19차시

Stack 문제 풀이

문제

- 스택 2 BOJ 28278
- 에디터 BOJ 1406
- 단체줄넘기 BOJ 30457
- 오큰수 BOJ 17298

문제 풀이에 앞서

- 알고리즘을 적용시키는 눈을 기르자

- 지금 당장 문제를 풀지 못해도 된다
- 코드를 쓰는 능력은 정말 풀어본 문제의 수, 노력한 양에 비례
- 하지만 문제를 읽고 구상하는 것은 얼마든지 가능
- 문제를 풀지 못해도 어떤 방식으로 이 문제를 코딩할지 생각해보도록 하자

문제 풀이에 앞서

- 문제 풀이를 볼 때, 이런 식으로 자료구조를 사용할 수도 있다는 것을 깨우치는 것도 좋다
- 왜 그 자료구조를 사용해야 하는지, 무슨 특성을 보고 어떤 자료구조를 선택하는지 공부하는게 매우 중요하다

스택 2 BOJ 28278

- 스택 구현, 사용 문제
- std::stack 사용법 또는 스택 구현 및 사용을 연습해보자

에디터 BOJ 1406

- 16차시의 연결 리스트 문제, 어떻게 스택으로 해결이 가능할까?

에디터 BOJ 1406

- 커서의 왼쪽만 생각해보자.
- 일반적으로 글자는 왼쪽에서 오른쪽으로 쓴다
- 왼쪽에 있는 글자(커서에서 멀리 있는 쪽)는 오래된 글자
- 오른쪽에 있는 글자(커서에서 가까이 있는 쪽)는 최신 글자이다

에디터 BOJ 1406

- 커서를 왼쪽으로 옮긴다면, 가장 마지막에 들어간 데이터를 빼내야 하므로, 스택과 동일하다는 것을 알 수 있다
- 이를 이용해 커서를 왼쪽으로 옮기는 것을 커서 왼쪽에 있는 모든 글자를 스택에 넣어두고 스택에서 빼는 것으로 나타낼 수 있다

에디터 BOJ 1406

- 커서를 왼쪽으로 옮기는 것은 글자 입장에서는 커서의 왼쪽에 있던 글자가 오른쪽으로 이동하는 것
- 앞선 설명에서 이 내용을 스택에서 글자가 빠지는 것으로 구현할 수 있었다
- 반대로 커서를 오른쪽으로 이동하는 것은 제일 마지막에 뺐던 글자를 다시 가져오는 것
- 마지막에 다룬 데이터를 가져오므로 이 또한 스택과 일치한다는 것을 알 수 있다

에디터 BOJ 1406

- 이 둘을 조합하면 커서 기준으로 왼쪽과 오른쪽에 존재하는 글자를 스택으로 각각 관리 하면 커서를 왼쪽과 오른쪽으로 이동하는 것을 관리 할 수 있다

에디터 BOJ 1406

- abcd를 처음에 넣었다 생각하자
- 이는 왼쪽 스택에 a, b, c, d가 들어간 것

에디터 BOJ 1406

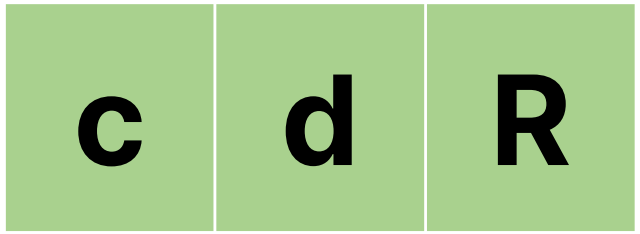
- 커서를 왼쪽으로 옮기는 것은 왼쪽 스택에서 빼서 오른쪽 스택으로 옮기는 것과 동일
- 커서를 오른쪽으로 옮기는 것은 오른쪽 스택에서 빼서 왼쪽 스택으로 옮기는 것과 동일

에디터 BOJ 1406

- 커서 왼쪽 이동

에디터 BOJ 1406

- 커서 왼쪽 이동



에디터 BOJ 1406

- 커서 왼쪽 이동

에디터 BOJ 1406

- 커서 오른쪽 이동

에디터 BOJ 1406

- 커서 오른쪽 이동

에디터 BOJ 1406

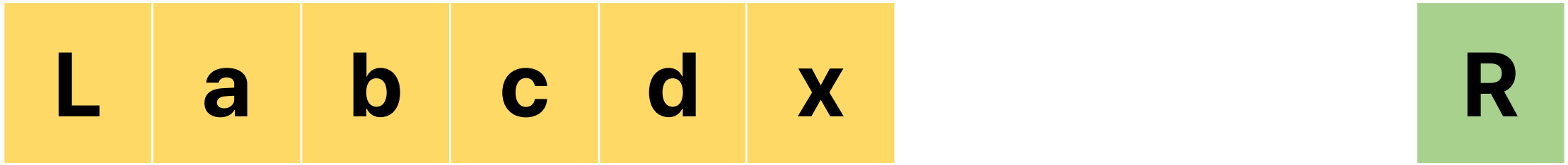
- 커서 오른쪽 이동
- 이동하려 할 때, 빼낼 스택이 비어있다면, 커서가 끝에 도달해 더 이상 움직이지 못하는 것을 의미

에디터 BOJ 1406

- 새로운 글자를 추가하는 것은 왼쪽 스택에 추가하는 것과 동일
- 글자를 제거하는 것은 왼쪽 스택에서 오른쪽 스택으로 이동하지 않고 제거하는 것과 동일

에디터 BOJ 1406

- x 삽입



에디터 BOJ 1406

- L 연산

에디터 BOJ 1406

- y 삽입

에디터 BOJ 1406

- 남은 글자는 왼쪽에서부터 읽으면 된다
- abcdyx

단체줄넘기 BOJ 30457

- N명의 학생들을 줄 세워야 한다
- 학생들은 왼쪽 또는 오른쪽을 보고 있음
- 모든 학생들은 자신이 바라보는 방향에 자기 보다 작은 학생들만 존재해야 한다
- $N \leq 1\,000$

단체줄넘기 BOJ 30457

- 내가 바라보는 방향에 나보다 작은 학생들만 있으면 되므로 키가 제일 큰 학생은 어떠한 제약도 존재하지 않는다
- 제일 큰 학생을 넣은 후, 학생을 한 명씩 넣어보자
- 제일 키가 큰 학생을 넣은 후 다른 학생들은 모두 제일 키가 큰 학생을 등지고 서야 한다

  제일 키가 큰 학생을 바라보는 경우, 나보다 키가 큰 학생이 존재하게 되므로

단체줄넘기 BOJ 30457

- 제일 키가 큰 학생을 기준으로 양 옆에 서로 양 끝을 바라보는 학생들이 반드시 오게 된다
- 그 이후에는 한 쪽 끝을 바라보고 있는 학생 앞에는 그 학생보다 키가 작은 학생이 와야 하며, 그 학생 또한 키가 큰 학생을 바라보지 못하므로, 같은 방향을 보게된다

- 계속하여 반복적으로 일어난다

단체줄넘기 BOJ 30457

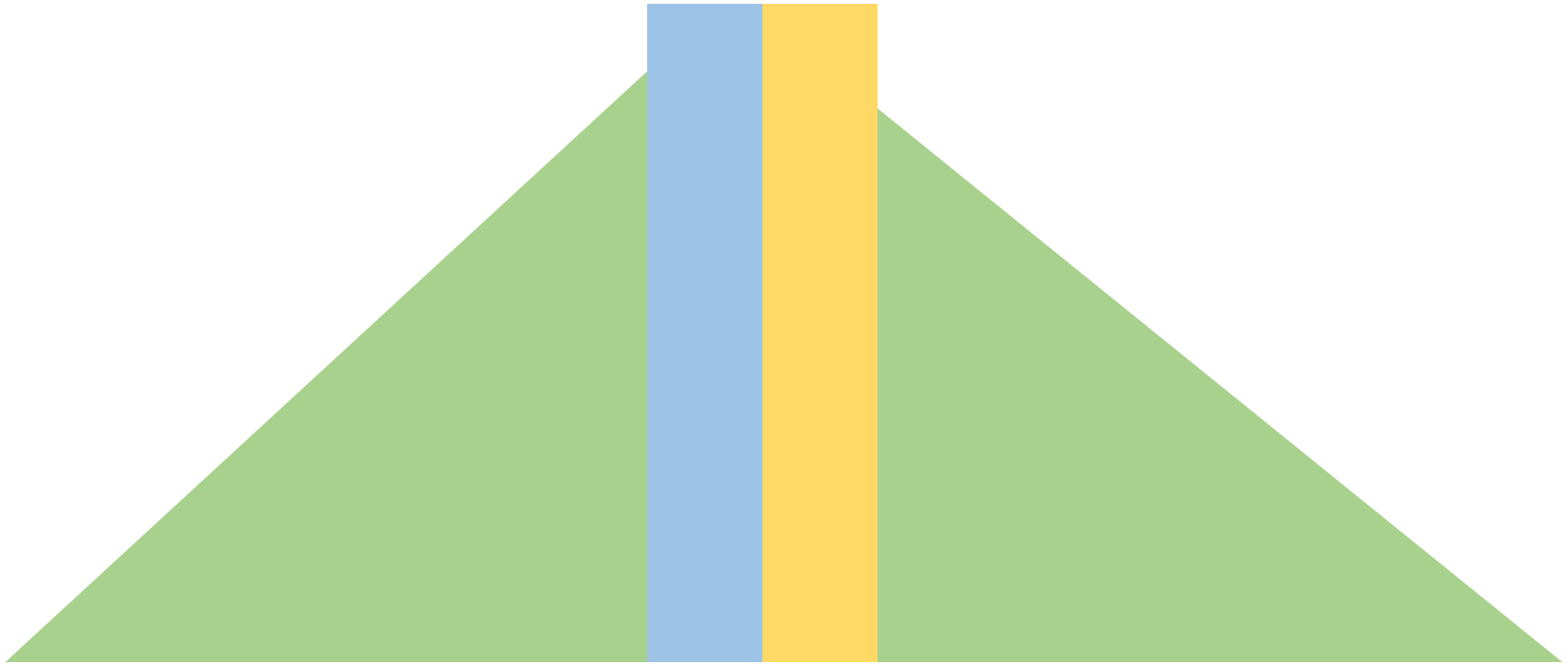
- 이를 이용하여 대략적인 모양이 삼각형으로 나오는 것을 알 수 있다

단체줄넘기 BOJ 30457

- 하지만 키가 동일한 학생도 존재한다
- 점차 키가 작은 학생이 들어가야 하지만 제일 키가 큰 학생은 한 쪽 방향만 바라보고 있다
- 키가 제일 큰 학생과 동일한 키를 가진 학생이 존재한다면, 두 학생이 서로 등지고 서게 할 수 있다

