2025 겨울방학 알고리즘 스터디

비트마스킹

컴퓨터 공학과 20230546 서보경

목차

1. 비트 연산 다들 하실 줄 아시죠?
2. 비트마스킹 활용 – 집합 편
3. 비트마스킹 활용 – 부분 집합 탐색 편
4. Nim 게임과 스프라그-그런디 정리

기본적인 비트 연산

AND 연산

두 비트(이진 비트)를 비교하여 둘 다 1인 경우 결과가 1, 나머지는 0이 되는 연산. 기호로는 &
ex) 5 (0101) & 3 (0011) = 1 (0001)

OR 연산

두 비트(이진 비트)를 비교하여 하나라도 1인 경우 결과가 1, 나머지는 0이 되는 연산. 기호로는 |
ex) 5 (0101) | 3 (0011) = 7 (0111)

NOT 연산

1은 0으로, 0은 1로 비트를 반전시키는 연산. 기호로는 ~
ex) ~5 (0101) = 10 (1010)

XOR 연산

두 비트를 비교하여 다르면 1, 같으면 0이 되는 연산. 기호로는 ^
ex) 5 (0101) ^ 3 (0011) = 6 (0110)

비트마스킹에서 각 연산의 사용 방법

AND 연산 – 특정 비트를 확인 할 때 / 필요한 비트만 남겨야 할 때

OR 연산 – 특정 비트를 켤 때 (0 -> 1)

NOT 연산 – 비트의 반전이 필요할 때 / 보수 연산을 해야 할 때 (부호 비트도 반전) <- 잘 안씀

XOR 연산 – 두 값을 스왑할 때 / 중복된 값을 제거할 때

+) 시프트 연산 : 어떤수에 2의 거듭제곱을 곱하거나 나눌 때 씀. 기호로는 << / >>

ex) 1 (00000001) << 7 = 2^7 (10000000)

2^7 (10000000) >> 7 = 1 (00000001)

비트마스킹의 특징

시간과 공간적으로 효율적이다.

- 비트연산은 CPU가 하드웨어 수준에서 직접 처리하는 가장 빠른 연산 중 하나이다. (O(1) 보장)
- 비트마스크는 n개의 원소를 단일 변수형 변수로 저장할 수 있다.

ex) n이 32일 때, 32개의 원소 상태를 int 변수 단 하나로 표현 가능.

직관적이고 구현이 간단하다.

- 비트 연산은 상태 표현과 조작을 논리적으로 간단하게 표현할 수 있다.
  - 특정 상태 추가 : mask |= (1 << k)
  - 특정 상태 제거 : mask &= ~(1 << k)
  - 특정 상태 확인 : mask & (1 << k)
  - 상태 반전 : mask ^ (1 << k)

대규모 탐색 문제에 유리하다.

- 특히 비트마스킹은 2^n 크기의 부분 집합을 탐색하는데 최적화 되어있다.
  - n개의 원소로 이루어진 집합에서 2^n 크기의 부분 집합을 표현 가능하다.
  - 그래프 탐색, 조합 탐색, 최적화 문제에서 엄청난 무기이다.

원소 개수가 64개 (long long) 범위를 넘어가는 순간 사용이 불가능하거나 효율이 매우 떨어진다.
➔ 파이썬의 큰 수 연산은 리스트 기반이여서 비트마스킹 불가, JAVA BigInteger은 스트링 기반이라 불가

또한 연산자 우선순위에 상당히 주의해야한다!! (괄호로 감싸고 비트마스킹 사용하는것을 추천)

비트마스킹 활용 – 집합 편

시간 제한	메모리 제한	제출	정답
1.5 초	4 MB (하단 참고)	129375	39644

문제

비어있는 공집합 S가 주어졌을 때, 아래 연산을 수행하는 프로그램을 작성하시오.

