



실전 알고리즘 0x0C강 이분탐색

BaaaaaaaaaaaaaaaaarkingDog

목차



0x00 이분탐색(Binary Search)

0x01 예시 문제 1 : 수 찾기

0x02 예시 문제 2 : 숫자 카드 2

0x03 주의사항

0x04 관련 STL

0x00 이분탐색(Binary Search)



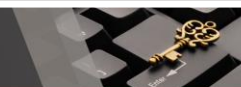
- 이분탐색은 코딩테스트 기준으로 굉장히 변별력 있는 알고리즘 중 하나입니다. 이분탐색 문제는 나올 확률이 그다지 높지 않고, 설령 이분탐색 문제가 나왔다고 하더라도 이를 풀지 못해서 당락이 바뀔 가능성은 거의 없습니다.
- 그러나 이분탐색 문제는 일단 제대로 익히고 나면 구현이 그다지 어렵지 않은 편이기 때문에 코딩테스트에서 굉장한 우위에 설 수 있습니다.
- 그러므로 이전 강의 내용들을 잘 따라왔다면 이분탐색을 깊게 공부하시고, 그렇지 않다면 이번 강에서 개념만 적당히 익힌 채로 넘어가고 먼저 이전 강의 내용들을 완벽하게 숙지할 수 있도록 합시다.

0x00 이분탐색(Binary Search)



- 이분탐색은 정렬되어 있는 배열에서 특정 데이터를 찾기 위해 모든 데이터를 순차적으로 확인하는 대신 탐색 범위를 절반으로 줄여가며 찾는 탐색 방법입니다.
- 우리는 0x01강에서 이분탐색을 사용한 알고리즘을 본 적이 있습니다.

0x00 시간, 공간복잡도



- 시간복잡도 = 입력의 크기와 문제를 해결하는데 걸리는 시간의 상관관계.
- 문제 : 대회장에 N 명의 사람들이 일렬로 서있다. 거기서 당신은 이름이 '가나다' 인 사람을 찾기 위해 사람들에게 이름을 물어볼 것이다. 이 때 사람들은 이름순으로 서있다. 이름을 물어보고 대답을 듣는데까지 1초가 걸린다면 얼마만큼의 시간이 필요할까? (=시간복잡도가 얼마인가?)
- 답 : 엥대운게임을 하듯이 중간 사람에게 계속 물어보면 된다. 최악의 경우 $\lg N$ 초, 최선의 경우 1초, 평균적으로 대략 $\lg N$ 초가 필요하다. 걸리는 시간은 $\lg N$ 에 비례한다.

7


0x01강에서의 예시

0x01 예시 문제 1 : 수 찾기



- BOJ 1920번 : 수 찾기 문제를 풀어봅시다. 이 문제는 굉장히 정직하게 이분탐색을 할 것을 요구하고 있습니다. 미리 배열 A 를 정렬해둔 후 M 개의 수들에 대해 이분탐색을 수행하기만 하면 되는데 구현을 어떤 식으로 해야 할지 굉장히 막막할 것입니다.
- 구현은 각 수에 대해 해당 수가 있을 수 있는 A 내의 index의 범위를 계속 절반으로 줄이는 방식으로 구현됩니다. 같이 해봅시다.

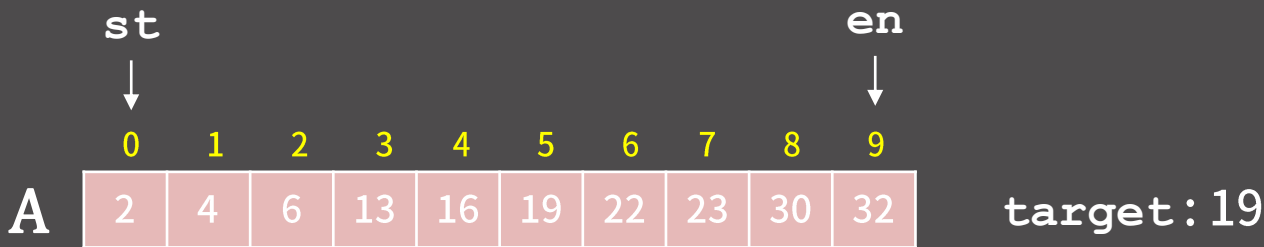
0x01 예시 문제 1 : 수 찾기



	0	1	2	3	4	5	6	7	8	9	
A	2	4	6	13	16	19	22	23	30	32	target : 19

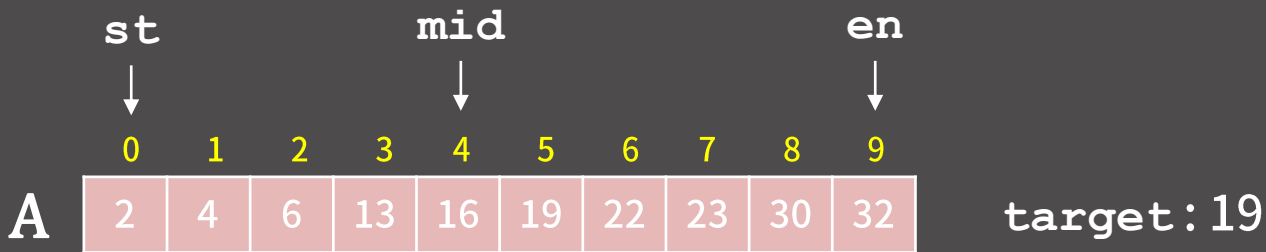
-
- A는 이미 정렬되어 있는 상태라고 가정합니다. 원래 문제에서는 A에 19가 있는지만 판단하면 되지만, 더 확장해 19가 있는 인덱스를 반환하는 함수를 만들도록 하겠습니다.