단체줄넘기 BOJ 30457

- 키가 제일 큰 친구를 두 명을 넣고 점차 작아지면 된다



단체줄넘기 BOJ 30457

- 제일 많은 학생을 넣어야 하므로 키가 제일 큰 학생부터 그 다음 큰 학생 순서로 순차적으로 넣어야 제일 많이 넣을 수 있다
- 정렬 후 키가 큰 친구부터 넣는다

단체줄넘기 BOJ 30457

- 큰 친구부터 넣으므로, 현재 넣을 학생의 키는 이전 학생의 키보다 작거나 동일하다
- 넣은 모든 학생과 비교할 필요 없이 마지막에 넣은 학생과 동일한지만 확인하면 된다
- 마지막에 넣은 데이터를 확인 -> 스택

단체줄넘기 BOJ 30457

- 왼쪽을 바라보는 학생들을 저장할 스택
- 오른쪽을 바라보는 학생들을 저장할 스택
- 이 두 개를 준비한다

단체줄넘기 BOJ 30457

- 학생들을 정렬한 후, 제일 키가 큰 학생 두 명을 각각 스택 하나에 한 명씩 넣는다
- 그리고 키가 큰 학생부터 살펴보며 왼쪽과 오른쪽 스택 중 한 곳에 넣는다.
- 두 스택 모두 들어갈 수 없는 경우 학생을 넣을 수 없는 경우이므로, 넣지 않는다

오큰수 BOJ 17298

- 수열이 주어질 때, k번째 오큰수는 수열의 k번째 숫자보다 오른쪽에 있으면서 k번째 숫자보다 큰 수 중에 제일 왼쪽에 있는 수를 찾는 것
- 그러한 수가 없는 경우 오큰수는 -1이다
- $N \leq 1\,000\,000$

오큰수 BOJ 17298

- 예제를 그려보자
- 예제 1

  3 5 2 7

  5 7 7 -1

오큰수 BOJ 17298

- 예제를 그려보자
- 예제 2

  9 5 4 8

  -1 8 8 -1

오큰수 BOJ 17298

- 오른쪽에서 나오는 수 중에서 제일 먼저 나오는 큰 수가 오큰수
- 오른쪽에서 나오므로 왼쪽부터 오른쪽으로 보자
- 내가 지금까지 나온 높이들을 기억하고 있다가 나보다 큰 수가 나오면 오큰수로 정하면 됨

오큰수 BOJ 17298

- 다시 정리하면
- 왼쪽부터 숫자를 보면서 기억함
- 나보다 큰 수가 오는 경우 오큰수가 확정됨
- 그 이후에 더 큰 수가 나오더라도 제일 왼쪽에 있는 수를 고르므로 영향을 주지 않음
- 나보다 작은 수가 나오는 경우 오큰수가 확정되지 않으므로 기다림

오큰수 BOJ 17298

- 3 5 2 7
- 3이 먼저 들어감

오큰수 BOJ 17298

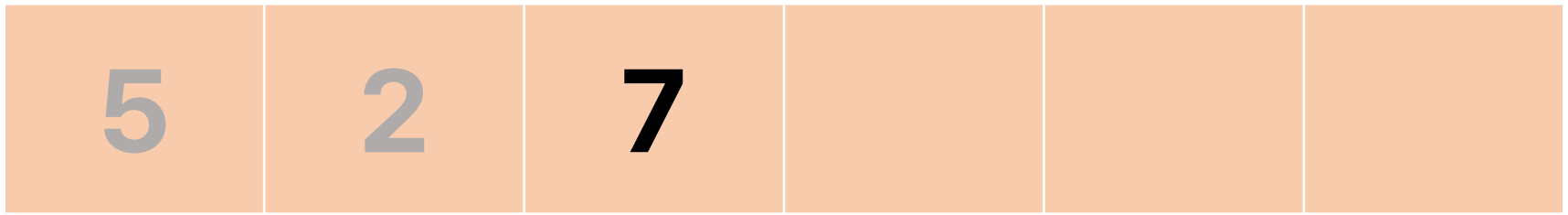
- 3 5 2 7
- 5가 들어가는 경우, 3보다 큰 수가 나오므로 3의 오큰수는 확정이 남

오큰수 BOJ 17298

- 3 5 2 7
- 2는 5보다 크지 않으므로 들어감

오큰수 BOJ 17298

- 3 5 2 7
- 7이 들어가는 경우, 7보다 작은 수들은 전부 오큰수가 정해지게 됨



오큰수 BOJ 17298

- 자료구조에는 작은 수가 나올때만 저장되고, 큰 수가 나오는 경우 다 나오게 된다
- 즉, 점차 작아지는 순서대로 (Monotone하게) 자료구조에 저장되게 된다

오큰수 BOJ 17298

- 마지막에 저장된 수가 제일 작고 점차 커지므로, 마지막에 넣은 수부터 현재 수보다 같거나 큰 수가 나올 때까지 자료구조에서 빼내가며 오큰수를 지정해주면 된다

