report

2-Python(1)

1-divide-and-conquer-multiplication

```
from __future__ import annotations
import copy


"""
TODO:
- __setitem__ 구현하기
- __pow__ 구현하기 (__matmul__을 활용해봅시다)
- __repr__ 구현하기
"""


class Matrix:
    MOD = 1000
```

```
    def __init__(self, matrix: list[list[int]]) -> None:
        self.matrix = matrix

    @staticmethod
    def full(n: int, shape: tuple[int, int]) -> Matrix:
        """모든 원소가 n인 행렬을 생성"""
        return Matrix([[n] * shape[1] for _ in range(shape[0]

    @staticmethod
    def zeros(shape: tuple[int, int]) -> Matrix:
        """모든 원소가 0인 행렬을 생성"""
        return Matrix.full(0, shape)

    @staticmethod
    def ones(shape: tuple[int, int]) -> Matrix:
        """모든 원소가 1인 행렬을 생성"""
        return Matrix.full(1, shape)

    @staticmethod
    def eye(n: int) -> Matrix:
        """단위 행렬 생성"""
        matrix = Matrix.zeros((n, n))
        for i in range(n):
            matrix[i, i] = 1
        return matrix

    @property
    def shape(self) -> tuple[int, int]:
        """행렬의 형태를 반환"""
        return (len(self.matrix), len(self.matrix[0]))

    def clone(self) -> Matrix:
        """행렬의 복사본을 생성"""
        return Matrix(copy.deepcopy(self.matrix))

    def __getitem__(self, key: tuple[int, int]) -> int:
        """행렬의 특정 원소에 접근"""
```

```
        return self.matrix[key[0]][key[1]]

    def __setitem__(self, key: tuple[int, int], value: int) -
        """행렬의 특정 원소에 값을 설정"""
        self.matrix[key[0]][key[1]] = value % Matrix.MOD

    def __matmul__(self, matrix: Matrix) -> Matrix:
        """행렬의 곱셈을 수행"""
        x, m = self.shape
        m1, y = matrix.shape
        assert m == m1  # 행렬의 곱셈이 가능하려면 두 행렬의 열 수가

        result = self.zeros((x, y))

        for i in range(x):
            for j in range(y):
                for k in range(m):
                    result[i, j] += self[i, k] * matrix[k, j]

        return result

    def __pow__(self, n: int) -> Matrix:
        """행렬의 n제곱을 수행"""
        result = Matrix.eye(self.shape[0])  # 단위 행렬로 초기화
        base = self.clone()

        while n > 0:
            if n % 2 == 1:
                result = result @ base
            base = base @ base
            n //= 2

        return result

    def __repr__(self) -> str:
        """행렬을 문자열로 변환하여 출력"""
        return '\n'.join(' '.join(str(self[i, j]) for j in ra
```

각각의 행렬 연산들이 어떻게 입력되고 출력되는지 이해하려 노력하였습니다.

2-trie

trie란

트라이(Trie)는 문자열을 저장하고 검색하기 위한 효율적인 데이터 구조입니다.

트라이의 기본 구조

1. **노드(Node)**: 트라이의 각 노드는 하나의 문자를 저장합니다. 루트 노드는 아무 문자도 저장하지 않으며, 하위 노드는 특정 문자를 저장합니다.
2. **간선(Edge)**: 간선은 부모 노드와 자식 노드 사이의 연결을 의미하며, 각 간선은 문자 하나를 나타냅니다.

트라이의 주요 특징

1. **접두사 공유**: 같은 접두사를 가진 문자열들이 같은 경로를 공유하므로 메모리를 절약할 수 있습니다.
2. **빠른 검색**: 문자열의 검색, 삽입, 삭제가 시간 복잡도 O(m)으로 가능하며, 여기서 m은 문자열의 길이입니다.

트라이의 동작 방식

1. **삽입(Insertion)**:
   - 문자열을 삽입할 때, 문자열의 각 문자를 차례로 트리의 노드에 추가합니다.
   - 만약 해당 문자가 이미 트리에 존재하면 새로운 노드를 만들지 않고, 기존 노드를 그대로 사용합니다.
   - 문자열의 끝에 도달하면, 끝 노드에 문자열의 종료를 나타내는 표시를 추가합니다.
2. **검색(Search)**:
   - 검색할 문자열의 각 문자를 차례로 따라가며 트리에서 해당 문자가 존재하는지 확인합니다.
   - 모든 문자를 찾으면 문자열이 트리에 존재한다고 판단합니다.
   - 문자열의 끝에 도달했는지 여부도 확인하여 정확한 검색을 보장합니다.
3. **삭제(Deletion)**:
   - 삭제할 문자열을 따라가며 각 노드를 확인합니다.

- 문자열의 끝에 도달하고, 해당 노드가 더 이상 다른 문자열과 공유되지 않으면 노드를 삭제합니다.
- 삭제 과정에서 부모 노드와의 연결도 고려하여 트리가 유효한 상태를 유지하도록 합니다.

트라이의 장점

- **메모리 효율성**: 접두사를 공유함으로써 메모리를 절약할 수 있습니다.
- **빠른 검색 속도**: 문자열의 길이에 비례하는 시간 복잡도를 가지므로, 큰 데이터 집합에서도 빠른 검색이 가능합니다.
- **자동 완성**: 접두사 기반 검색이 가능하여, 자동 완성 기능 구현에 유용합니다.

트라이의 단점

- **메모리 사용**: 모든 가능한 문자를 고려해야 하므로, 특히 문자 집합이 크거나 문자열이 많을 때 메모리 사용량이 증가할 수 있습니다.
- **복잡성**: 구현이 복잡할 수 있으며, 특히 노드 삭제와 같은 연산이 추가적인 로직을 필요로 할 수 있습니다.

