Report

김준호

**최종 코드 파일은 submission 파일이다. 각 1, 2, 3번 폴더 안의 lib.py 파일과 문제번호.py 파일은 mypy test를 거치기 전의 파일이라 최종 버전은 아니다. (백준 제출에서 정답 처리는 된다.) mypy test를 거친 최종파일은 submission 폴더 안의 파일이다. **

본 report 문서에선 코드 안에서 구현한 메서드의 간단한 메커니즘을 설명한다.

1. 분할정복을 이용한 거듭제곱 문제: 백준 10830 행렬 제곱

def __setitem__(self, key: tuple[int, int], value: int) -> None:

key를 통해 행렬의 특정 위치에 접근한다. key[0]은 행(row), key[1]은 열(col)을 나타낸다.

value를 self.MOD로 나눈 나머지 값으로 설정한다. 이는 행렬 원소를 MOD 값으로 모듈로 연산하여 저장하기 위한 것이다.

def __pow__(self, n: int) -> Matrix:

n이 1인 경우, 행렬의 복사본을 반환한다. 이는 기저 사례(base case)로, n이 1일 때 행렬 자체를 반환한다.

n이 짝수인 경우, self ** (n // 2)를 먼저 계산하여 half_pow에 저장한다. 이후 half_pow @ half_pow로 전체 제곱을 계산한다.

n이 홀수인 경우, self ** (n // 2)를 먼저 계산하여 half_pow에 저장하고, half_pow @ half_pow @ self로 전체 제곱을 계산한다.

계산된 행렬의 각 원소를 self.MOD로 모듈로 연산한 결과로 저장하여 반환한다.

def __repr__(self) -> str:

행렬의 각 행을 순회하며 문자열로 변환한다. 각 행(row)은 리스트 형태이므로, map(str, row)를 사용해 각 원소를 문자열로 변환한 후, ' '.join()을 사용해 공백으로 구분된 하나의 문자열로 만든다.

각 행을 문자열로 변환한 결과를 다시 '₩n'.join()을 사용해 새로운 행(row)으로 구분된 하나의 큰 문자열로 만든다.

최종적으로 행렬의 각 행이 줄바꿈 문자(₩n)로 구분된 하나의 큰 문자열이 반환된다.

이 메서드는 행렬 객체를 프린트하거나 문자열로 변환할 때 유용하게 사용된다.

위 메서드들을 통해 Matrix 클래스는 행렬의 특정 위치 값 설정, 행렬의 거듭제곱 연산, 그리고 행렬의 문자열 표현 기능을 제공하게 된다.

2. Trie 문제: 백준 3080 아름다운 이름

def push(self, seq: Iterable[T]) -> None:

pointer는 현재 노드를 가리킨다. 초기 값은 0으로, 루트 노드를 가리킨다. 시퀀스의 각 element를 순회한다. 현재 노드의 자식 노드들을 순회하면서, element를 가진 자식 노드가 있는지 확인한다. 있으면 pointer를 그 자식 노드의 인덱스로 설정하고, found를 True로 설정한다. 없으면 새로운 노드를 만들어 트라이에 추가하고, 현재 노드의 자식 목록에 그 노드의 인덱스를 추가한다. pointer는 새로 추가된 노드의 인덱스로 설정한다. 시퀀스의 모든 요소를 처리한 후, 마지막 노드를 끝 노드로 설정한다 (is_end = True).

def find_next_index(self, pointer: int, element: T) -> Optional[int]:

현재 노드의 자식 노드들을 순회하면서, element를 가진 노드가 있는지 확인한다. 있으면 그 자식 노드의 인덱스를 반환하고, 없으면 None을 반환한다.

def search(self, seq: Iterable[T]) -> bool:

pointer는 현재 노드를 가리킨다. 초기 값은 0으로, 루트 노드를 가리킨다. 시퀀스의 각 element를 순회한다.

find_next_index 메서드를 사용해 현재 노드에서 element를 가진 다음 노드를 찾는다. 다음 노드가 없으면 (None을 반환하면) False를 반환한다. 다음 노드가 있으면 pointer를 그 노드의 인덱스로 설정한다. 시퀀스의 모든 요소를 처리한 후, 마지막 노드가 끝 노드인지 확인하고, 끝 노드이면 True를 반환한다. 끝 노드가 아니면 False를 반환한다.

def get_common_prefix(str1: str, str2: str) -> str:

def find_prefixes(sorted_names: list[str]) -> list[str]:

get_common_prefix 함수는 두 문자열의 공통 접두사를 반환한다.

find_prefixes 함수는 정렬된 이름 목록에서 각 이름의 고유 접두사를 찾는다.

이름 목록을 정렬한 후, find_prefixes 함수를 통해 각 이름의 고유 접두사를 찾는다.

3. Trie 문제: 백준 5670 휴대폰 자판

Lib.py는 2번과 동일하다.

def count(trie: Trie, query_seq: str) -> int:

pointer는 현재 노드를 가리킨다. 초기 값은 0으로, 루트 노드를 가리킨다. cnt는 버튼 클릭 횟수를 센다. query_seq의 각 element를 순회하면서 다음을 수행한다:

현재 노드의 자식 노드가 두 개 이상이거나 현재 노드가 단어의 끝이면 cnt를 1 증가시킨다. 이는 여러 선택지가 있거나 단어가 끝날 때마다 추가로 버튼을 눌러야 하기 때문이다.

find_next_index 메서드를 사용해 현재 노드에서 element를 가진 다음 노드를 찾는다.