- 스택을 사용하면 된다는 것을 의미

- 더 큰 예시를 가지고 해보자

오큰수 BOJ 17298

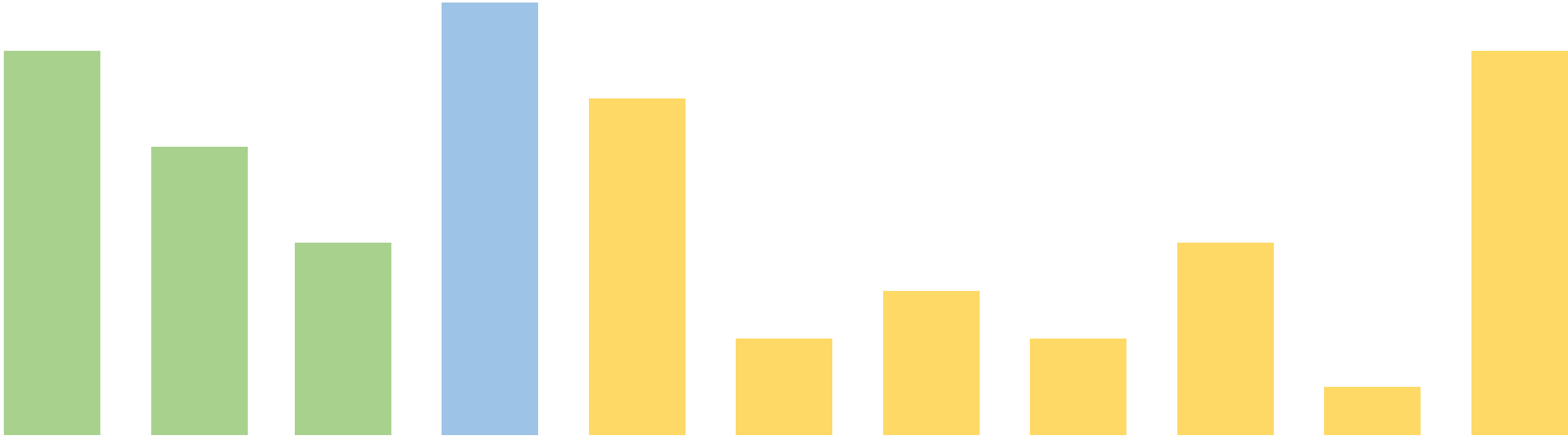
- 8 6 4 9 7 2 3 2 4 1 8

오큰수 BOJ 17298

- 나보다 큰 수가 나오는 경우 오큰수가 확정

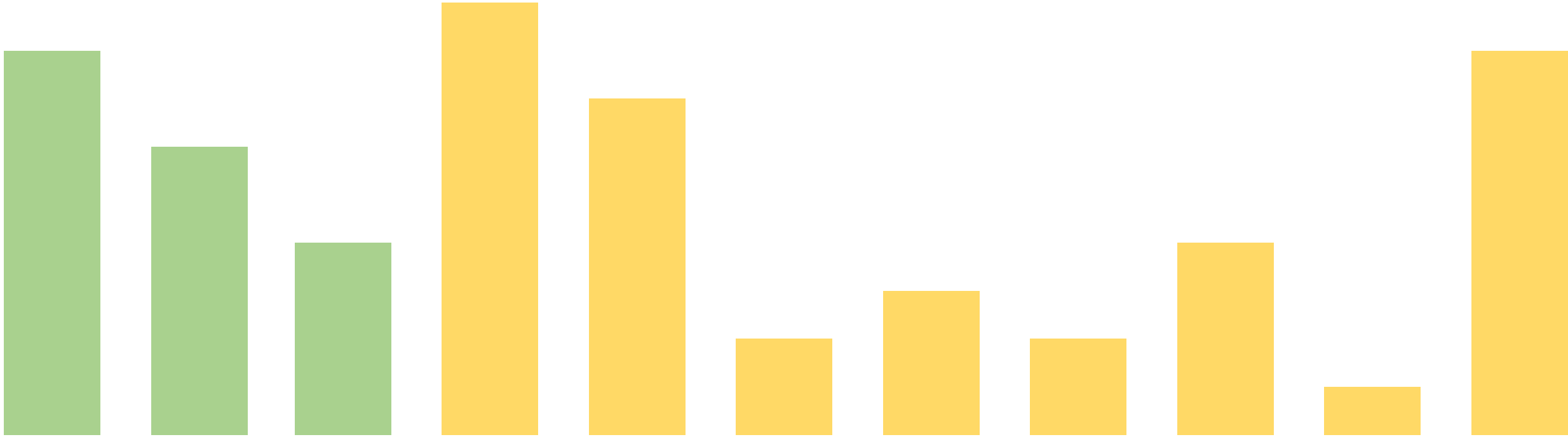
오큰수 BOJ 17298

- 초록색 숫자들은 파란색이 나오는 순간 모두 오큰수가 결정됨



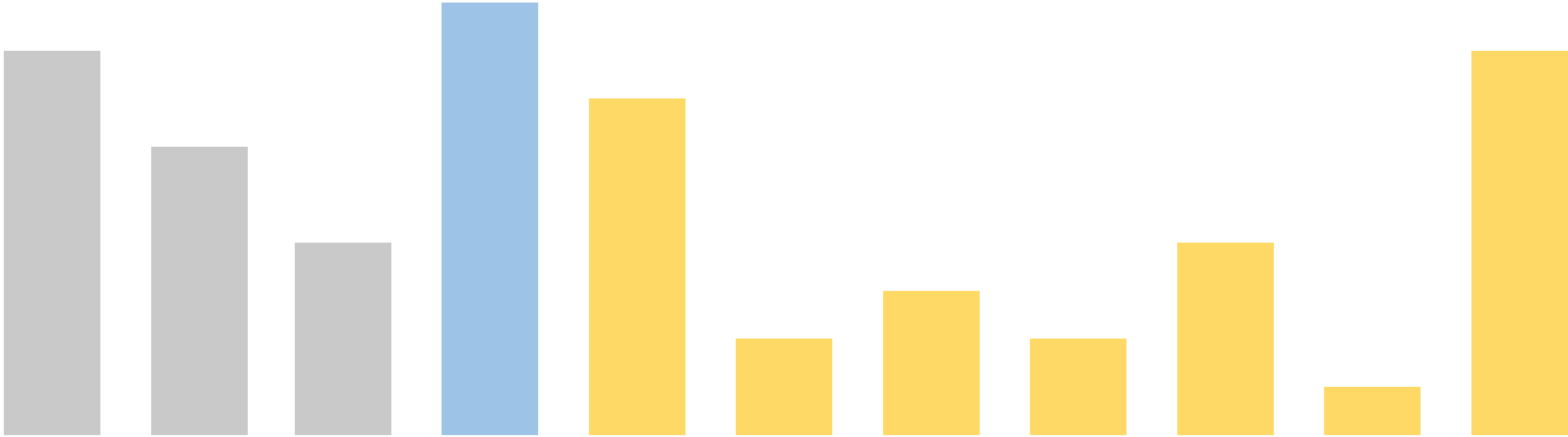
오큰수 BOJ 17298

- 오큰수가 지정되지 않은 애들을 모아보면 같거나 감소하는 형태이다(초록색)



오큰수 BOJ 17298

- 파란색이 누구의 오큰수인지를 확인해야한다



오큰수 BOJ 17298

- 매번 회색인 후보들을 모두 살펴보는 것은 손해
- 우리가 어떠한 자료구조를 사용해서 저장할 예정
- 저장한 데이터는 들어온 순서대로 점차 작아지는 형태를 보임(정렬 되어 있음)
- 정렬 되어 있으므로 작은 값부터 살펴보다 파란색보다 같거나 큰 값이 나온 경우, 더 이상 오큰수로 설정될 수 없음

오큰수 BOJ 17298

- 어떠한 자료구조를 사용해야하는가?
- 작은 수부터 꺼낼 자료구조
- 작은 수는 마지막에 저장된 요소임, Last In First Out

- 스택을 사용하면 문제를 풀 수 있다는 것을 알아낼 수 있다

오큰수 BOJ 17298

- 자료구조에서 관리할 값을 정의해야한다
- 자료구조에서 관리할 값: 아직 오큰수가 정해지지 않은 수

- 오큰수가 정해진다면 자료구조에서 빼내야한다

오큰수 BOJ 17298

- 원소를 왼쪽에서부터 순차적으로 본다
- 자료구조에 넣기 전에, 오큰수가 정해진 수를 빼야한다
- 오큰수가 정해진 수: 왼쪽에 있는 수들 중에서 오큰수가 정해져 있지 않으며, 현재 보는 값보다 작은 수

오큰수 BOJ 17298

- 자료구조에서 작은 값부터 보면서 오큰수가 정해지는 수들을 빼낸다
- 자료구조에는 현재 값보다 같거나 큰 수만 남는다

- 따라서 자료구조에서 먼저 넣은 수는 항상 마지막에 넣은 수보다 크거나 같다
- Monotone하다

오큰수 BOJ 17298

- 스택을 이용해 다시 한번 풀어보자
- 8 6 4 9 7 2 3 2 4 1 8

오큰수 BOJ 17298

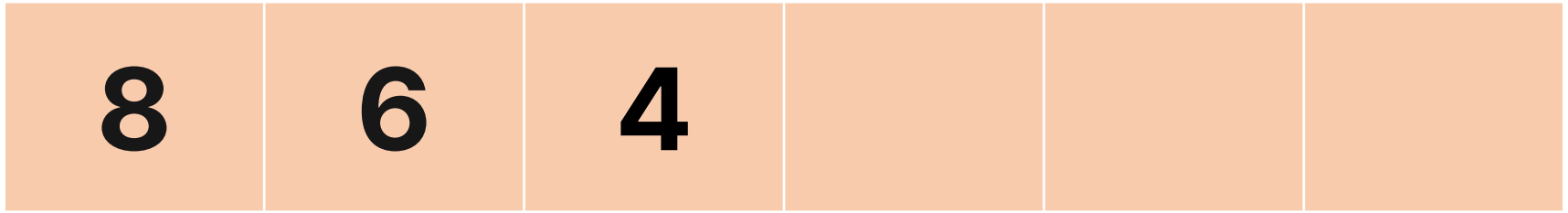
- 8 6 4 9 7 2 3 2 4 1 8
- 첫번째 수는 오큰수로 지정해줄 원소가 없다. 바로 스택에 Push

오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 두번째 수를 넣기 전에 스택을 먼저 살펴본다.
- 스택에 제일 작은 수가 8이므로, 6은 오큰수로 지정될 수 없다. Push

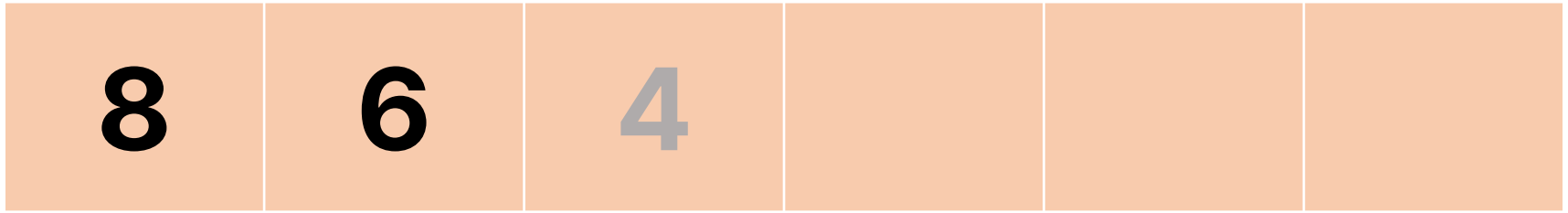
오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 스택에 제일 작은 수가 6이므로, 4은 오큰수로 지정될 수 없다. Push



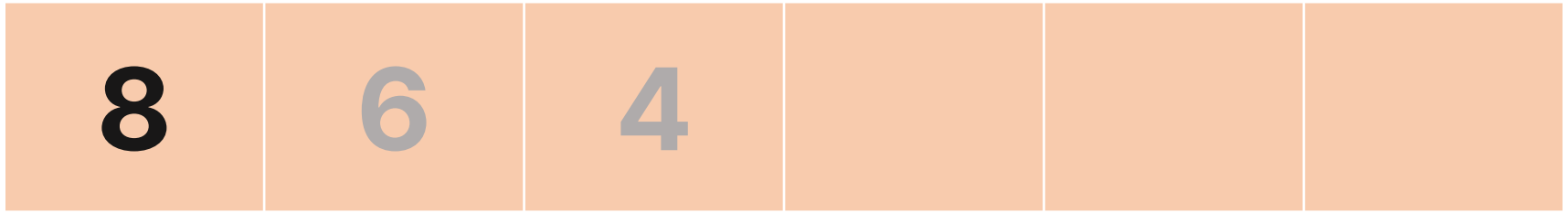
오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 스택에 제일 작은 수가 4이므로, 9는 4의 오큰수다
- 4는 오큰수가 정해졌으므로, 스택에서 빼낸다



