안녕하세요, 동계 대학생 S/W 알고리즘 특강의 열한 번째 시간인 오늘은 이분 탐색에 대해 다루어보도록 하겠습니다.

1. 기초 강의

동영상 강의 컨텐츠 확인 > 10. 이분탐색

Link :

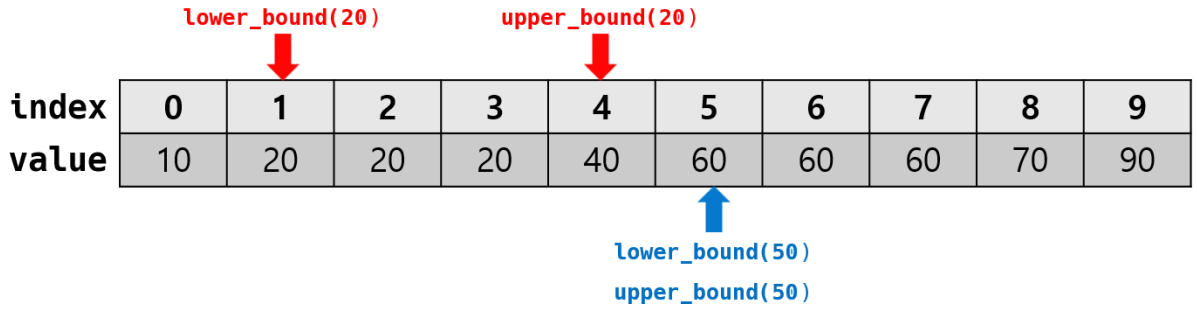
https://swexpertacademy.com/main/learn/course/subjectDetail.do?courseId=CONTENTS_REVIEW&subjectId=AYVXyQd6RHIDFARs

※ 출석은 강의 수강 내역으로 확인합니다.

2. 실전 강의

2.1. std::lower_bound, std::upper_bound

이분 탐색은 False 구간/ True 구간 두 개의 파티션으로 분할된 구간에서 분할 경계의 위치를 O(logN)에 찾아주는 알고리즘입니다. C++에서 가장 유명한 이분 탐색 함수로는 std::lower_bound 와 std::upper_bound 가 있습니다. 정렬된 배열에서 x 의 lower_bound 는 정렬 상태를 유지한 채 x 를 삽입할 수 있는 첫 번째 위치이며, upper_bound 는 마지막 위치입니다.



이분 탐색적인 설명으로는, lower_bound 는 x 보다 **크거나 같은** 원소의 첫 번째 위치고 upper_bound 는 x 보다 **큰** 원소의 첫 번째 위치입니다. 배열의 값이 x 이상인지, x 초과인지에 따라 배열을 False / True 두 파티션으로 나눌 수 있습니다. 두 함수 모두 첫 번째 True 가 가리키는 위치를 찾아줍니다.

lower_bound(20)

index	0	1	2	3	4	5	6	7	8	9
value	10	20	20	20	40	60	60	60	70	90
value >= x	False	True	True	True	True	True	True	True	True	True

upper_bound(20)

index	0	1	2	3	4	5	6	7	8	9
value	10	20	20	20	40	60	60	60	70	90
value > x	False	False	False	False	True	True	True	True	True	True

2.2. 첫 번째 True 찾기

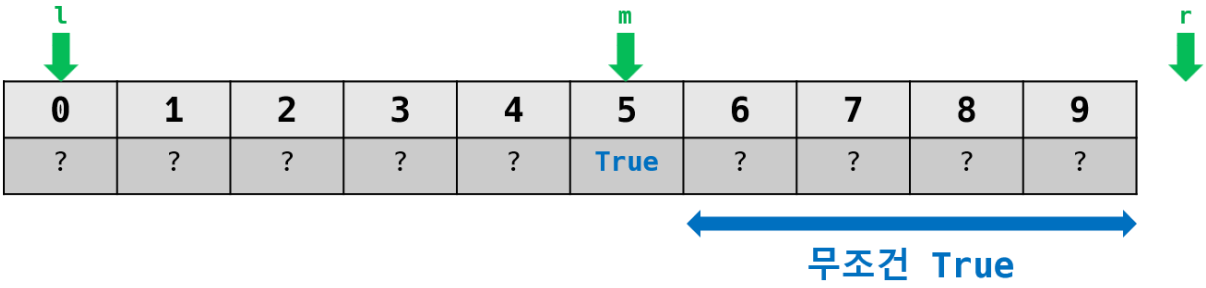
MIN 이상 MAX 이하인 구간에서 f(i) = True 인 첫 번째 i 를 찾는 코드는 아래와 같습니다.

```
// f(i) = True 인 첫 번째 i 를 리턴
// 만약 [ MIN , MAX ]이 전부 False 라면 MAX + 1 을 리턴
Type first_true(Type MIN, Type MAX) {
	Type l = MIN, r = MAX + 1;
	// [ l , r )은 "False 일 가능성이 있는 미확인 구간"입니다.
	while (l != r) {
		// (l + r) / 2 는 안됩니다.
		// 오버플로우의 위험이 있고, l + r 이 음수일 경우 나누기 2 가 다르게
동작합니다.
		Type m = l + (r - l) / 2;
		// 결과에 따라 남은 구간을 반으로 줄입니다.
		f(m) ? r = m : l = m + 1;
	}
	// 미확인 구간이 전부 없어지면 첫 번째 True 는 바로 그 다음 위치입니다.
	return l;
}
```