- `add x` : S에 x를 추가한다. (1 ≤ x ≤ 20) S에 x가 이미 있는 경우에는 연산을 무시한다.
- `remove x` : S에서 x를 제거한다. (1 ≤ x ≤ 20) S에 x가 없는 경우에는 연산을 무시한다.
- `check x` : S에 x가 있으면 1을, 없으면 0을 출력한다. (1 ≤ x ≤ 20)
- `toggle x` : S에 x가 있으면 x를 제거하고, 없으면 x를 추가한다. (1 ≤ x ≤ 20)
- `all` : S를 {1, 2, ..., 20} 으로 바꾼다.
- `empty` : S를 공집합으로 바꾼다.

입력

첫째 줄에 수행해야 하는 연산의 수 M (1 ≤ M ≤ 3,000,000)이 주어진다.

둘째 줄부터 M개의 줄에 수행해야 하는 연산이 한 줄에 하나씩 주어진다.

딱 보면 배열로 구현해도 시간 제한이 널널해 보인다.
(연산 300만 * 원소 개수 20개) - > 6000만 (0.6초)

하지만 배열로 구현하면 시간 초과가 난다. 왜?

-> 실제로 한번 연산당 최대 20의 연산량이 소모되는건 맞다.
그러나 메모리 접근 비용이 존재하며, CPU 캐시와 메모리
사이 병목 현상으로 인해 연산량이 생각보다 더 많아짐!
또한, 입출력도 해야 하므로 여러가지 변수를 따지면 배열은
이 문제를 해결하기엔 부족하다!!

입출력 연산을 빠르게 하는데는 상한선이 있으니까, 각 연산을
엄청 빠르게 해결하기 위해 무조건 O(1)이 보장되는
비트마스킹을 쓰면 되겠네!! 원소도 20개고!

비트마스킹 활용 – 집합 편

int S

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(int S = 0)

비트마스킹 활용 – 집합 편

int S

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

query : add 7

집합에 7을 추가하는 연산이다.

특정 비트를 켜기 위해서는 OR연산이 적합하다.

고로 S = S | (1 << 7)을 하여 비트를 켜주자! (시프트 연산으로 켤 비트를 지정하는 방식)

비트마스킹 활용 – 집합 편

int S

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0

query : add 10

집합에 10을 추가하는 연산이다.

특정 비트를 켜기 위해서는 OR연산이 적합하다.

고로 S = S | (1 << 10)을 하여 비트를 켜주자!

비트마스킹 활용 – 집합 편

int S

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

query : remove 7

집합에서 7을 삭제하는 연산이다.

특정 비트를 없애기 위해서는 NOT 연산과 AND 연산을 조합해주면 된다.

먼저, ~(1<< 7)을 하여 비트에서 7만 꺼준 후, S = S & ~(1<<7) 연산으로 특정비트를 끌 수 있다.
나머지 비트는 전부 1이여서 영향 X

비트마스킹 활용 – 집합 편

int S

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

query : toggle 18

집합에서 18이 있으면 제거하고, 없으면 삽입하는 연산이다.

즉, 특정 비트를 반전 시키는 XOR 연산이 적합하다.

고로, S = S ^ (1<< 18)을 하여 특정 비트를 반전시켜 주자!
(다른 비트들은 [1 ^ 0 = 1] or [0 ^ 0 = 0] 으로 원상 유지되기 때문에 신경써줄 필요 없다.

비트마스킹 활용 – 집합 편

int S

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

query : empty

집합을 전부 비우는 연산이다.

비트를 일일히 확인하면서 AND 연산으로 비워줘야 할까?

집합이 비었다는건, 모든 비트가 0인 상태이다. 고로 S = 0; 연산으로 초기화 시켜주면 된다.

비트마스킹 활용 – 집합 편

int S

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

query : all

집합을 {1,2…..,20}으로 전부 채우는 연산이다.

비트를 일일히 확인하면서 OR 연산으로 채워줘야 할까?