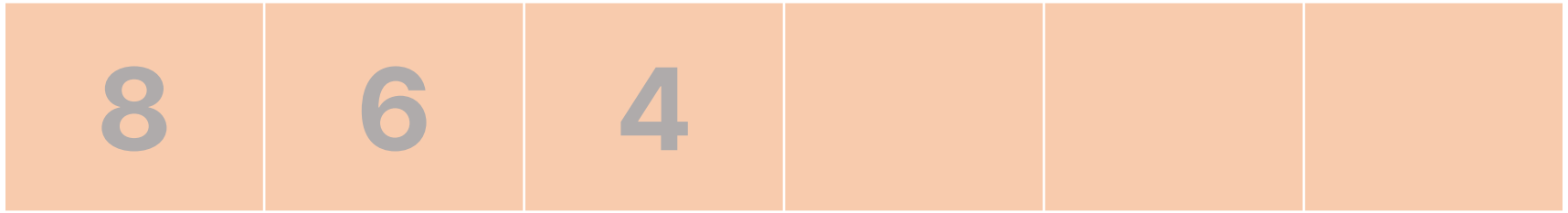
오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 그 다음 스택에서 작은 수가 6이므로, 6도 오큰수가 정해지게 된다
- 6도 오큰수가 정해졌으므로, 스택에서 빼낸다



오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 스택에 남은 수인 8도 9보다 작으므로 오큰수가 정해지게 된다
- 8도 오큰수가 정해졌으므로, 스택에서 빼낸다



오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 더 이상 스택에 숫자가 남아있지 않으므로 9를 넣는다

오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 스택에 가장 작은 수가 9이므로 7보다 큰 수들만 존재한다. Push

오큰수 BOJ 17298

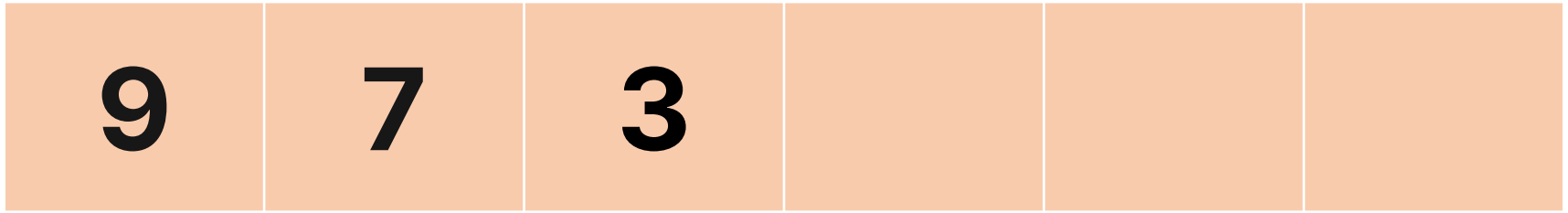
- 8 6 4 9 7 2 3 2 4 1 8
- 스택에 가장 작은 수가 7이므로 2보다 큰 수들만 존재한다. Push

오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 스택에 가장 작은 수가 2이므로 3보다 작다
- 2는 오큰수가 정해지므로 빼낸다

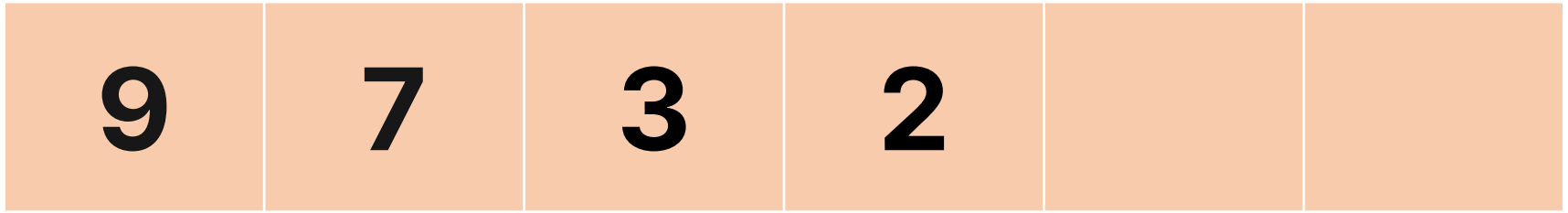
오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 그 다음 작은 수가 7이므로 더 이상 오큰수를 지정할 수 없다. Push



오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 스택에 가장 작은 수가 3이므로 오큰수로 지정할 수가 없다. Push



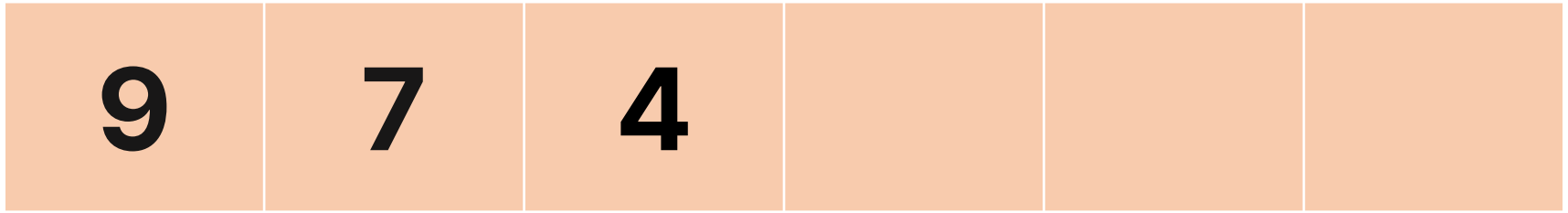
오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 스택에 4보다 같거나 큰 수가 나올 때까지 오큰수로 지정하고, 빼낸다.

9	7	3	2		

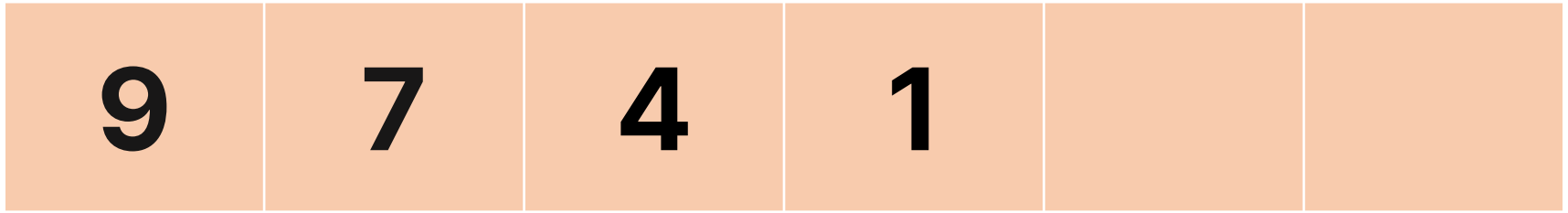
오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 7은 4보다 큰 수이므로 멈추고 4를 넣는다. Push



오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 스택에 가장 작은 수가 4이므로 스택에느 1보다 큰 수들만 존재한다
- 1을 넣는다. Push



오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 스택에서 8보다 같거나 큰 수가 나올 때까지 오큰수를 정하고 빼낸다.

9	7	4	1		

오큰수 BOJ 17298

- 8 6 4 9 7 2 3 2 4 1 8
- 9은 8보다 큰 수이므로 멈추고 8을 넣는다. Push

오큰수 BOJ 17298

- 모든 수를 다 살펴보았다
- 스택에 남아있는 수는 오큰수가 존재하지 않는 수이다
- -1을 출력해준다

9	8				

Bitmasking 문제 풀이

문제

- 사격 내기 BOJ 27960
- 막대기 BOJ 1094
- 데스스타 BOJ 11811
- 외판원 순회 BOJ 2098

사격 내기 BOJ 27960

- 1, 2, 4, 8, 16, 32, 64, 128, 256, 512점 표적이 있으며 A와 B의 점수가 주어진다
- C는 A와 B 중에서 한 명만 맞춘 표적을 맞췄으며, A와 B가 모두 맞추지 못하거나 맞춘 표적은 맞추지 못하였다

사격 내기 BOJ 27960

- 어떤 한 점수를 만드는 조합이 여러가지 있다면 매우 어려운 문제가 되었을 것
- 만약 표적의 점수가 1, 2, 3, 4, 5 점이며, A는 6점, B는 2점이라고 가정하자
- 이런 경우, A는 1, 5점과 2, 4점의 경우가 2가지 존재하며, B의 경우 2점 1가지만 존재한다

사격 내기 BOJ 27960

- A가 1, 5점 표적을 맞춘 경우, A만 맞춘 표적이 1, 5점, B만 맞춘 표적이 2점이므로 C는 $1 + 2 + 5 = 8$점을 얻게 된다
- A가 2, 4점 표적을 맞춘 경우, A만 맞춘 표적이 4점, A와 B가 같이 맞춘 표적이 2점이므로, C는 4 점을 얻게 된다

