코드포스 7주차 문제 풀이

문제 난이도

	문제 번호	문제 이름	난이도
A	17128	소가 정보섬에 올라온 이유	S2
B	27968	사사의 사차원 사탕 봉지	S2
C	26007	Codepowers	S2
D	22345	누적 거리	G3
E	3013	부분 수열의 중앙값	G3
F	31422	AND, OR, XOR 2	G1
G	20918	좋은 배열 세기	P5
H	26151	NATO 음성 기호와 쿼리	P3

A. 소가 정보섬에 올라온 이유

수학, 구현, 누적 합

A. 소가 정보섬에 올라온 이유

- 이 문제는 원형으로 둘러 앉은 소들의 품질 점수를 이용해 특정 연속한 네 마리 소들의 점수를 곱한 값의 합을 계산하는 문제입니다.
- **초기 S 값 계산**: 모든 연속한 네 마리 소들의 품질 점수를 곱한 값을 계산하여 초기 S를 구합니다.
- 변화 추적: 각 쿼리마다 특정 소의 품질 점수를 변경할 때, 그 소가 속한 네 개의 연속 부분 집합에 대해 S 값을 업데이트합니다.

A. 소가 정보섬에 올라온 이유

- ✓ **쿼리 처리:**
  - ✓ 각 쿼리마다 특정 소의 품질 점수를 변경하고, 그 소가 속한 네 개의 연속 부분 집합에 대해 S 값을 업데이트합니다.
  - ✓ 변경된 품질 점수에 대한 새로운 S 값을 계산합니다.

- ✓ 시간복잡도: O(N + Q)

B. 사사의 사차원 사탕 봉지

이분 탐색, 누적 합

B. 사사의 사차원 사탕 봉지

- **누적합을 사용하여 사탕의 총합 계산:**
  - 사사가 사탕을 꺼내는 횟수에 따른 누적합을 미리 계산해 놓습니다.
  - 누적합 배열을 사용하면 특정 시점까지 꺼낸 사탕의 총합을 빠르게 계산할 수 있습니다.
- **이진 탐색을 사용하여 필요한 꺼내는 횟수 찾기:**
  - 각 아이가 원하는 사탕의 수에 대해, 누적합 배열에서 해당 수를 초과하는 최소 인덱스를 이진 탐색으로 찾습니다.
  - 이 방법을 사용하면 각 아이에 대해 필요한 꺼내는 횟수를 효율적으로 계산할 수 있습니다.

B. 사사의 사차원 사탕 봉지

- ✓ **시간 복잡도:**
  - ✓ 누적합 계산 : O(M)
  - ✓ 각 아이에 대해 이진 탐색 : O(NlogM)
  - ✓ 전체 시간 복잡도 : O(M + NlogM)

C. CODEPOWERS

누적 합

C. Codepowers

- ✓ **핵심 아이디어 :**
  - ✓ **누적합 배열을 사용하여 구간 내에서 레이팅이 K보다 낮은 횟수를 계산**:
    - ✓ 라운드마다 레이팅 변화를 누적하면서 K보다 낮은 경우를 카운트하는 배열을 만듭니다.
    - ✓ 누적합 배열을 통해 특정 구간의 K보다 낮은 레이팅 횟수를 빠르게 계산할 수 있습니다.

C. Codepowers

- ✓ **단계별 구현**
  - ✓ **누적합 배열 생성**:
    - ✓ 주어진 수열 A를 이용하여 라운드마다 레이팅 변화를 누적하고 K보다 낮은 경우를 카운트합니다.
    - ✓ 누적 카운트 배열을 생성하여 각 라운드마다 K보다 낮은 레이팅 횟수를 저장합니다.
  - ✓ **쿼리 처리**:
    - ✓ 주어진 구간 [l, r)에 대해 누적합 배열을 이용하여 빠르게 답을 구합니다.
    - ✓ 구간 내의 K보다 낮은 레이팅 횟수는 sum[r] - sum[l] 로 계산됩니다.

- ✓ **시간 복잡도 :** O(N+M)

D. 누적 거리

수학, 정렬, 이분 탐색, 누적 합

D. 누적 거리

- ✓ 각 후보 모임 장소에 대해 계산을 반복하면 비효율적이므로, 누적합을 이용하여 효율적으로 계산합니다.

- ✓ **입력 데이터 파싱 및 정렬:**
  - ✓ 마을 데이터를 위치를 기준으로 정렬합니다. 이는 이후 계산을 간단하게 하기 위함 입니다.

- ✓ **누적 합 배열 생성**:
  - ✓ 두 개의 누적 합 배열을 생성합니다.
  - ✓ 인구수 누적 합, 인구 이동 거리 합
  - ✓ 이 두 누적 합 배열을 사용하여 각 구간에 대한 총 이동 거리를 빠르게 계산할 수 있습니다.