0x01 예시 문제 1 : 수 찾기



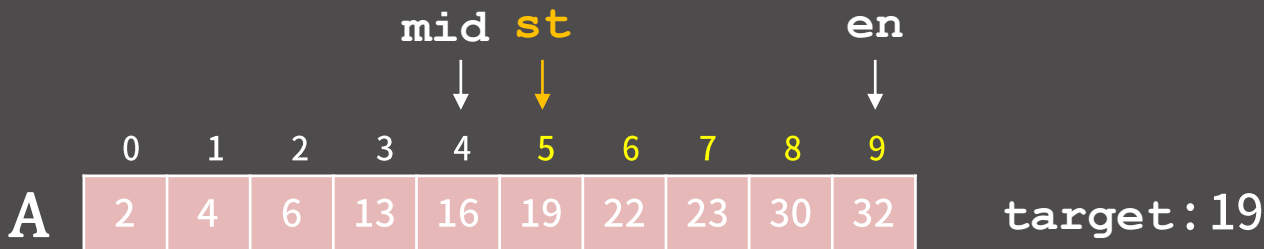
- st와 en은 target이 있는 인덱스로 가능한 범위를 나타냅니다. 만약 A 내에 target이 있다면 0에서 9사이임은 자명합니다. 이 범위는 계속해서 줄어들 예정입니다.

0x01 예시 문제 1 : 수 찾기




- $mid = (st + en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 target의 값을 비교해 **st** 혹은 **en**을 적절하게 바꿀 것입니다.

0x01 예시 문제 1 : 수 찾기



- 비교 결과 $A[mid] < target$ 입니다. 이 결과를 통해 적어도 mid 이하의 인덱스에는 $target$ 이 있을 수 없음을 알 수 있고, 이는 곧 $st = mid + 1$ 로 변경이 가능함을 의미합니다.

0x01 예시 문제 1 : 수 찾기



	0	1	2	3	4	st ↓ 5	6	mid ↓ 7	8	en ↓ 9	
A	2	4	6	13	16	19	22	23	30	32	target : 19

-
- 다시 $mid = (st+en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 $target$ 의 값을 비교해 st 혹은 en 을 적절하게 바꿀 것입니다.

0x01 예시 문제 1 : 수 찾기



	0	1	2	3	4	st ↓ 5	en ↓ 6	mid ↓ 7	8	9	
A	2	4	6	13	16	19	22	23	30	32	target: 19

- 비교 결과 $A[mid] > target$ 입니다. 이 결과를 통해 적어도 mid 이상의 인덱스에는 $target$ 이 있을 수 없음을 알 수 있고, 이는 곧 $en = mid - 1$ 로 변경이 가능함을 의미합니다.

0x01 예시 문제 1 : 수 찾기



- 다시 $mid = (st+en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 $target$ 의 값을 비교해 st 혹은 en 을 적절하게 바꿀 것입니다.

0x01 예시 문제 1 : 수 찾기



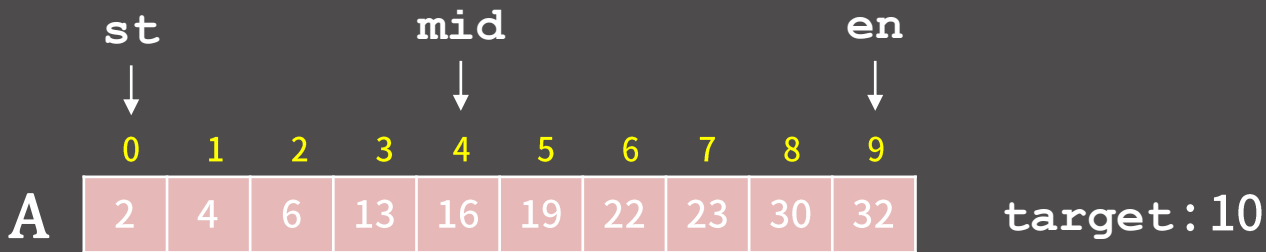
- 비교 결과 $A[mid] = target$ 입니다. $target$ 이 들어있는 인덱스를 찾았으므로 mid 의 값을 반환하고 과정을 종료합니다.

0x01 예시 문제 1 : 수 찾기



- 이 과정을 코드로 옮기기 전, A 내에 target이 없는 경우에는 어떻게 되는지도 확인해봅시다. 이전과 마찬가지로 st와 en은 target이 있는 인덱스로 가능한 범위를 나타냅니다. 만약 A 내에 target이 있다면 0에서 9사이임은 자명합니다. 이 범위는 계속해서 줄어들 예정입니다.

0x01 예시 문제 1 : 수 찾기



- $mid = (st + en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 **target**의 값을 비교해 **st** 혹은 **en**을 적절하게 바꿀 것입니다.

0x01 예시 문제 1 : 수 찾기



- 비교 결과 $A[mid] > target$ 입니다. 이 결과를 통해 적어도 mid 이상의 인덱스에는 $target$ 이 있을 수 없음을 알 수 있고, 이는 곧 $en = mid - 1$ 로 변경이 가능함을 의미합니다.


0x01 예시 문제 1 : 수 찾기



	st	mid	en										
	↓	↓	↓										
	0	1	2	3	4	5	6	7	8	9			
A	2	4	6	13	16	19	22	23	30	32		target: 10	

-
- 다시 $mid = (st+en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 $target$ 의 값을 비교해 st 혹은 en 을 적절하게 바꿀 것입니다.

0x01 예시 문제 1 : 수 찾기



	mid	st	en								
	↓	↓	↓								
	0	1	2	3	4	5	6	7	8	9	
A	2	4	6	13	16	19	22	23	30	32	target: 10

-
- 비교 결과 $A[mid] < target$ 입니다. 이 결과를 통해 적어도 mid 이하의 인덱스에는 $target$ 이 있을 수 없음을 알 수 있고, 이는 곧 $st = mid + 1$ 로 변경이 가능함을 의미합니다.