- 만일, 이러한 경우가 존재하는 경우 문제에 추가 조건이 필요하다

사격 내기 BOJ 27960

- 결론: 이번 문제에서는 어떤 점수를 만들 때, 가능한 경우의 수가 반드시 하나이므로 가능하다

- 이걸 직관적으로 당연히 그렇겠지 하고 문제를 풀 수 있지만, 왜 그런지 알 수 있어야 한다

사격 내기 BOJ 27960

- 가능한 경우를 모두 만들어 보는 것도 한가지 방법이다
- 제일 직관적으로 보이는 내용을 사용해보자

- **표적의 점수는 2의 거듭제곱이다**

사격 내기 BOJ 27960

- 2의 거듭제곱에서 알 수 있는 부분은 이진수로 나타냈을 때 1인 비트가 1개이다는 것을 알 수 있다

- 비트마스킹에서 배운 것처럼 왼쪽으로 shift하는 것은 값을 2배 하는 것이다
- 1은 0000 0001처럼 제일 오른쪽에 비트 1이 하나 있다
- 나머지 점수들은 전부 1을 왼쪽으로 shift해서 만들어진다

사격 내기 BOJ 27960

표적 점수	비트	표적 점수	비트
1	0000 0000 0001	32	0000 0010 0000
2	0000 0000 0010	64	0000 0100 0000
4	0000 0000 0100	128	0000 1000 0000
8	0000 0000 1000	256	0001 0000 0000
16	0000 0001 0000	512	0010 0000 0000

사격 내기 BOJ 27960

- 다음 비트를 보았을 때 비트가 1인 숫자 중에서 자리가 동일한 숫자가 없다

- 어떤 한 자리의 비트를 1로 만들기 위해서는 두 가지가 있다
- 해당 자리가 1인 숫자를 더하는 방법
- 그것보다 낮은 자리에서 올림 수로 올라오는 것

사격 내기 BOJ 27960

- 올림 수로 낮은 자리에서 숫자가 올라오려면 비트가 1이 같은 자리에 있는 두 숫자가 더해져야 한다

- 0001_2과 0010_2 을 더하면 같은 자리에 1이 없기 때문에 올림수가 생기지 않는다
- 올림수가 생기기 위해서는 0010_2과 0011_2처럼 같은 자리에 1이 있어야 둘이 더해지면서 올림수가 생긴다

사격 내기 BOJ 27960

- 해당 표적 점수들은 같은 자리에 1인 비트가 있는 수의 조합이 아무것도 없으므로 올림 수가 생기지 않는 것을 알 수 있다
- 따라서 k번째 비트가 1인 숫자를 만들기 위해서는 반드시 k번째 비트가 1인 숫자를 넣어야 한다

사격 내기 BOJ 27960

- 종합적으로 보면 만들 수 있는 수는 반드시 단 한가지의 경우만 존재한다는 것을 알 수 있다
- 그러면 만들 수 있는 범위를 살펴보자
- 0000 0000 0000부터 0011 1111 1111까지 만들 수 있으므로 0부터 1023까지 만들 수 있음을 알 수 있다
- 이를 통해 입력으로 들어오는 숫자를 모두 만들 수 있으며, 만들 수 있는 경우는 단 한 가지만 존재함을 알 수 있다

사격 내기 BOJ 27960

- 점수를 분석하여 A가 맞춘 표적의 종류, B가 맞춘 표적의 종류를 알아내면 C가 맞춘 표적의 점수를 알아낼 수 있다
- 우리가 아직 사용하지 않은 정보가 있다
- 모든 표적의 점수를 2진수로 나타내면 1인 비트는 단 하나이며 자리가 겹치는 경우가 없다
- 이를 이용해 우리는 점수를 이진수로 나타내면 어떤 표적을 맞췄는지 바로 알 수 있다

사격 내기 BOJ 27960

- 비트가 곧 점수이므로 색상처럼 변환표가 따로 필요없다
- A와 B가 다른 경우 표적을 맞춘 경우이며 둘이 같은 경우 표적을 못 맞춘 경우이다
- 이는 XOR 연산과 동일하다

- 따라서 A와 B의 점수를 XOR 하면 C의 점수가 나온다

사격 내기 BOJ 27960

```cpp
int main() {
    int a, b;
    cin >> a >> b;
    cout << (a ^ b);

    return 0;
}
```

막대기 BOJ 1094

- 64cm 막대기에서 시작한다
- Xcm보다 내가 가지고 있는 막대기가 길면 가지고 있는 막대기 중 가장 작은 것을 절반으로 자른다
- 절반을 버렸을 때(만약 없다고 생각했을 때), 가지고 있는 길이가 X보다 길면 그 절반을 버린다
- Xcm를 만들 때까지 반복한다

막대기 BOJ 1094

- 간단하게 만들어보자
- 64를 32 2개로 자르고 만일 X가 32보다 작다면 하나를 버린다
- 이런 식으로 반복하다 보면 64cm 하나 이거나 32, 16, 8, 4, 2, 1 길이의 막대기로 조합한 결과를 만들 수 있음
- 2의 제곱 형태 -> 비트로 사용할 수 있다

막대기 BOJ 1094

- 길이의 종류를 표적 점수라고 생각한다면 앞선 문제와 동일하다
- 따라서 Xcm가 주어진다면 길이의 조합을 생각하면 된다
- 하지만 몇 개인지를 나타내면 되므로, 비트가 1인 자리의 개수를 출력하면 된다

- 비트가 1인 것의 개수를 세보자

막대기 BOJ 1094

- 길이의 종류를 표적 점수라고 생각한다면 앞선 문제와 동일하다
- 따라서 Xcm가 주어진다면 길이의 조합을 생각하면 된다
- 하지만 몇 개인지를 나타내면 되므로, 비트가 1인 자리의 개수를 출력하면 된다

- 비트가 1인 것의 개수를 세보자

막대기 BOJ 1094

- 비트가 1인 것을 세는 방법
- 값이 0 이면 더 이상 비트가 1인 자리가 존재하지 않는다
- 반대로 값이 0이 아니라면 비트가 1인 자리가 하나 이상 존재하는 것

막대기 BOJ 1094

- 1을 AND 연산 하면 홀수 짝수 판별과 같이 제일 오른쪽 자리에 1이 존재하는지 아닌지 알 수 있다
- 오른쪽 shift를 하면 제일 오른쪽 비트를 지워가면서 자리를 밀어낼 수 있다

- 이 둘을 조합하자

막대기 BOJ 1094

```
int count_bit(int x) {
    int cnt = 0;
    while (x) {
    if (x & 1)
        cnt++;
        x >>= 1;
    }

        return cnt;
}
```

막대기 BOJ 1094

- 또는 만들어진 함수를 사용하자

```
// gcc 내장 함수
__builtin_popcount(x);
__builtin_popcountll(x);

// C++20부터
std::popcount(x);
```

데스스타 BOJ 11811

- 원래의 수열을 구하자
- 수열의 각 원소를 $a_1, a_2, \ldots, a_N$이라고 하자
- 행렬이 주어지며 $i \neq j, 1 \leq i, j \leq N$인 i, j에 대하여 a_i와 a_j를 AND 비트연산 한 결과가 주어진다
- $1 \leq N \leq 1\,000$

데스스타 BOJ 11811

- 가능한 경우만 입력으로 주어진다
- 따라서 입력을 그대로 사용하면 된다

- 입력에 대한 답이 존재한지 또는 불가능한지 확인하려면 2-SAT 문제로 변질된다

데스스타 BOJ 11811

- 만약 a_i와 a_i를 AND 한 결과가 주어지는 경우 a_i를 바로 알 수 있다

- 같은 수를 AND하므로 0과 0을 AND하거나 1과 1을 AND하게 되는데, 0과 0을 AND 하는 경우 0, 1과 1을 AND 하는 경우 1이 나온다.
- A & A = A이므로 바로 값이 나오게 된다

데스스타 BOJ 11811

- AND의 연산 결과는 두 숫자 모두 해당 자리에 1을 가지고 있어야 1이 나오게 된다
- a_i와 a_j를 AND 비트연산한 결과에서 어떠한 자리에 1이 있다면 a_i와 a_j는 반드시 그 자리에 1을 둘 다 갖고 있다