결과에 따라 남은 구간을 반으로 줄이는 동작은 아래처럼 진행됩니다.

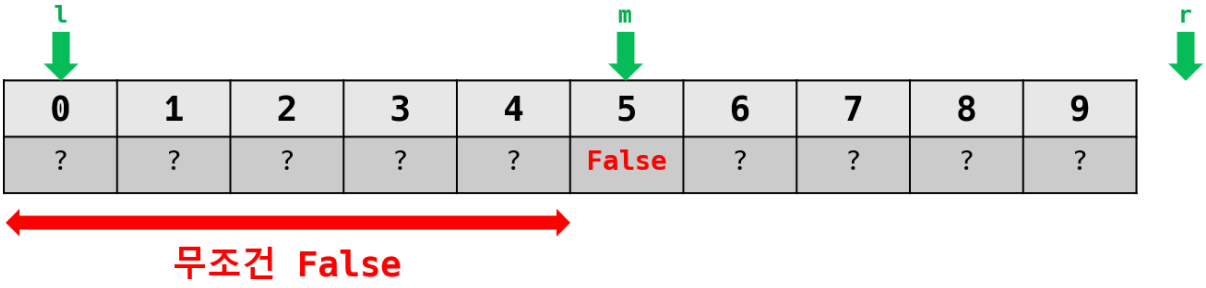
중간값이 True 일 경우

[m + 1 , r)은 무조건 True 입니다. False 일 가능성이 있는 미확인 구간은 [l , m)입니다.



중간값이 False 일 경우

[l , m)은 무조건 False 입니다. False 일 가능성이 있는 미확인 구간은 [m + 1 , r)입니다.



위의 방식을 그대로 적용해서 구현한 lower_bound 와 upper_bound 입니다.

```
int* lower_bound(int* lo, int* hi, int x) {
	while (lo != hi) {
		int* const mid = lo + (hi - lo) / 2;
		*mid >= x ? hi = mid : lo = mid + 1;
	}
	return lo;
}

int* upper_bound(int* lo, int* hi, int x) {
	while (lo != hi) {
		int* const mid = lo + (hi - lo) / 2;
		*mid > x ? hi = mid : lo = mid + 1;
	}
	return lo;
}
```

2.3. 마지막 True 찾기

구간이 True / False 파티션으로 나눠져 있을 때 마지막 True 를 찾는 함수도 비슷한 방법으로 구현할 수 있습니다. 2.2 에서 사용한 논리를 그대로 뒤집으면 됩니다.

MIN 이상 MAX 이하인 구간에서 f(i) = True 인 마지막 i 를 찾는 코드는 아래와 같습니다.

```
// f(i) = True 인 마지막 i 를 리턴
// 만약 [ MIN , MAX ]이 전부 False 라면 MIN - 1 을 리턴
Type last_true(Type MIN, Type MAX) {
	Type l = MIN – 1, r = MAX;
	while (l != r) {
		Type m = r - (r - l) / 2;
		f(m) ? l = m : r = m - 1;
	}
	return l; // 또는 r
}
```

2.4. Parametric Search

이분 탐색을 단순히 정렬된 배열 안에서 원소를 찾는 알고리즘으로 한정하면 안됩니다. 이를 응용하여 최적화 문제를 결정 문제로 바꾸어 푸는 파라매트릭 서치 알고리즘이 있습니다.

최적화 문제 -> f(x) = True 가 되는 x 의 최대값을 구하라
결정 문제 -> 어떤 x 에서 f(x) = True 인가?