0x01 예시 문제 1 : 수 찾기



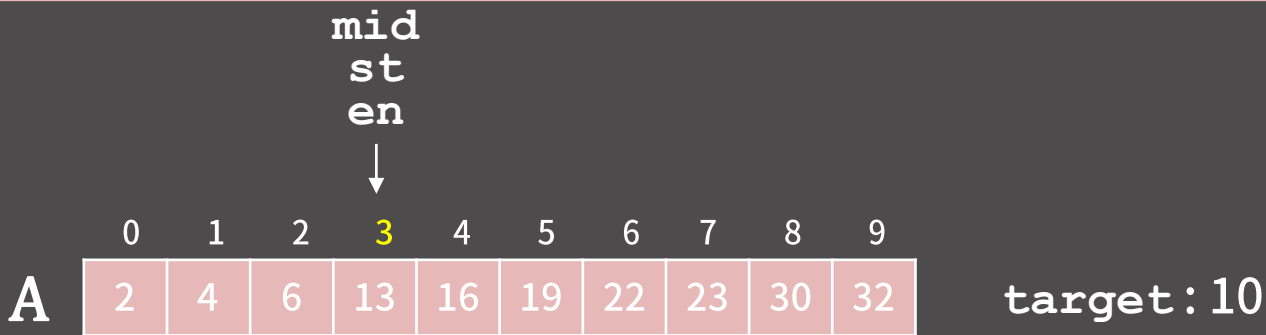
- 다시 $mid = (st + en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 `target`의 값을 비교해 st 혹은 en 을 적절하게 바꿀 것입니다.

0x01 예시 문제 1 : 수 찾기



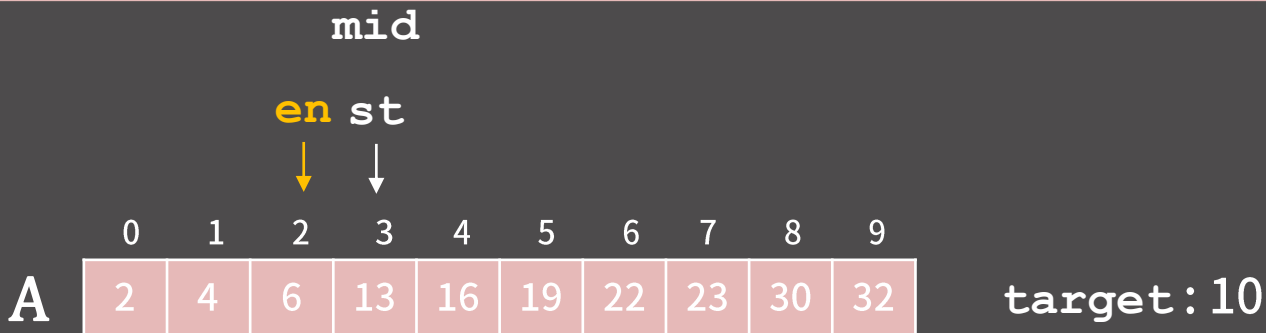
- 비교 결과 $A[mid] < target$ 입니다. 이 결과를 통해 적어도 mid 이하의 인덱스에는 $target$ 이 있을 수 없음을 알 수 있고, 이는 곧 $st = mid + 1$ 로 변경이 가능함을 의미합니다.

0x01 예시 문제 1 : 수 찾기



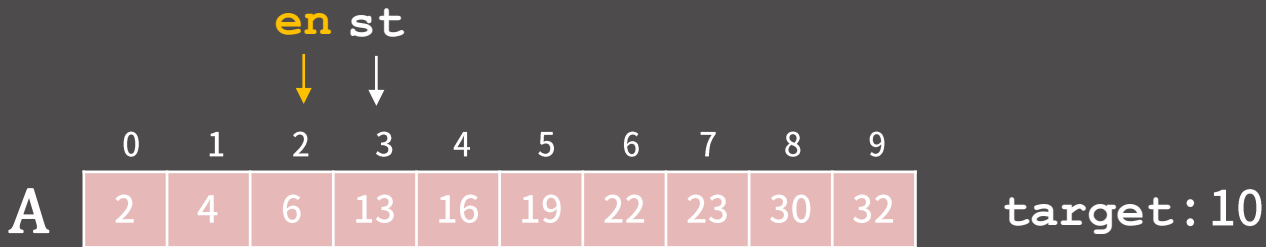
- 다시 $mid = (st + en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 `target`의 값을 비교해 `st` 혹은 `en`을 적절하게 바꿀 것입니다.

0x01 예시 문제 1 : 수 찾기



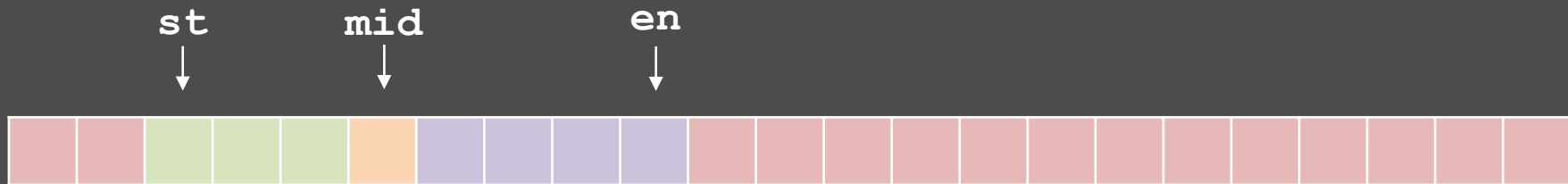
- 비교 결과 $A[mid] > target$ 입니다. 이 결과를 통해 적어도 `mid` 이상의 인덱스에는 `target`이 있을 수 없음을 알 수 있고, 이는 곧 `en = mid - 1` 로 변경이 가능함을 의미합니다.