데스스타 BOJ 11811

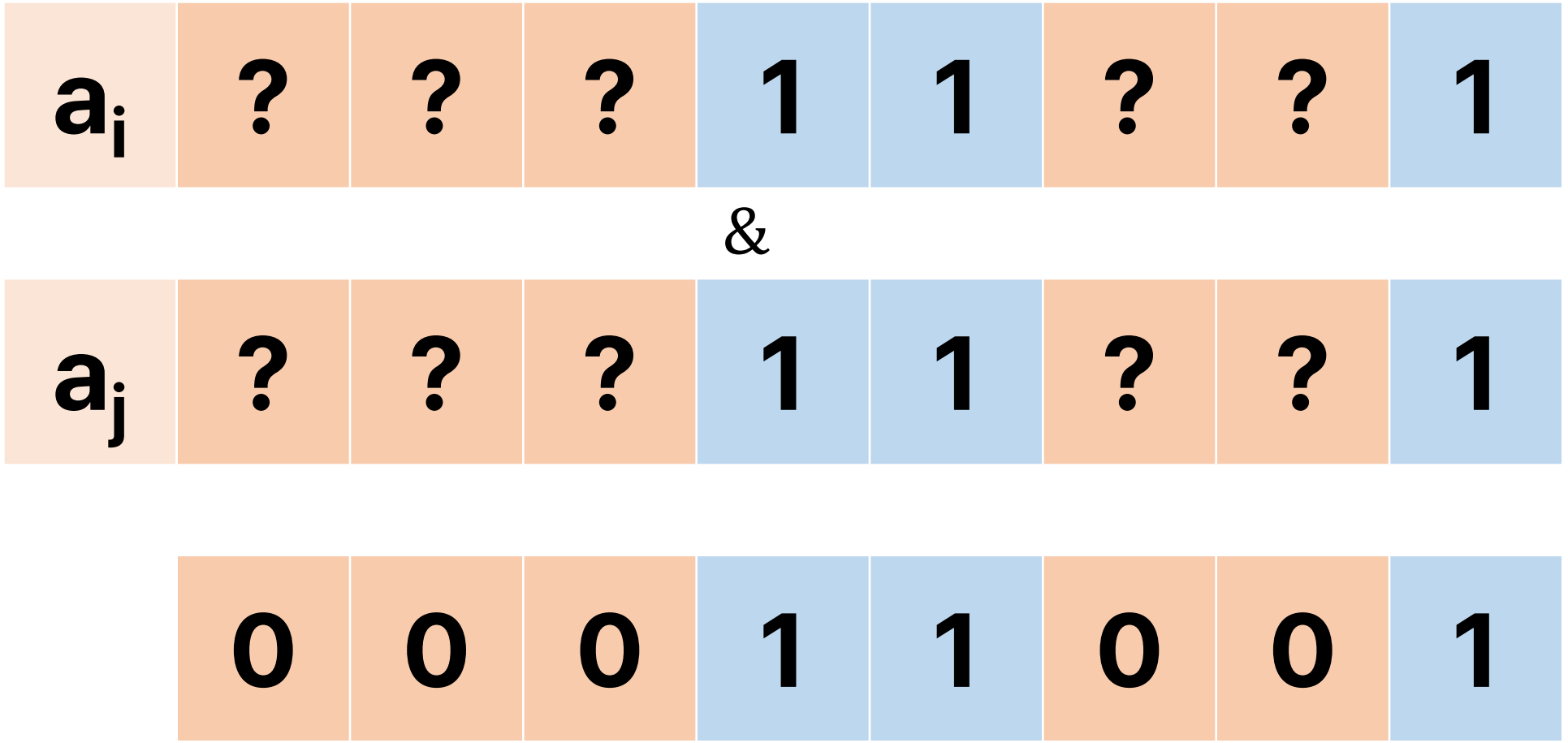
a_i	?	?	?	?	?	?	?	?

&

a_j	?	?	?	?	?	?	?	?

	0	0	0	1	1	0	0	1

데스스타 BOJ 11811



데스스타 BOJ 11811

- 모든 결과를 이용해 모든 수열에 1이 필수로 필요한 자리를 채우자
- 1로 채우려면 어떻게 해야 할까?

데스스타 BOJ 11811

- a_i에 결과값을 OR 연산하면 된다

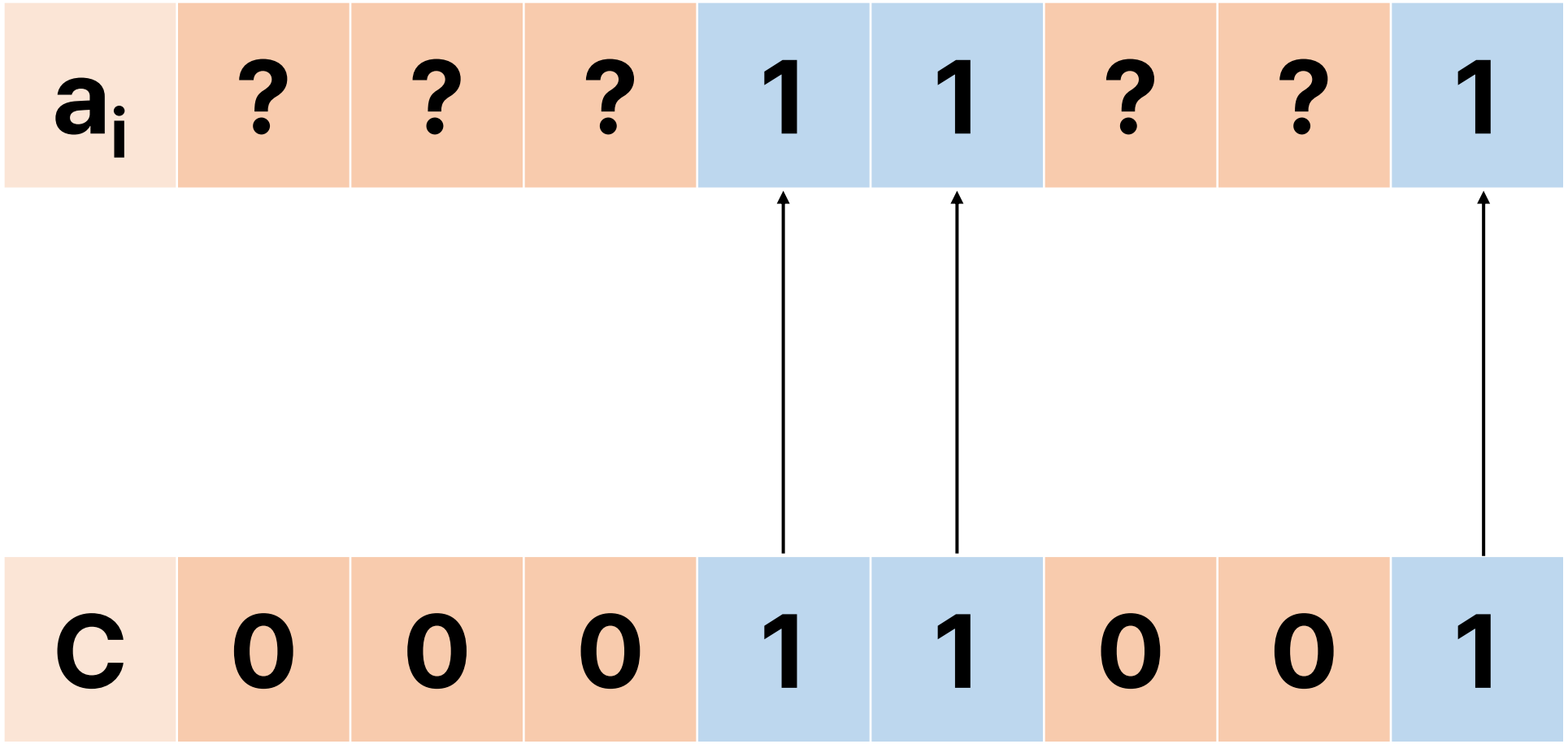
- a_i에 결과값을 OR 연산하면 결과값 중 1인 부분은 1로 채워진다
- 결과값 중 0인 부분은 a_i의 값으로 남아있는다

데스스타 BOJ 11811

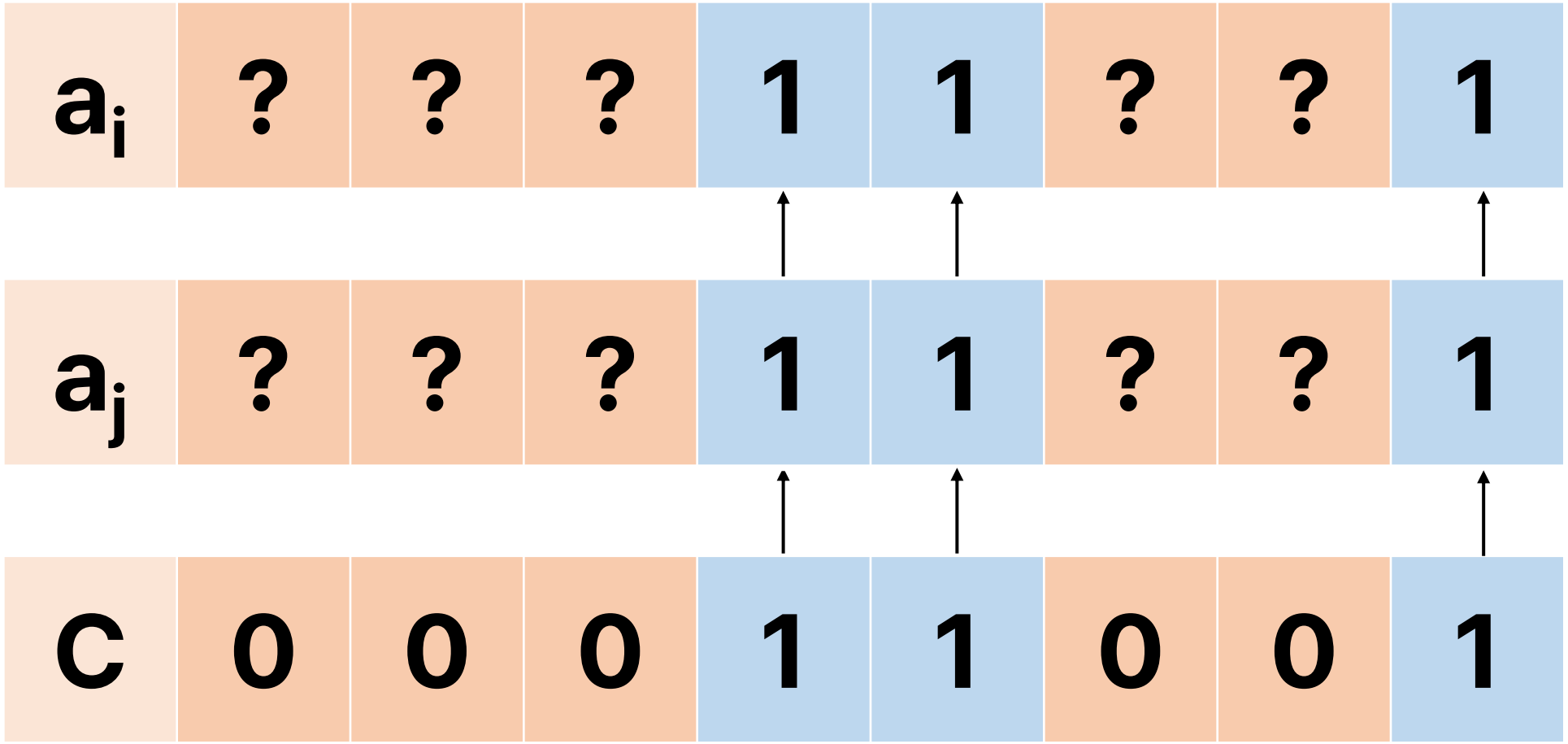
a_i	?	?	?	?	?	?	?	?

