0; i<n; i++)	
19	scanf("%d", A+i);	
20	scanf("%d",&k);	
21	printf("%d\n", solve(0, n-1));	
22	return 0;	
23	}	

문제 4

lower bound

n개로 이루어진 정수 집합에서 원하는 수 k 이상인 수가 처음으로 등장하는 위치를 찾으시오.

단, 입력되는 집합은 오름차순으로 정렬되어 있으며, 같은 수가 여러 개 존재할 수 있다.

입력

첫째 줄에 한 정수 n이 입력된다.
둘째 줄에 n개의 정수가 공백으로 구분되어 입력된다.
셋째 줄에는 찾고자 하는 값 k가 입력된다.

(단, 2 <= n <= 1,000,000 , 각 원소의 크기는 100,000,000을 넘지 않는다.)

출력

찾고자 하는 원소의 위치를 출력한다. 만약 모든 원소가 k보다 작으면 n+1을 출력한다.

입력 예	출력 예
5 1 3 5 7 7 7	4
8 1 2 3 5 7 9 11 15 6	5
5 1 2 3 4 5 7	6
5 2 2 2 2 2 1	1