0x01 예시 문제 1 : 수 찾기



- **st**와 **en**이 역전되었습니다. 이는 곧 **target**과 일치하는 인덱스가 존재하지 않음을 의미합니다. -1을 반환하고 과정을 종료합니다.

0x01 예시 문제 1 : 수 찾기



- `A[mid]`와 `target`의 대소비교 결과에 따라 구간이 어떻게 줄어드는지 보겠습니다.
- `A[mid] > target`일 때는 초록색 구간으로, `en = mid-1`이 됩니다.
- `A[mid] = target`일 때는 주황색 구간으로, `st = en = mid`이 됩니다. 사실 굳이 `st`, `en`를 바꿀 필요 없이 바로 `mid`를 반환하면 됩니다.
- `A[mid] < target`일 때는 보라색 구간으로, `st = mid+1`이 됩니다.

0x01 예시 문제 1 : 수 찾기



```
int a[10] = {2, 4, 6, 13, 16, 19, 22, 23, 30, 32};
int len = 10;
int BinarySearch(int target, int len){
    int st = 0;
    int en = len-1;
    while(st <= en){
        int mid = (st+en)/2;
        if(a[mid] < target)
            st = mid+1;
        else if(a[mid] > target)
            en = mid-1;
        else
            return mid;
    }
    return -1; // st > en일 경우 while문을 탈출함
}
```

- 설명한 것 그대로 `BinarySearch` 함수를 구현하면 됩니다. 구간의 범위가 매 반복문마다 계속 절반으로 줄어드므로 $O(\lg N)$ 임을 알 수 있습니다.
- 정답 코드 : <http://boj.kr/243e43ba8fdf41fe8f4d8540d59537fb>

0x02 예시 문제 2 : 숫자 카드 2



- 현재는 배열에 `target`이 여러 번 들어있을 경우, 아무 인덱스나 반환하게 됩니다. 만약 여러 번 들어있을 때 제일 왼쪽의 인덱스 혹은 제일 오른쪽의 인덱스를 반환해야 하면 어떻게 할까요?
- 더 나아가 값을 정확히 찾는 것이 아닌, `target`이 삽입되어도 오름차순 순서가 유지되는 제일 왼쪽/오른쪽의 인덱스를 찾는 문제를 생각해봅시다. 다음 장의 그림을 보고 정확한 의미를 파악해보세요.

0x02 예시 문제 2 : 숫자 카드 2



0	1	2	3	4	5	6	7	8	9
2	7	11	11	16	19	22	22	22	x

↑
11



0	1	2	3	4	5	6	7	8	9
2	7	11	11	11	16	19	22	22	22

모순이 생기지 않음

0	1	2	3	4	5	6	7	8	9
2	7	11	11	16	19	22	22	22	x

↑
30



0	1	2	3	4	5	6	7	8	9
2	7	11	11	16	19	22	22	22	30

모순이 생기지 않음

0	1	2	3	4	5	6	7	8	9
2	7	11	11	16	19	22	22	22	x


↑
11



0	1	2	3	4	5	6	7	8	9
2	7	11	11	16	19	11	22	22	22

모순이 생김

0x02 예시 문제 2 : 숫자 카드 2



0	1	2	3	4	5	6	7	8	9
2	7	11	11	16	19	22	22	22	x

↑
11

0	1	2	3	4	5	6	7	8	9
2	7	11	11	11	16	19	22	22	22


- 위의 예시에서 11은 2, 3, 4번째 인덱스에 들어가더라도 모순이 생기지 않습니다. 즉 11이 삽입되어도 오름차순 순서가 유지되는 제일 왼쪽의 인덱스는 2이고 제일 오른쪽의 인덱스는 4입니다.
- 이를 어떻게 구할 수 있을지, 그리고 제일 왼쪽/오른쪽의 인덱스는 어떤 것과 연관이 있을지를 한 번 고민해보세요.

0x02 예시 문제 2 : 숫자 카드 2



- 이것 또한 이분탐색으로 해결할 수 있습니다. 더 나아가 오름차순 순서가 유지되는 **제일 왼쪽/오른쪽의 인덱스의 차이가 해당 배열 내에 target의 등장 횟수**입니다. 직접 앞의 슬라이드를 참고해서 고민해보세요.
- BOJ 10816번: 숫자 카드 2 문제를 봅시다. 제일 왼쪽과 오른쪽의 인덱스를 구하고 그 차이를 계산하면 그 수가 몇 번 적혔는지를 알 수 있습니다. 마찬가지로 `st`와 `en`을 이용한 이분탐색으로 이를 구할 수 있습니다.
- 우선 제일 왼쪽의 인덱스를 같이 구해봅시다.