C	0	0	0	1	1	0	0	1

데스스타 BOJ 11811



데스스타 BOJ 11811



데스스타 BOJ 11811

- 모든 결과를 이용해 모든 수열에 1이 필수로 필요한 자리를 채우자
- ?에는 0과 1중 어떤 것을 채워야 할까?

a_i	?	?	?	1	1	?	?	1

데스스타 BOJ 11811

- ?는 결과가 0이 나온 자리이다

- a_i와 a_j 둘 다 해당 자리에 0을 가지고 있거나 둘 중 하나만 1을 가지고 있는다
- 항상 가능한 경우만 주어지므로 둘 다 1을 갖게 하는 입력은 들어오지 않는다

데스스타 BOJ 11811

- ex) a_i가 다른 수와 AND 연산을 한 결과가 해당 자리에 1이 포함
- a_j도 다른 수와 AND 연산을 한 결과가 해당 자리에 1이 포함
- 이 경우 a_i와 a_j 둘 다 1을 가져가 하므로 모순이 생긴다(답이 존재할 수 없음)

데스스타 BOJ 11811

- 이러한 방식으로 해답의 존재유무를 파악하고 가능한 결과가 존재하는 지 확인하는 알고리즘이 존재
- K-SAT
- 그래프 기반 2-SAT 알고리즘이 존재, 매우 어려움

데스스타 BOJ 11811

- 1을 넣으려고 한다면 다른 수들을 전부 살펴보아 해당 자리에 1이 있는 수가 존재하는지 확인해야한다
- 존재하는 경우, 연산 결과가 달라지므로 1을 넣을 수 없다

- 모든 경우의 수를 찾아야 한다면 1을 고려해야 하지만 가능한 경우 중 한가지만 얻어내면 되므로 ?자리에는 0을 넣어주도록 하자

데스스타 BOJ 11811

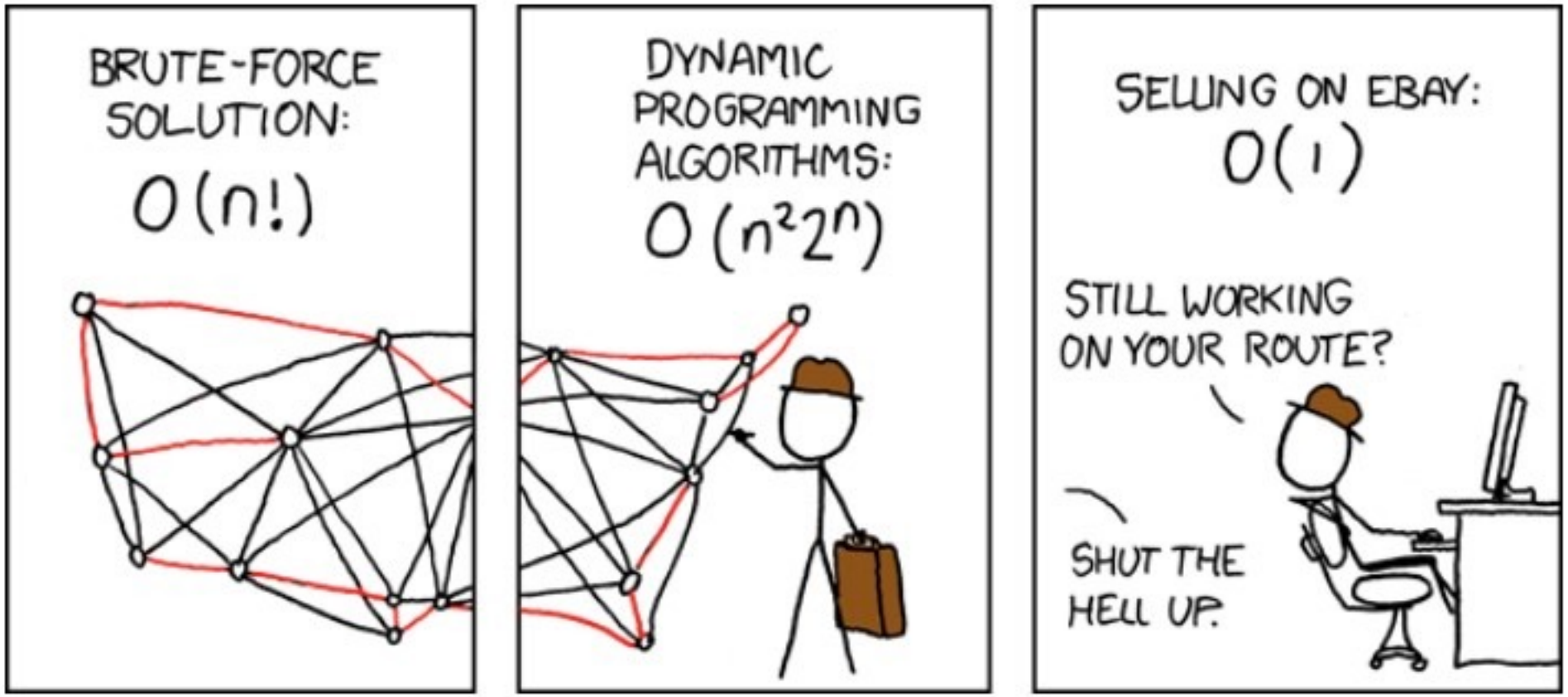
- 따라서 종합해보면 0으로 전부 초기화 해두고, 결과값을 OR 연산해 필요한 자리에 1을 넣으면 된다

- 그렇게 나오는 결과는 가능한 경우 중 한 가지이다
- 다른 해답 또한 존재

외판원 순회 BOJ 11811

- 모든 도시를 순회하고 돌아오는데 최소 비용이 걸리도록 하는 문제
- a도시에서 b도시로 가는 비용과 b도시에서 a도시로 가는 비용은 동일하지 않을 수 있다
- Traveling Salesman Problem(TSP)라고 불리며 CS에서 많이 접하게 되는 문제이다

외판원 순회 BOJ 11811



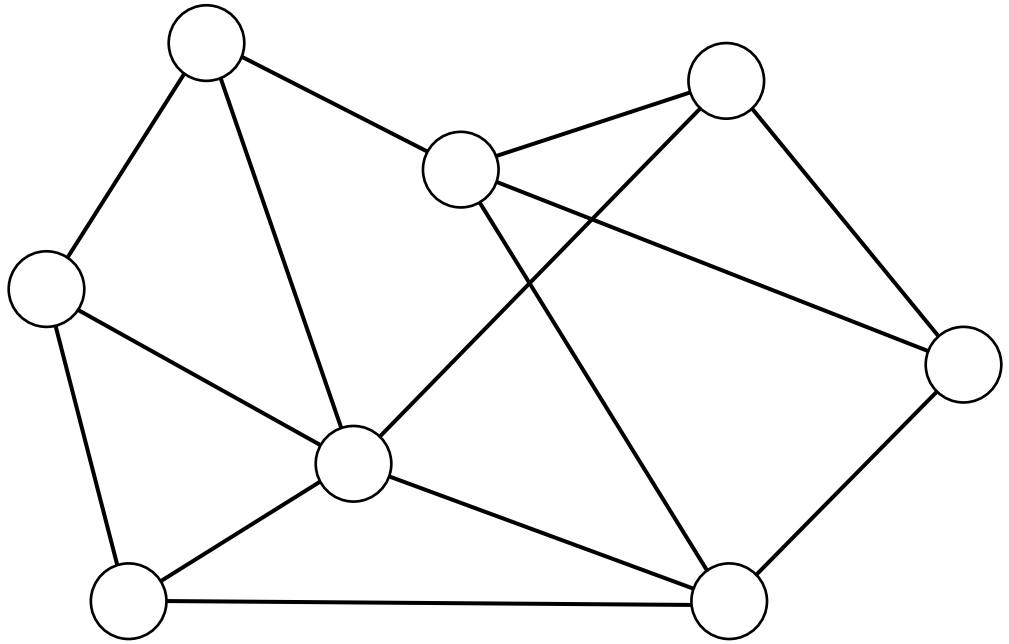
외판원 순회 BOJ 11811

- 가장 간단한 풀이
- 모든 경우의 수를 다 해본다
- N개의 도시를 방문할 경우를 세야하므로 N!의 경우의 수가 존재한다

- 더 줄일 수 있는 방법은 없을까?