먼저, (1 << 21)을 생각하자. 여기서 1을 빼준다면 결론적으로 1부터 20까지 모든 비트가 켜진 상태가 된다! 고로 S = ((1 << 21) - 1) | S를 하여 모든 원소를 O(1)에 추가할 수 있다.

비트마스킹 활용 – 집합 편

int S

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

query : check 10

집합에서 10이 있으면 1을, 없으면 0을 출력하는 연산이다.

집합에 원소를 넣는 것과 같이, 특정 원소를 확인하기 위해서는 시프트 연산 (1 << 10)을 활용한다.

고로, 1과 AND 연산을 이용하면 현재 비트가 켜져 있는지 꺼져 있는지 쉽게 확인이 가능하다.

S & (1<< 10)

비트마스킹 활용 – 집합 편

int S

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

이처럼 비트마스킹을 활용하면 방문 처리뿐만 아니라 상태 관리도 효율적으로 할 수 있다.
(특히, 백트래킹이라던가 그래프 탐색에서!)

하지만 원소가 64개를 초과하는 경우에는 일반 배열이나 다른 자료구조를 사용하는 것이 더 적합하다.

비트마스킹 활용 – 부분 집합 탐색편

도영이는 짜파구리 요리사로 명성을 날렸었다. 이번에는 이전에 없었던 새로운 요리에 도전을 해보려고 한다.

지금 도영이의 앞에는 재료가 N개 있다. 도영이는 각 재료의 신맛 S와 쓴맛 B를 알고 있다. 여러 재료를 이용해서 요리할 때, 그 음식의 신맛은 사용한 재료의 신맛의 곱이고, 쓴맛은 합이다.

시거나 쓴 음식을 좋아하는 사람은 많지 않다. 도영이는 재료를 적절히 섞어서 요리의 신맛과 쓴맛의 차이를 작게 만들려고 한다. 또, 물을 요리라고 할 수는 없기 때문에, 재료는 적어도 하나 사용해야 한다.

재료의 신맛과 쓴맛이 주어졌을 때, 신맛과 쓴맛의 차이가 가장 작은 요리를 만드는 프로그램을 작성하시오.

입력

첫째 줄에 재료의 개수 N(1 ≤ N ≤ 10)이 주어진다. 다음 N개 줄에는 그 재료의 신맛과 쓴맛이 공백으로 구분되어 주어진다. 모든 재료를 사용해서 요리를 만들었을 때, 그 요리의 신맛과 쓴맛은 모두 1,000,000,000보다 작은 양의 정수이다.

출력

첫째 줄에 신맛과 쓴맛의 차이가 가장 작은 요리의 차이를 출력한다.

원소가 10개 밖에 안되니까 아까 배운 비트마스킹 연산으로 날먹 할 수 있겠네.

어 근데, 모든 부분 집합을 생성하는게 까다로운데, 더 간단하게 부분집합을 판단하는 방법이 없을까?

-> 비트마스킹과 시프트 연산을 이용해서 모든 부분집합 탐색하기!

비트마스킹 활용 – 부분 집합 탐색편

```
2
3 8
5 8
```

int tasty

2 (5,8)	1(3,8)
0	1

최소 차이 : INF

먼저, 순회 하기위해 변수 하나를 준비한다.

요리를 만들긴 해야 하기 때문에 1부터 시작한다. (안 만들어도 되면 0부터 가능)

신맛과 쓴맛의 차이가 가장 작은 요리의 차이를 구해야 하므로 처음 차이값은 크게 초기화 시킨다.

비트마스킹 활용 – 부분 집합 탐색편

```
2
3 8
5 8
```

int tasty

2 (5,8)	1(3,8)
0	1

최소 차이 : 5

1은 이진수로 “01”이다, 즉 1번째 비트만 켜져 있다.

고로 신맛의 곱은 3 / 쓴맛의 합은 8이다.

차이가 5이므로 INF보다 작다. 고로 갱신해준다. 그 후, tasty값을 1 증가시켜준다.