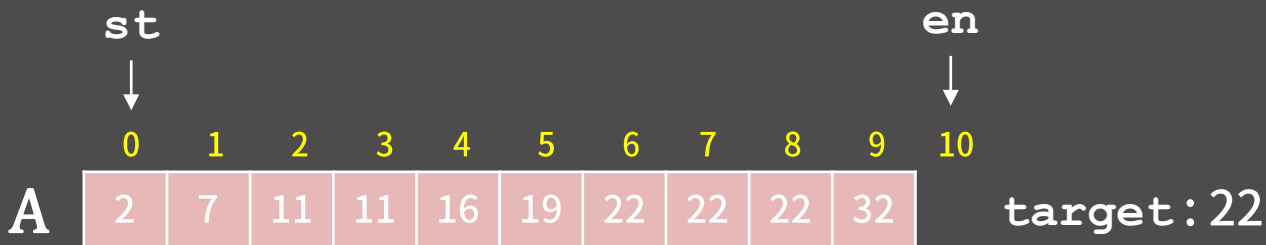
0x02 예시 문제 2 : 숫자 카드 2



	0	1	2	3	4	5	6	7	8	9	
A	2	7	11	11	16	19	22	22	22	32	target : 22

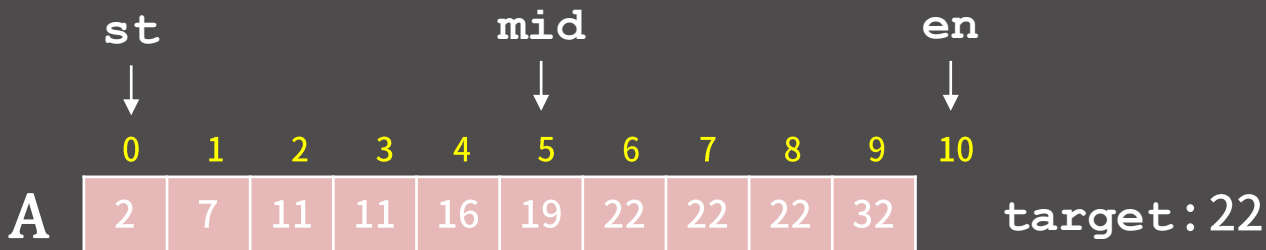
-
- A는 이미 정렬되어 있는 상태라고 가정합니다. A에 22가 들어가고도 모순이 생기지 않는 인덱스 중에서 가장 왼쪽의 것을 반환하는 함수를 만들어봅시다.

0x02 예시 문제 2 : 숫자 카드 2



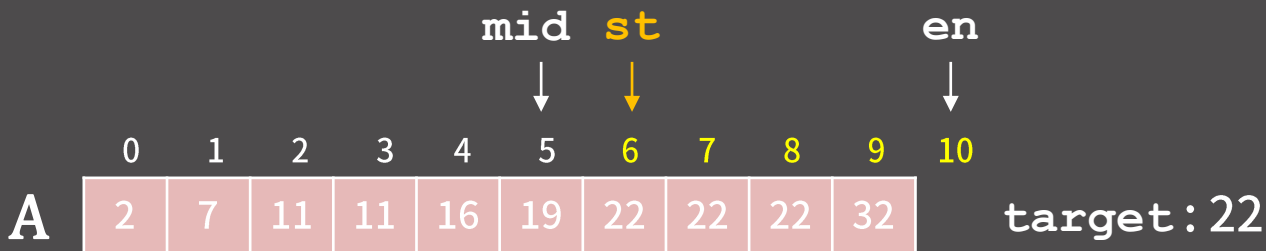
- st와 en은 target이 있는 모순이 생기지 않는 인덱스로 가능한 범위를 나타냅니다.
en이 9가 아니라 **10임에 주의**합니다. 이 범위는 계속해서 줄어들 예정입니다.

0x02 예시 문제 2 : 숫자 카드 2




- $mid = (st + en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 target의 값을 비교해 st 혹은 en을 적절하게 바꿀 것입니다.

0x02 예시 문제 2 : 숫자 카드 2



- 비교 결과 $A[mid] < target$ 입니다. 이 결과를 통해 target은 mid보다 더 큰 인덱스에 삽입되어야 함을 알 수 있고, 이는 곧 $st = mid + 1$ 로 변경이 가능함을 의미합니다.

0x02 예시 문제 2 : 숫자 카드 2



	0	1	2	3	4	5	st ↓ 6	7	mid ↓ 8	9	en ↓ 10	
A	2	7	11	11	16	19	22	22	22	32		target: 22

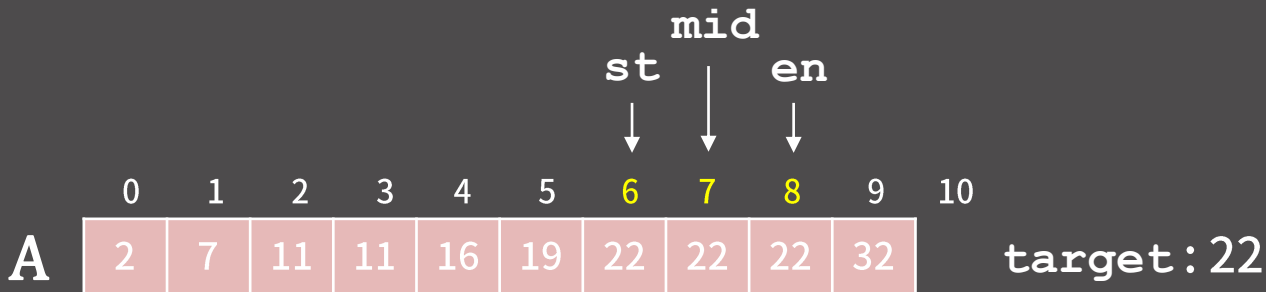
-
- 다시 $mid = (st+en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 target의 값을 비교해 st 혹은 en을 적절하게 바꿀 것입니다.

0x02 예시 문제 2 : 숫자 카드 2



- 비교 결과 $A[mid] = \text{target}$ 입니다. 이 결과를 통해 target은 mid에 삽입이 가능함을 알 수 있습니다. 그런데 저희는 모순이 생기지 않는 인덱스 중에서 가장 왼쪽의 것을 찾고 있으므로, 여기서 mid를 반환하고 끝내면 안됩니다. 대신 그 인덱스는 mid 이하에 존재한다는 사실을 기록하고 다음 단계를 진행해야 합니다. 이는 곧 $en = mid$ 로 변경이 가능함을 의미합니다.