D. 누적 거리

- ✓ **쿼리 처리**:
  - ✓ 각 후보 모임 장소에 대해 누적 합 배열을 이용하여 총 이동 거리를 계산합니다.
  - ✓ 이분탐색을 사용하여 후보 장소가 위치할 인덱스를 찾고, 해당 인덱스를 기준으로 좌측 및 우측의 누적 합 값을 사용하여 결과를 계산합니다.

- ✓ **시간복잡도:**
  - ✓ **마을 데이터 정렬 :** O(NlogN)
  - ✓ **누적합 배열 생성 :** O(N)
  - ✓ **각 쿼리 처리 :** O(logN)
  - ✓ **총 시간 복잡도 :** O(NlogN + QlogN)

E. 부분 수열의 중앙값

누적 합

E. 부분 수열의 중앙값

- **입력 처리 및 변수 초기화**:
  - N과 B를 입력 받고, 수열 A를 입력 받습니다.
  - 중앙값 B를 기준으로 작은 수의 개수와 큰 수의 개수를 저장할 변수를 초기화합니다. (small, big이라고 합시다.)

- **누적합을 이용한 부분 수열 계산**:
  - 수열을 순회하면서 B보다 작은 수의 개수를 증가시키는 small과 큰 수의 개수를 증가시키는 big 변수를 사용하여 누적 차이를 계산합니다.
  - B보다 작은 수와 큰 수의 개수 차이 big-small을 저장하는 변수를 선언합니다. (dif 라고 합시다.)
  - dif 값을 이용해 현재 위치까지의 누적합 정보를 저장하는 배열(양수 저장 : pos배열, 음수 저장 : neg배열 이라고 합시다.)을 사용하여 중앙값이 B인 부분 수열의 개수를 누적 계산합니다.

E. 부분 수열의 중앙값

- ✓ **중앙값 B 발견 후 처리**:
  - ✓ B를 발견하기 전까지는 pos와 neg 배열에 누적 합 정보를 저장합니다.
  - ✓ B를 발견한 이후부터 dif값을 이용하여 현재 위치까지의 누적 합 정보를 참조하고 중앙값이 B인 부분 수열의 개수를 계산합니다.

- ✓ **시간 복잡도:**
  - ✓ 수열을 한 번 순회하면서 누적합 정보를 업데이트하므로, 시간 복잡도는 O(N)입니다.

F. AND, OR, XOR 2

수학, 누적 합, 조합론, 비트마스킹

F. AND, OR, XOR 2

- ✓ **입력 처리 및 변수 초기화**:
  - ✓ N과 수열 A를 입력 받습니다.
  - ✓ 각 연산에 대한 결과를 저장할 변수를 초기화합니다.
- ✓ **비트별로 계산**:
  - ✓ 주어진 수열에서 각 수는 2^{30} 미만이므로, 비트 수는 최대 30비트입니다.
  - ✓ 각 비트에 대해 AND, OR, XOR 연산을 수행합니다.
  - ✓ 이를 위해 각 비트가 1인지 0인지를 확인하고, 해당 비트가 1일 때와 0일 때의 값을 각각 계산합니다.

F. AND, OR, XOR 2

- **누적 합을 이용한 부분 수열 계산**:
  - 각 비트별로 현재 비트가 1일 때와 0일 때를 분리하여 누적 합을 계산합니다.
  - 현재 비트가 1이면 이전 상태에서 해당 비트가 1인 경우와 그렇지 않은 경우를 누적해서 결과를 갱신합니다.
  - 현재 비트가 0이면 이전 상태에서 해당 비트가 0인 경우와 그렇지 않은 경우를 누적해서 결과를 갱신합니다.
  - 이를 통해 각 비트에 대해 AND, OR, XOR 연산의 결과를 누적합니다.

- **시간 복잡도**:
  - 비트별로 각 원소를 순회하면서 계산하므로, 시간 복잡도는 O(NlogM)입니다. 여기서 M은 최대 비트 수 30입니다.
  - 따라서 전체 시간 복잡도는 O(N * 30) = O(N) 입니다.

G. 좋은 배열 세기

다이나믹 프로그래밍, 누적 합

G. 좋은 배열 세기

- **초기화 및 DP 테이블 설정**:
  - 먼저, DP 테이블 dp[i][j]를 설정합니다. 여기서 dp[i][j]는 길이 i + 1인 좋은 배열의 score가 j이하인 배열의 개수를 의미합니다.
  - 초기값으로, dp[1][i]는 모두 1로 설정합니다. 길이 2인 배열에서 n이 두 번 들어가는 경우로 Score가 항상 0입니다.