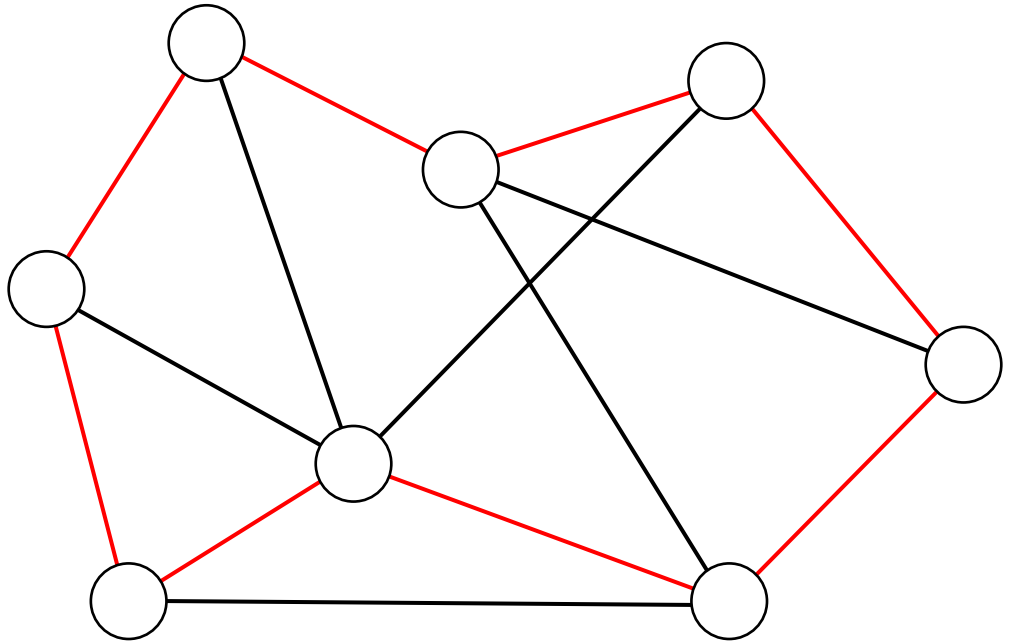
외판원 순회 BOJ 11811

- 우선 어느 도시에서 출발할 지 정해보자



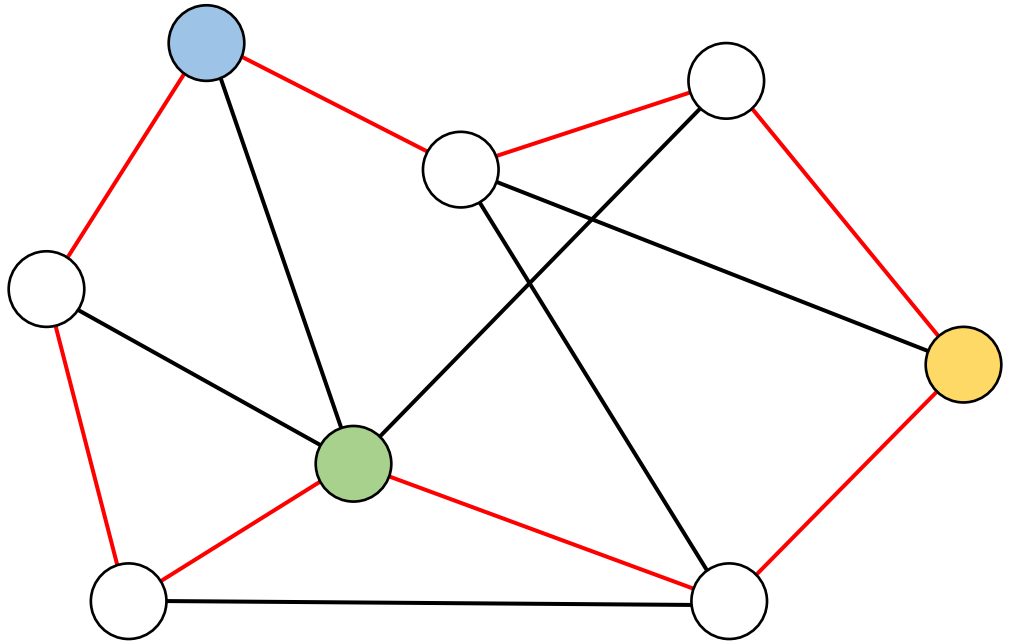
외판원 순회 BOJ 11811

- 빨간색 최소 비용이 드는 정답이라고 가정하자



외판원 순회 BOJ 11811

- 어떤 도시에서 시작하여도 정답은 동일하다, 즉 시작도시는 아무 곳이나 해도 된다



외판원 순회 BOJ 11811

- 도시가 5개가 있다고 생각해보자
- 이때 가능한 경우를 살펴보자

  1 → 2 → 3 → 4 → 5 → 1

  4 → 1 → 2 → 3 → 5 → 1

- 1 → 2 → 3으로 가는 경로는 겹치는 것을 알 수 있다

외판원 순회 BOJ 11811

- 모든 경우의 수를 다 살펴보아야 답을 구할 수 있다
- 1 → 2 → 3 → 4 → 5 → 1으로 갈 때 1 → 2 → 3 을 기억하고 있다면

  4 → 1 → 2 → 3 → 5 → 1을 살펴볼 때 4 → 1 + 기억한 값 +3 → 5 → 1로 찾는 다면 계산 횟수를 줄일 수 있다

- 우리는 방문했던 경로들을 **기억**하고 있어야한다

외판원 순회 BOJ 11811

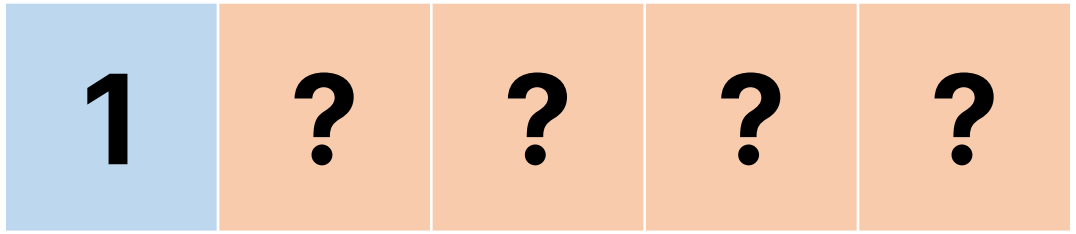
- 무슨 값들을 저장하고 있어야 할까?
- 어떤 도시들을 방문했는지 기억하고 있어야 한다
- 어떤 도시를 방문했는지 어떻게 표시하는가?

- 여러 개의 변수를 사용할 수 있지만 비트마스킹을 통해 하면 간편하다

외판원 순회 BOJ 11811

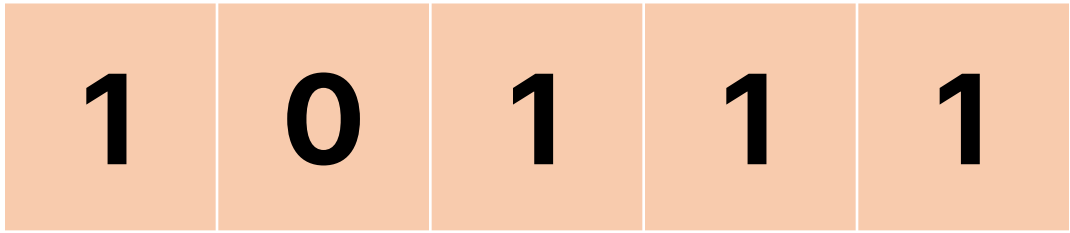
- 5개의 도시라면 5개의 비트가 필요하다
- 1번째 비트가 1이면 1번째 도시를 방문했다는 뜻이다

  1번째 비트가 0이면 1번째 도시를 방문하지 않았다는 뜻이다



외판원 순회 BOJ 11811

- 비트 10111은 다음과 같이 해석할 수 있다
- 첫번째 도시 방문
- 두번째 도시 미방문
- 세번째 도시 방문
- 네번째 도시 방문
- 다섯번째 도시 방문



외판원 순회 BOJ 11811

- 비트를 사용하면 좋은 점 배열로써 다룰 수 있다
- 해당 비트를 배열의 인덱스로써 사용하면 쉽게 값을 저장, 접근할 수 있다

1	0	1	1	1

외판원 순회 BOJ 11811

- 외판원 순회는 다이나믹 프로그래밍을 기반으로 사용한다
- 우리가 저장할 배열: dp[방문한 도시들(비트마스킹)][현재 있는 도시]
- 배열에 저장하는 값: 나머지 도시들을 방문하고 출발 도시로 갈 때의 최소 비용

- 해당 비용들을 저장해두고 이후에 재사용될 때 저장해둔 값을 이용해 연산을 줄인다
- 저장된 값이 없는 경우, 구해야한다

외판원 순회 BOJ 11811

```
int tsp(int visited, int cur) {
    if (__builtin_popcount(visited) == n) {
        if (!arr[cur][0])
            return INF;
        return arr[cur][0];
    }
    if (dp[visited][cur])
        return dp[visited][cur];
    dp[visited][cur] = INF;
    for (int i = 1; i < n; i++) {
        if (visited & (1 << i))
            continue;
        if (!arr[cur][i])
            continue;
        dp[visited][cur] = min(dp[visited][cur], tsp(visited | (1 << i), i) + arr[cur][i]);
    }
    return dp[visited][cur];
}
```