다음 노드를 찾지 못하면 반복을 종료한다. 다음 노드를 찾으면 pointer를 그 노드의 인덱스로 설정한다. 루트 노드의 자식이 하나인 경우를 고려해 cnt를 조정하여 반환한다.

def main() -> None:

input = sys.stdin.read를 통해 모든 입력을 읽어온다.

data = input().split()을 통해 입력을 공백으로 구분하여 리스트로 변환한다.

idx를 사용해 현재 위치를 추적한다.

결과를 저장할 리스트 results를 초기화한다.

입력 데이터 data를 순회한다. 첫 번째 요소는 단어의 수 N을 나타낸다. 다음 N개의 요소는 단어 목록 words이다. Trie 객체를 생성하고, 각 단어를 트라이에 추가한다. count 함수를 사용해 각 단어를 입력하기 위해 필요한 버튼 클릭 횟수를 계산한다. 총 버튼 클릭 횟수를 계산하고, 단어의 수 N으로 나누어 평균 클릭 횟수를 구한다. 결과를 소수점 둘째 자리까지 포맷팅하여 results 리스트에 추가한다.

4. Segment Tree 문제: 백준 2243 사탕상자

def update(self, index: int, value: U):

index 위치의 값을 value로 갱신한다.

트리의 리프 노드에서 시작하여 부모 노드로 거슬러 올라가며 값을 갱신한다.

def query(self, left: int, right: int) -> U:

구간 [left, right)의 값을 쿼리한다.

트리의 리프 노드에서 시작하여 부모 노드로 거슬러 올라가며 구간 합을 계산한다.

def find_kth(self, k: int) -> int:

k번째 원소의 위치를 찾는다.

트리의 루트에서 시작하여 리프 노드까지 내려가며 값을 찾는다.

def change(self, target: int, diff: U, idx: int, start: int, end: int):

특정 위치의 값을 변경한다.

재귀적으로 트리의 노드를 갱신한다.

def print_sum(self, count: int, idx: int, start: int, end: int) -> int:

특정 개수를 가진 원소의 위치를 찾는다.

재귀적으로 트리의 노드를 탐색한다.

def main() -> None:

첫 번째 값은 쿼리의 수 n이다.

SegmentTree 객체를 생성한다. 여기서는 캔디의 맛을 카운트하기 위해 세그먼트 트리를 사용한다.

쿼리를 순회하며 다음을 수행한다:

A가 1이면 B번째로 맛있는 캔디를 찾아 results에 추가하고, 해당 캔디의 개수를 하나 줄인다.

A가 2이면 캔디의 맛 B의 개수를 C만큼 증가시키거나 감소시킨다.

5. Segment Tree 문제: 백준 3653 영화 수집

lib.py는 4번과 같다.

def main() -> None:

(1) 입력 처리:

sys.stdin.read().strip()을 사용해 입력을 읽고, 공백으로 구분하여 리스트 data에 저장한다.

첫 번째 값 T는 테스트 케이스의 수를 나타낸다.

idx 변수를 통해 현재 읽고 있는 위치를 추적한다.

(2) 테스트 케이스 처리:

각 테스트 케이스마다 n (DVD의 수)와 m (시청 요청의 수)를 읽는다.

시청 요청 리스트 movies를 읽는다.

(3) 세그먼트 트리 초기화:

MAX_POS는 최대 위치를 나타낸다 (n + m).

SegmentTree 객체 seg_tree를 초기화한다. 이 트리는 DVD 위치를 관리한다.

position 리스트는 각 DVD의 현재 위치를 저장한다.

(4) DVD 초기 위치 설정:

각 DVD의 초기 위치를 설정한다. DVD i의 초기 위치는 m + i 이다.

이 초기 위치를 세그먼트 트리에 업데이트한다 (update2 메서드 사용).

(5) 요청 처리:

현재 위치를 pos에 저장한다.

seg_tree.query(1, pos)를 사용하여 pos 위치보다 작은 모든 위치의 DVD 개수를 구한다. 이는 현재 DVD가 몇 번째로 위에 있는지를 계산한다.

현재 위치에서 DVD를 제거 (update2(pos, -1)).

current_top을 갱신하여 DVD를 스택의 맨 위로 이동한다.

새 위치를 position 리스트와 세그먼트 트리에 업데이트한다.

6. Segment Tree 문제: 백준 17408 수열과 쿼리 24

def main() -> None:

(1) 입력 처리:

sys.stdin.read().strip()을 사용하여 모든 입력을 읽고, 공백으로 구분하여 리스트 data에 저장한다.

idx 변수를 사용하여 현재 읽고 있는 위치를 추적한다.

(2) 초기화:

첫 번째 값은 배열의 크기 n이다.

다음 n개의 값은 배열 arr의 요소들이다.

그 다음 값은 쿼리의 수 m이다.

(3) 세그먼트 트리 초기화:

세그먼트 트리 seg_tree를 Pair 객체를 저장하도록 초기화한다.

Pair.f_merge 함수는 두 Pair를 합치는 역할을 하고, Pair.default()는 기본값을 제공한다.

배열 arr의 각 요소를 Pair로 변환하여 세그먼트 트리에 업데이트한다.

(4) 쿼리 처리:

쿼리의 수 m만큼 반복하여 쿼리를 처리한다.

query_type이 1이면 업데이트 쿼리이다:

배열 arr의 i번째 값을 v로 업데이트한다.

Pair.f_conv(v)를 사용하여 값을 Pair로 변환한 후 세그먼트 트리에 업데이트한다.

idx를 3만큼 증가시켜 다음 쿼리로 이동한다.

query_type이 2이면 범위 쿼리이다:

범위 [l, r)에 대해 쿼리하여 구간 내 두 번째로 큰 값을 포함한 두 개의 최대 값을 합산한 결과를 result 리스트에 추가한다.

idx를 3만큼 증가시켜 다음 쿼리로 이동한다.

끝.