비트마스킹 활용 – 부분 집합 탐색편

```
2
3 8
5 8
```

int tasty

2 (5,8)	1(3,8)
1	0

최소 차이 : 3

2은 이진수로 “10”이다, 즉 2번째 비트만 켜져 있다.

고로 신맛의 곱은 5 / 쓴맛의 합은 8이다.

차이가 3이므로 5보다 작다. 고로 갱신해준다. 그 후, tasty값을 1 증가시켜준다.

비트마스킹 활용 – 부분 집합 탐색편

```
2
3 8
5 8
```

int tasty

2 (5,8)	1(3,8)
1	1

최소 차이 : 3

3은 이진수로 “11”이다, 모든 비트가 전부 켜져 있다.

신맛의 곱은 5 * 3 = 15 / 쓴맛의 합은 8 + 8 = 16이다.

차이가 1이므로 3보다 작다. 고로 갱신해준다.
2개의 부분 집합의 개수인 3개에 도달 했으므로 부분 집합 탐색을 멈춘다.

비트마스킹 활용 – 부분 집합 탐색편

```
2
3 8
5 8
```

int tasty

2 (5,8)	1(3,8)
1	1

최소 차이 : 3

물론 다른 방식으로 구현하는 경우가 충분히 있을 것이다.

그러나, 비트 마스킹을 이용하면 반복문을 통해 매우 쉽게 구현할 수 있다!

이 방법도 n이 작을 때(약 n <= 20) 가능하다. (NP-Hard의 본질상, 입력 크기가 커질수록 탐색 공간이 지수적으로 증가하기 때문에 현실적으로 해결하기 어렵다.)

Nim 게임 이란

무모순 게임

두 플레이어에게 동일한 이동 / 시행 기회가 주어지는 게임. 즉, 어떤 상태에서 두 플레이어가 선택할 수 있는 선택의 집합이 동일하다. 아래의 있는 Nim 게임이 무모순 게임의 일종이다.

Nim 게임

Nim 게임은 조합 게임 이론에서 가장 기본적이고 중요한 게임 중 하나이다.
이 게임은 스프라그-그런디 정리의 기본 예시로 자주 사용된다.

Nim 게임의 규칙

1. 더미
   - 게임에는 여러개의 돌 더미가 있다. 각 더미에는 돌이 $a_1, a_2, a_3 \ldots, a_n$개 존재한다.

2. 플레이어 턴
   - 두 플레이어가 번갈아 가며 돌을 제거 한다.
   - 각 턴에서 한 플레이어는 한 더미에서 원하는 만큼의 돌을 제거할 수 있다.
   - 단, 돌은 반드시 1개 이상 제거해야 하며, 마지막 돌을 제거하는 플레이어가 승리한다.

Nim 게임의 예시

Nim 합

각 더미에 남은 돌의 개수를 XOR 연산으로 계산한 값. ex) $a_1 \oplus a_2 \oplus \cdots \oplus a_n$ = Nim 합

승리와 패배 조건

Nim 합이 0 일 때 : 현재 상태는 패배 상태이며, 상대가 최적의 플레이를 하면 반드시 패배한다.

Nim 합이 0이 아닐 때 : 현재 상태는 승리 상태이며, 본인이 최적의 플레이를 하면 승리할 수 있다.

ex 1) 두 개의 더미가 존재할 때

돌의 개수 : {3, 4} / Nim 합 계산 : $3 \oplus 4 = 7$
Nim 합이 0이 아니므로 선공이 이긴다.

ex 2) 세 개의 더미가 존재할 때

돌의 개수 : {1, 4,5} / Nim 합 계산 : $1 \oplus 4 \oplus 5 = 0$
Nim 합이 0이므로 선공이 패배한다.

이게 어떻게 가능한거지?