가능한 모든 x 값마다 f(x)가 True 인지 False 인지 검사하면 최적화 문제를 결정 문제로 풀 수 있습니다. 그렇게 검사한 값 중 True 인 최소/최대값을 고르면 됩니다. 이 방법은 매우 비효율적입니다.

하지만 한 가지 조건을 만족하면 위 과정을 더 빠르게 할 수 있습니다. x1 < x2 이고 f(x2) = True 이면 f(x1) = True 라는 조건입니다. 이 조건을 만족한다면 f(x)의 결과값이 정렬된 형태이기 때문에 이분 탐색 아이디어를 그대로 적용할 수 있습니다.

임의의 값 x 에 대해 f(x)가 True 인지 False 인지 검사합니다. f(x)가 True 일 경우 x 이하도 전부 True 이며, f(x)가 False 일 경우 x 이상도 전부 False 임을 알 수 있습니다. 때문에 마치 이분 탐색을 하듯이 x 에 대한 범위를 절반씩 줄일 수 있고, O(log(x 의 범위))번의 결정 문제로 최적화 문제를 풀 수 있습니다.

예시 문제)
n 개의 섬과 m 개의 다리가 있다. 다리마다 무게 제한(1<= 제한 <= 10^9)이 있어서 제한보다 무거운 차량이 지나가면 다리가 무너진다. 0 번 섬에서 n – 1 번 섬으로 이동할 수 있는 차량 무게의 최댓값은 얼마인가?

이 문제를 푸는 방법은 Dijkstra, Union-Find, Parametric Search 등이 있습니다. 여기선 파라매트릭 서치 해법을 소개합니다.
만약 무게가 x 인 차량이 0 번 섬에서 n – 1 번 섬으로 이동할 수 있다면, 그 길을 그대로 따라서 x 보다 가벼운 차량도 이동할 수 있습니다. 만약 무게가 x 인 차량이 0 번 섬에서 n – 1 번 섬으로 이동할 수 없다면 x 보다 무거운 차량도 이동할 수 없습니다.
따라서 f(i) = 무게가 i 인 차량이 0 번 섬에서 n – 1 번 섬으로 이동 가능한가? 의 결과를 표로 나타내면 아래처럼 이분 탐색이 가능한 꼴이 나옵니다.

i	1	…	x	x+1	…	10^9
$f(i)$	True	…	True	False	…	False

무게 제한이 1 이상 10^9 이하이므로, 1 보다 가벼운 차량과 10^9 보다 무거운 차량의 이동 여부는 조사할 필요가 없습니다. 탐색 범위를 1 이상 10^9 이하로 두고 이분 탐색을 하면 됩니다.

```
constexpr int MIN = 1;
constexpr int MAX = 1e9;

bool f(int limit) {
	// 제한이 limit 이하인 다리만 사용해서 0 -> n - 1 이동이 가능한가?
}

// 0 -> n - 1 로 이동 가능한 차량 무게의 최댓값 리턴
// 만약 0 이 리턴되면 0 -> n - 1 로 애초에 이동이 불가능한 경우다.
int solve() {
	int l = MIN - 1, r = MAX;
	while (l != r) {
		const int m = r - (r - l) / 2;
		f(m) ? l = m : r = m - 1;
	}
	return l;
}
```

2.5. 이분 탐색 구현 팁

이분 탐색을 구현법은 사람마다 다양합니다. 본문은 탐색 구간 끝에 1 을 더해서 반열린 구간으로 만든 다음 STL 구현 방식을 따랐습니다. 다른 방법도 있지만 가급적 본문 방식을 사용하는 걸 권합니다.
이분 탐색을 구현할 때 종료 조건과, 구간을 쪼개는 과정에서 무한 루프를 조심해야 합니다. 본문 구현에서 +1, -1 을 다른 쪽에 하거나, m 값을 구하는 방법을 바꾼다면 무한 루프가 되거나 어느 한 점은 탐색하지 않고 건너 뛰게 될 겁니다.
이분 탐색 구현에서 아래처럼 3 가지 경우를 고려하는 방식은 피하십시오. 항상 구간을 절반으로 나눠서 생각하는 게 올바른 이분 탐색 사고입니다.

```
// WARNING! 이렇게 하지 마세요!
while (l != r) {
	const int m = l + (r - l) / 2;
	if (array[m] < key) {
		// ...
	} else if (array[m] == key) {
		// ...
	} else {
		// ...
	}
}
```

3. 기본 문제

· 영어 공부
· 촛불 이벤트
· 사탕 가방
· 광고 시간 정하기
· 3 차원 농부