0x02 예시 문제 2 : 숫자 카드 2



- 다시 $mid = (st+en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 target의 값을 비교해 st 혹은 en 을 적절하게 바꿀 것입니다.

0x02 예시 문제 2 : 숫자 카드 2



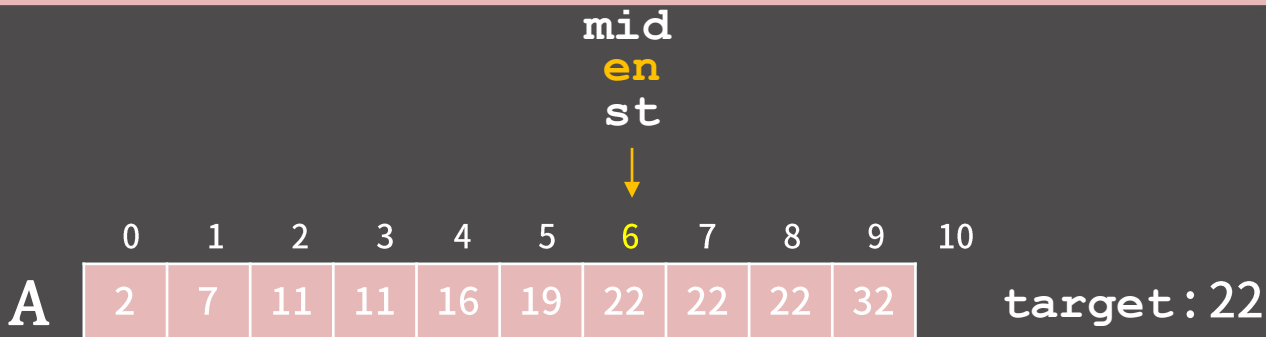
- 비교 결과 `A[mid] = target` 입니다. 앞에서 한 것과 마찬가지로 `en = mid`로 변경이 가능함을 의미합니다.

0x02 예시 문제 2 : 숫자 카드 2



- 다시 $mid = (st+en) / 2$ 로 계산합니다. 우리는 $A[mid]$ 와 target의 값을 비교해 st 혹은 en 을 적절하게 바꿀 것입니다.

0x02 예시 문제 2 : 숫자 카드 2



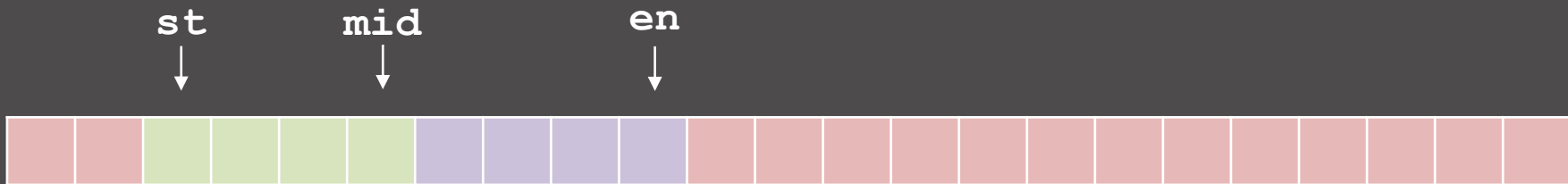
- 비교 결과 $A[mid] = target$ 입니다. 앞에서 한 것과 마찬가지로 $en = mid$ 로 변경이 가능함을 의미합니다.

0x02 예시 문제 2 : 숫자 카드 2



- `st = en`이 되어 후보군이 1개로 고정되었습니다. `st`를 반환하고 과정을 종료합니다.

0x02 예시 문제 2 : 숫자 카드 2



- `A[mid]`와 `target`의 대소비교 결과에 따라 구간이 어떻게 줄어드는지 보겠습니다.
- `A[mid] >= target`일 때는 초록색 구간으로, `en = mid`이 됩니다.
- `A[mid] < target`일 때는 보라색 구간으로, `st = mid+1`이 됩니다.

0x02 예시 문제 2 : 숫자 카드 2



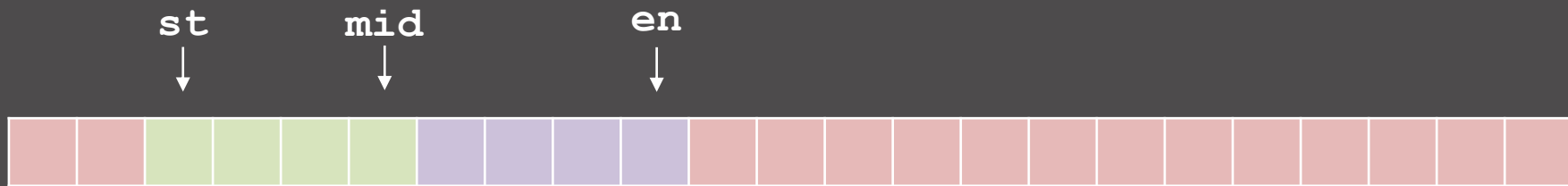
```
int lower_idx(int target, int len){
    int st = 0;
    int en = len;
    while(st < en){
        int mid = (st+en)/2;
        if(a[mid] >= target) en = mid;
        else st = mid+1;
    }
    return st; // st = en으로 가능한 후보가 1개로 확정될 경우 while문을 탈출함
}
```

- 설명한 것 그대로 제일 왼쪽의 인덱스를 구하는 `lower_idx` 함수를 구현하면 됩니다. 구간의 범위가 매 반복문마다 계속 절반으로 줄어드므로 $O(\lg N)$ 임을 알 수 있습니다.