- **DP 테이블 채우기**:
  - 길이 2 이상의 배열에 대해 DP 테이블을 채웁니다.
  - 길이 i+1 인 배열을 구성하는 경우, 첫 번째 요소가 1부터 n까지 중 하나이고, 나머지 부분 배열에 대해서는 i길이의 배열이 됩니다.
  - dp[i][j] = dp[i][j – 1] + dp[i - 1][j] – dp[i - 1][j – i -1] (j < i + 1)
  - dp[i][j] = dp[i][j – 1] + dp[i - 1][j] (j >= i + 1)

G. 좋은 배열 세기

- ✓ **누적 합을 이용하여 구간합 계산**:
  - ✓ 주어진 범위 [a, b]에 대해, DP 테이블의 값으로 범위 내의 배열 개수를 구합니다.
  - ✓ dp[n][b] – dp[n][a – 1]로 구할 수 있으며, a가 0인 경우엔 dp[n][b]만 사용합니다.

- ✓ **시간 복잡도**
  - ✓ DP 테이블을 채우는 과정은 $O(N^2)$입니다. N이 최대 1000이므로, 최대 연산수는 1,000,000 입니다.
  - ✓ 각 테스트 케이스를 처리하는 데 필요한 시간은 O(1)입니다.
  - ✓ 총 시간 복잡도는 $O(N^2 + T)$입니다. 여기서 T는 테스트 케이스의 수 입니다.

H. NATO 음성 기호와 쿼리

다이나믹 프로그래밍, 문자열, 이분 탐색, 누적 합

H. NATO 음성 기호와 쿼리

- ✓ NATO 쿼리에 대해 분석해봅시다.
- ✓ 우선 모든 NATO 음성 문자의 길이가 4 이상이므로, 문자열의 길이는 쿼리를 1번 적용시킬 때마다 최소 4배가 됩니다.
- ✓ 출력 쿼리로 주어지는 위치가 최대 10^{18}이므로, 쿼리를 40번만 줘도 문자열의 뒷부분에 대한 정보는 필요가 없어짐을 생각해볼 수 있습니다.

H. NATO 음성 기호와 쿼리

- ✓ 또한 모든 NATO 음성 문자는 대응되는 글자로 시작하므로, 쿼리를 적용해도 바뀌지 않는 무언가가 있음을 기대해볼 수 있습니다.
- ✓ 실제로 그런 특징을 찾아볼 수 있으며, 아래와 같습니다.
- ✓ 편의상 S^k를 S 에 NATO 쿼리를 k번 적용시켰을 때의 문자열로 둡시다.
- ✓ 자명히, S^0 = S 입니다.

H. NATO 음성 기호와 쿼리

- ✓ 이제 S^0에 쿼리를 한 번 적용해봅시다.
- ✓ NATO 음성 문자의 맨 앞 글자는 항상 대응되는 글자와 동일하기 때문에, S^0과 S^1 의 첫 글자는 동일하게 됩니다.
- ✓ 이제 S^1 에 쿼리를 한 번 더 적용해봅시다.
- ✓ 그럼, S^0 의 첫 1글자와 S 1의 첫 1글자가 동일하고, 모든 NATO 음성 문자의 길이가 최소 4 이상이기 때문에, S 1과 S 2의 첫 4글자는 반드시 동일하게 됩니다.
- ✓ 이런 식으로, S^i 과 S^{i+1} 의 첫 4^i 글자는 반드시 동일함을 알아낼 수 있습니다.
- ✓ 출력 쿼리로 주어지는 위치가 최대 10^{18} 이므로, 쿼리를 40번만 줘도 실질적으로 신경 써야 하는 부분은 전혀 바뀌지 않음을 알 수 있습니다.

H. NATO 음성 기호와 쿼리

- ✓ 이제 이를 토대로 DP를 세워봅시다.

- ✓ $LEN_{i,c}$를 글자 c에 NATO 쿼리를 i번 적용시킬 때의 결과 문자열의 길이라고 하면, $LEN_{i,c} = \sum_{c' \in NATO[c]} LEN_{i-1,c'}$ 이 됩니다.

- ✓ 가능한 알파벳의 종류는 26가지고, 신경 써야 하는 NATO 쿼리의 횟수는 40번 정도가 되기 때문에, 이 DP의 공간 복잡도는 O(26×40)이 됩니다.

H. NATO 음성 기호와 쿼리

- ✓ 이제, 지금까지 NATO 쿼리가 k번 적용되었다고 할 때,
- ✓ DP_i 를 현재까지 적용된 쿼리를 기준으로, 초기 문자열 S^0의 첫 i 글자가 차지하는 글자 수라고 하면, $DP_i = DP_{i-1} + LEN_{k,S_i^0}$ 가 됩니다.
- ✓ 실제로 출력할 때는 위 DP를 토대로 초기 문자열의 몇 번째 글자를 펼쳐야 하는지 이진 탐색으로 알아낸 다음, 이에 해당되는 실제 글자를 LEN 을 역추적해나가면서 찾아주면 됩니다.