스프라그-그런디 정리

스프라그-그런디 정리

모든 2인 조합 게임을 Nim 게임으로 환원 할 수 있다는 이론.
각 게임 상태에 대해 Grundy 수(그런디 수)를 정의하며, Grundy 수는 해당 상태가 승리 상태인지 패배 상태인지 판별하는데 사용된다.
Grundy 수는 MEX(Minimum EXcludant)를 통해 계산된다.

Grundy 수

Grundy 수 G(S)는 현재 상태 S에서 도달 가능한 모든 다음 상태의 Grundy수의 MEX(최소 누락값)이다.

$$G(S) = MEX(\{G(S_1), G(S_2), \dots, G(S_k)\})$$

여기서 $S_1, S_2, \dots, S_k$는 상태 S에서 가능한 다음 상태들이다.

MEX 정의

MEX(Minimum EXcludant)란 주어진 숫자 집합에서 포함되지 않은 가장 작은 비음수 정수이다.

ex) 집합 {0,1,3}의 MEX = 2 / 집합 {1,2,3}의 MEX = 0

스프라그-그런디 정리

게임의 합

여러 개의 독립적 게임을 동시에 진행하는 것으로, 플레이어는 자신의 턴에 단 하나의 구성 게임에서만 이동을 수행할 수 있다.

Nim 더미와 동등(equivalence)

게임 상태 S가 크기 m인 Nim 더미와 동등하다고 할 때, 둘을 결합(Disjunctive Sum / xor과 동일 개념)해도, 항상 첫 번째 플레이어의 승리(혹은 패배) 여부가 동일하다는 뜻.

단일 게임 상태와 Nim 더미의 동등성 증명

주장 : 모든 무모순 게임 상태 S에 대해서, S는 크기 G(S)인 Nim 더미와 게임적으로 동등하다.

종료 상태 T가 있다고 가정하자.

- 종료 상태에서는 더 이상 이동이 없으므로 $\{S_1, S_2, \dots, S_n\} = \emptyset$
- 따라서 G(T) = MEX(∅) = 0
- Nim 게임에서 돌 더미의 크기 0은 더 이상 움직일 수 없는 상태이며, 이는 종료 상태와 동일하다.

고로 T는 크기 0인 Nim 더미와 동등하다.

단일 게임 상태와 Nim 더미의 동등성 증명

귀납 가정 : 임의의 상태 S'에 대해, S'가 크기 G(S')인 Nim 더미와 동등하자고 하자. 여기서 S'는 S보다 작은 상태, 즉 S'로 갈 수 있는 상태이다.

임의의 상태 S를 고려하자. S에서 가능한 다음 상태들을 $\{S_1, S_2, \dots, S_k\}$라고 하자.

1. 하위 상태에 대한 동등성

- 귀납 가정에 의해 각 S_i 는 크기 G(S_i)인 Nim 더미와 동등하다.

2. 상태 S의 이동과 Nim 게임의 관계

- S에서 한 수를 두어 S_i로 이동하는 것은, 크기 G(S_i)인 Nim 더미로의 이동과 동일하다.
- Nim 게임에서는 더미의 크기를 줄이는 모든 가능한 이동들이 정의되어 있으며, 이를 인해 얻을 수 있는 새 더미 크기들의 집합은 $\{G(S_1), G(S_2), \dots, G(S_k)\}$와 대응된다.

3. Grundy수 정의와 Nim 이동

- Nim 게임에서 현재 더미 크기를 m이라고 하면, 가능한 다음 상태들은 크기가 m' < m인 Nim 더미로의 이동이다.
- 상태 S에서 가능한 이동으로 얻을 수 있는 Nim 더미들의 크기 집합은 $\{G(S_1), G(S_2), \dots, G(S_k)\}$와 같다.
- 따라서 Nim 게임의 관점에서, S와 동등한 Nim 더미의 크기 m은 이 집합에 포함되지 않는 가장 작은 정수, 즉

$$m = mex\{G(S_1), G(S_2), \dots, G(S_k)\}$$

이것이 바로 정의된 G(S)와 일치한다.