0x02 예시 문제 2 : 숫자 카드 2



- 제일 오른쪽의 인덱스를 구하는 과정도 비슷하게 진행됩니다. 자세한 설명은 생략하고 마찬가지로 $A[mid]$ 와 $target$ 의 대소비교 결과에 따라 구간이 어떻게 줄어드는지 보겠습니다. 참고로 $lower_idx$ 함수와 $A[mid] = target$ 일 때만 다릅니다.



- $A[mid] > target$ 일 때는 초록색 구간으로, $en = mid$ 이 됩니다.
- $A[mid] \leq target$ 일 때는 보라색 구간으로, $st = mid+1$ 이 됩니다.

0x02 예시 문제 2 : 숫자 카드 2



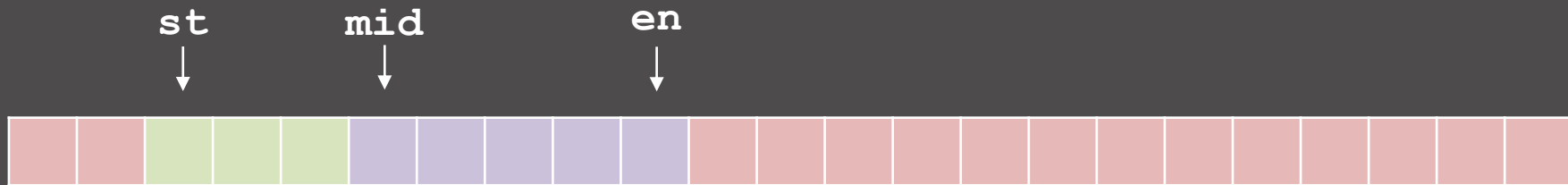
```
int upper_idx(int target, int len){
    int st = 0;
    int en = len;
    while(st < en){
        int mid = (st+en)/2;
        if(a[mid] > target) en = mid;
        else st = mid+1;
    }
    return st; // st = en으로 가능한 후보가 1개로 확정될 경우 while문을 탈출함
}
```

- 설명한 것 그대로 제일 오른쪽의 인덱스를 구하는 `upper_idx` 함수를 구현하면 됩니다. 마찬가지로 구간의 범위가 매 반복문마다 계속 절반으로 줄어드므로 $O(\lg N)$ 임을 알 수 있습니다.
- 정답 코드 : <http://boj.kr/a43f7a73796341babad1914461075d77>

0x03 주의사항



- 증가수열에서 `target`보다 값이 작은 원소 중에 가장 오른쪽에 있는 원소의 위치를 찾는 문제를 생각해봅시다. 예를 들어 배열이 `(-4, 1, 3, 4, 5, 6, 6)`이고 `target`이 4일 경우 3의 인덱스인 2를 반환합니다.
- 이 위치는 사실 `lower_bound` 함수의 반환값에서 한 칸 왼쪽으로 가면 되지만, 직접 이분탐색으로 구현해봅시다.



- $A[mid] < target$ 일 때는 보라색 구간으로, $st = mid$ 이 됩니다.
- $A[mid] \geq target$ 일 때는 초록색 구간으로, $en = mid - 1$ 이 됩니다.

0x03 주의사항



- 논리적으로 문제가 없기 때문에 이때까지 구현한 것과 비슷하게 구현을 하면 될 것 같지만, 코드로 짜서 실제로 실행해보면 일부 값에 대해 무한루프에 빠집니다.

```
int a[10] = {2,7,11,11,16,19,22,22,22,32};
int len = 10;
int func(int target, int len){
    cout << "target : " << target << '\n';
    int st = -1;
    int en = len-1;
    while(st < en){
        int mid = (st+en)/2;
        if(a[mid] < target) st = mid;
        else en = mid-1;
        cout << st << ' ' << en << '\n';
    }
    return st; // st = en으로 가능한 후보가 1개로 확정될 경우 while문을 탈출함
}
```

```
target : 16
-1 3
1 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
```

0x03 주의사항

- `st = 2`이고 `en = 3`일 때 `mid = 2`이고, `a[2] (=11) < target (=16)` 이기에 `st`는 변함없이 2가 되어 무한루프에 빠지게 됨을 알 수 있습니다.

```
int a[10] = {2,7,11,11,16,19,22,22,22,32};
int len = 10;
int func(int target, int len){
    cout << "target : " << target << '\n';
    int st = -1;
    int en = len-1;
    while(st < en){
        int mid = (st+en)/2;
        if(a[mid] < target) st = mid;
        else en = mid-1;
        cout << st << ' ' << en << '\n';
    }
    return st; // st = en으로 가능한 후보가 1개로 확정될 경우 while문을 탈출함
}
```

```
target : 16
-1 3
1 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
2 3
```

0x03 주의사항

- 지금 이 함수에서 이렇게 무한루프가 돌지 않게끔 하기 위해서는 `st`와 `en`이 1 차이날 때 `mid`가 `st`대신 `en`이 되게끔 해야 합니다. 즉, $mid = (st+en+1)/2$ 로 두면 됩니다.

```
int a[10] = {2,7,11,11,16,19,22,22,22,32};
int len = 10;
int func(int target, int len){
    cout << "target : " << target << '\n';
    int st = -1;
    int en = len-1;
    while(st < en){
        int mid = (st+en+1)/2;
        if(a[mid] < target) st = mid;
        else en = mid-1;
        cout << st << ' ' << en << '\n';
    }
    return st; // st = en으로 가능한 후보가 1개로 확정될 경우 while문을 탈출함
}
```

```
target : 16
-1 3
1 3
2 3
3 3
```


0x03 주의사항



- $(st = mid / en = mid-1)$ 일 때에는 $mid = (st+en+1)/2$,
 $(st = mid+1 / en = mid)$ 일 때에는 $mid = (st+en)/2$ 으로 두어야
무한루프를 방지할 수 있습니다. 꼭 이것을 공식처럼 외우지 않더라도 직접 st 와
 en 이 1 차이나는 경우를 생각해보면 판단할 수 있습니다.
- 그러나 매번 구현시마다 이것을 고려하기가 싫으면, 그냥 $en-st$ 가 어느 정도 이하로
줄어들면 그 안에서는 선형으로 탐색하도록 구현을 할 수도 있습니다. 단, 범위 내에서
제일 왼쪽(=최소)을 찾아야 하는지, 제일 오른쪽(=최대)를 찾아야 하는지에 따라
 st 부터 1씩 증가하며 살펴볼지, en 부터 1씩 감소하면서 살펴볼지를 정해야 합니다.