귀납법에 의해 상태 S는 크기 G(S)인 Num 더미와 동등하다.

게임의 합성과 XOR 연산 증명

목표 : 여러개의 독립적인 무모순 게임 $S_1, S_2, \dots, S_n$의 합 S = $S_1 + S_2 + \cdots + S_n$에 대해
$G(S) = G(S_1) \oplus G(S_2) \oplus \cdots \oplus G(S_n)$가 성립함을 보이자.

주장 : 합성 게임 $S_1 + S_2$가 크기 $G(S_1) \oplus G(S_2)$인 Nim 게임과 동등하다.

1. Nim 동등성

- $S_1 \equiv G(S_1)$의 Nim 더미 크기, $S_2 \equiv G(S_2)$의 Nim 더미 크기임을 앞서 보였다.
- 따라서 $S_1 + S_2 \equiv G(S_1)$의 Nim 더미 크기 + $G(S_2)$의 Nim 더미 크기라고 볼 수 있다.

2. Nim 더미의 합성 특성

- Nim 게임 이론에서 두 Nim 더미의 합성 상태는 두 더미를 하나의 게임으로 보는 것과 같다.
- 두 Nim 더미 H_1 (크기 $G(S_1)$)와 H_2 (크기 $G(S_2)$)의 합성 상태는 전체적으로 크기 $G(S_1) \oplus G(S_2)$인 하나의 Nim 더미와 게임적으로 동등하다.
- 이 사실은 Nim 게임의 기본 성질 중 하나이며, 두 더미의 합성 결과는 XOR 연산을 통해 하나의 더미 크기로 환원된다.

3. 동등성 전이

- $S_1 + S_2$ 가 두 Nim 더미 $H_1 + H_2$와 동등하고, $H_1 + H_2$가 크기 $G(S_1) \oplus G(S_2)$인 Nim 더미와 동등하다.
- 결론적으로 $S_1 + S_2 \equiv \left(G(S_1) \oplus G(S_2)\right)$ Nim 더미의 크기이다.
- 이 동등성은 합성 게임 $S_1 + S_2$ 의 Grundy수가 $G(S_1 + S_2) = G(S_1) \oplus G(S_2)$ 임을 의미한다.

게임의 합성과 XOR 연산 증명

귀납 가정 : k개의 게임에 대해 $G(S_1 + S_2 + \cdots + S_k) = G(S_1) \oplus G(S_2) \oplus \cdots \oplus G(S_k)$ 이 성립한다.

K+1개의 게임 $S_1, S_2, \ldots, S_k, S_{k+1}$가 있다고 생각하자.

1. k개의 게임 합 $S' = S_1 + S_2 + \cdots + S_k$
   - 귀납 가정에 의해 $G(S') = G(S_1) \oplus G(S_2) \oplus \cdots \oplus G(S_k)$임을 알고있다.

2. S_{k+1} 게임에 대해, 앞서 증명한 두 게임합에 대한 결과를 적용한다
   - $G(S' + S_{k+1}) = G(S') \oplus G(S_{k+1})$

3. 2번식에 귀납 가정의 결과를 대입한다.
   - $$\begin{aligned} G(S_1 + S_2 + \cdots + S_k + S_{k+1}) &= G(S' + S_{k+1}) \\ &= G(S') \oplus G(S_{k+1}) \\ &= \left(G(S_1) \oplus G(S_2) \oplus \cdots \oplus G(S_k)\right) \oplus G(S_{k+1}) \\ &= G(S_1) \oplus G(S_2) \oplus \cdots \oplus G(S_k) \oplus G(S_{k+1}) \end{aligned}$$

귀납법에 의해 모든 n개의 독립 무모순게임 $S_1, S_2, \ldots, S_n$ 에 대해서
$G(S_1 + S_2 + \cdots + S_n) = G(S_1) \oplus G(S_2) \oplus \cdots \oplus G(S_n)$ 가 성립한다.