0x03 주의사항

- st와 en이 3 이하로 차이나면 반복문을 종료하고 st와 en사이의 모든 index에 대해 순차적으로 조건을 만족하는지 확인합니다.

```
int a[10] = {2,7,11,11,16,19,22,22,22,32};
int len = 10;
int func(int target, int len){
    cout << "target : " << target << '\n';
    int st = -1;
    int en = len-1;
    while(en-st < 3){
        int mid = (st+en)/2;
        if(a[mid] < target) st = mid;
        else en = mid-1;
        cout << st << ' ' << en << '\n';
    }
    for(int i = st; i <= en; i++){ // st ~ en 사이의 모든 index에 대해 확인
        if((i==-1 or a[i] < target) and (i==len-1 or a[i+1] >= target))
            return i;
    }
    return -2; // st에서 en 사이에 반드시 답이 있으므로 여기는 절대 도달하지 않는다.
}
```

0x04 관련 STL



- Binary Search의 구현이 엄청 어려운 것은 아니지만, 실수할 여지가 있습니다. 하지만 STL을 활용하면 Binary Search를 직접 구현할 필요가 없어집니다.
- `binary_search`, `lower_bound`, `upper_bound` 함수가 존재합니다. 이 세 함수는 **오름차순으로 정렬되어 있는 배열/vector에서만 정상 작동하는 함수**입니다. 각 함수에 대해 알아보시다.

0x04 관련 STL



binary_search

- `binary_search` 함수는 주어진 범위 내에 원소가 들어있는지 여부를 $O(\lg N)$ 에 `true` 혹은 `false`로 반환해주는 함수입니다. 만약 그 범위가 오름차순으로 정렬되어 있지 않다면 실제로는 들어있으나 들어있지 않다고 판단할 가능성이 있습니다.

```
int arr[5] = {1,2,3,4,6};  
if(binary_search(arr,arr+5,5))  
    cout << "5 in arr\n";  
else  
    cout << "5 not in arr\n";  
  
vector<int> vec = {5,6,10,20};  
if(binary_search(vec.begin(),vec.end(),10))  
    cout << "10 in vec\n";  
else  
    cout << "10 not in vec\n";
```

5 not in arr
10 in vec

0x04 관련 STL



binary_search

- 이 함수를 이용하면 BOJ 1920번 : 수 찾기 문제를 더 간단하게 해결할 수 있습니다.
- 정답 코드 : <http://boj.kr/6c835c9daf6f4caf835a4a6bb889032b>

```
int arr[5] = {1,2,3,4,6};
if(binary_search(arr,arr+5,5))
    cout << "5 in arr\n";
else
    cout << "5 not in arr\n";

vector<int> vec = {5,6,10,20};
if(binary_search(vec.begin(),vec.end(),10))
    cout << "10 in vec\n";
else
    cout << "10 not in vec\n";
```

5 not in arr
10 in vec

0x04 관련 STL



`lower_bound, upper_bound`

- `lower_bound, upper_bound`는 각각 `target`이 삽입되어도 오름차순 순서가 유지되는 제일 왼쪽/오른쪽의 인덱스를 반환합니다. 앞에서 같이 구현한 `lower_idx, upper_idx` 함수와 거의 동일합니다. 단, `lower_idx/upper_idx`와는 다르게 `lower_bound/upper_bound`는 배열을 넘겨줄 경우 포인터를 반환하고 `vector`일 경우 `iterator`를 반환합니다.

0x04 관련 STL



`lower_bound, upper_bound`

- `lower_idx/upper_idx` 함수를 구현할 때 언급했듯 동일한 `target`에 대해 두 인덱스의 차이가 곧 `target`이 배열 안에 들어있는 횟수입니다. BOJ 10816번: 숫자 카드 2 문제를 STL을 이용해 풀어보세요.
- 정답 코드 : <http://boj.kr/61313b3eaa6a4f37ae806a43862d80e9>

0x04 관련 STL



- 이외에도 `equal_range` 라고 `lower_bound`, `upper_bound`의 결과를 `pair`로 반환해주는 함수도 있습니다.
- `lower_bound`, `upper_bound`가 무엇을 반환하는지, 어떻게 사용하는지 헛갈릴 수도 있지만 익혀두면 굉장히 유용하니 연습문제들을 풀면서 손에 익히는 것을 추천드립니다.

강의 정리



- 이분탐색의 구현법, 주의사항, 그리고 STL의 사용법을 익혔습니다.
- Parametric Search라는, 매개 변수의 값을 가지고 이분탐색을 수행해 주어진 문제를 decision problem으로 변환해 시간복잡도를 떨구는 기법도 있으나 코딩테스트 대비용으로는 너무 난이도가 높은 것 같아 배제했습니다. 단, 다른 것을 다 공부하고도 시간이 남는다면 공부해보세요.
- 연습문제가 꽤 어렵습니다. 코딩테스트 수준을 넘어서 깊은 사고를 필요로 하는 문제가 섞여있으니 풀리지 않는다고 너무 자책하지 마세요.