5670 휴대폰 자판

Trie 구조를 이해하고

```
from dataclasses import dataclass, field
from typing import TypeVar, Generic, Optional, Iterable


"""
TODO:
- Trie.push 구현하기
- (필요할 경우) Trie에 추가 method 구현하기
"""

T = TypeVar("T")

@dataclass
class TrieNode(Generic[T]):
    body: Optional[T] = None  # 노드의 값 (예: 문자 또는 숫자)
    children: list[int] = field(default_factory=lambda: [])
```

```
    is_end: bool = False  # 이 노드가 문자열의 끝인지 여부


class Trie(list[TrieNode[T]]):
    MOD = 1000000007

    def __init__(self) -> None:
        super().__init__()
        self.append(TrieNode(body=None))  # 루트 노드 추가

    def push(self, seq: Iterable[T]) -> None:
        """
        seq: T의 열 (list[int]일 수도 있고 str일 수도 있고 등등...)

        트라이에 시퀀스를 삽입하는 메서드
        """
        current = 0
        for char in seq:
            found = False
            for child in self[current].children:
                if self[child].body == char:
                    current = child
                    found = True
                    break
            if not found:
                new_node = TrieNode(body=char)
                self.append(new_node)
                self[current].children.append(len(self) - 1)
                current = len(self) - 1
        self[current].is_end = True

    def count_orderings(self, node_index: int = 0) -> int:
        node = self[node_index]
        if not node.children:
            return 1

        num_children = len(node.children)
        subtree_orderings = 1
```

```
        for child_idx in node.children:
            subtree_orderings *= self.count_orderings(child_i
            subtree_orderings %= self.MOD

        return (self.factorial(num_children) * subtree_orderi

    def factorial(self, n: int) -> int:
        result = 1
        for i in range(2, n + 1):
            result *= i
        return result

    def contains(self, seq: Iterable[T]) -> int:
        count = 0
        current = 0
        for c in seq:
            for child in self[current].children:
                if self[child].body == c:
                    current = child
                    break
            if len(self[current].children) > 1 or self[curren
                count += 1
        return count


import sys


"""
TODO:
- 일단 Trie부터 구현하기
- count 구현하기
- main 구현하기
"""


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

```
    """
    trie - 이름 그대로 trie
    query_seq - 단어 ("hello", "goodbye", "structures" 등)

    returns: query_seq의 단어를 입력하기 위해 버튼을 눌러야 하는 횟수
    """
    pointer = 0
    cnt = 0

    for element in query_seq:
        # 현재 글자에 해당하는 자식 노드를 찾음
        for child_index in trie[pointer].children:
            if trie[child_index].body == element:
                pointer = child_index
                break

        # 현재 노드에서 자식이 2개 이상이거나, 현재 노드 자체가 단어의 끝
        if len(trie[pointer].children) > 1 or trie[pointer].i
            cnt += 1

    return cnt


def main() -> None:
    while True:
        try:
            N = int(sys.stdin.readline())
        except:
            break

        trie = Trie[str]()
        words = []
        for _ in range(N):
            word = sys.stdin.readline().rstrip()
            trie.push(word)
            words.append(word)

        total_presses = sum(count(trie, word) for word in wor
```

```
        print(f"{total_presses / N:.2f}")

if __name__ == "__main__":
    main()
```

3-segment-tree

Segment Tree란

Segment Tree는 배열과 같은 데이터를 구간 단위로 처리할 수 있도록 해주는 자료구조입니다. 주로 구간 합이나 최소값, 최대값 등을 빠르게 계산할 수 있도록 돕습니다. 이진 트리의 구조를 이용하며, O(log N) 시간 복잡도로 구간 쿼리와 구간 업데이트를 처리할 수 있습니다.

Segment Tree는 두 가지 중요한 작업을 처리하는 데 주로 사용됩니다:

1. **구간 쿼리 (Range Query)**: 배열의 특정 구간에 대한 합, 최소값, 최대값 등을 계산합니다.
2. **구간 업데이트 (Range Update)**: 배열의 특정 인덱스나 구간의 값을 수정합니다.

Segment Tree의 구조

- **리프 노드**: 배열의 각 원소를 표현합니다.
- **내부 노드**: 자식 노드들을 이용해 부모 노드를 계산합니다. 예를 들어, 구간 합을 계산하는 트리라면 부모 노드는 두 자식 노드의 합입니다.

구현 전략

1. **생성자 (Constructor)**: 배열 크기에 맞게 트리의 크기를 결정하고, 트리를 초기화합니다.
2. **구간 쿼리 함수 (Query)**: 주어진 구간에 대해 원하는 값을 계산하여 반환합니다.
3. **업데이트 함수 (Update)**: 특정 인덱스의 값을 수정하고, 그로 인해 영향을 받은 상위 노드들을 갱신합니다.

```
from __future__ import annotations

from dataclasses import dataclass, field
from typing import TypeVar, Generic, Optional, Callable
```

```
"""
TODO:
- SegmentTree 구현하기
"""



T = TypeVar("T")
U = TypeVar("U")

class SegmentTree(Generic[T, U]):
    """
    세그먼트 트리(Segment Tree) 클래스입니다.
    주어진 배열에 대해 특정 범위의 값을 빠르게 계산하기 위해 사용됩니다.
    예를 들어, 구간 합, 최소값, 최대값 등을 계산할 수 있습니다.
    """

    def __init__(self, n: int, func: Callable[[T,T], T], T = 
        """
        세그먼트 트리의 초기화 메서드입니다.

        :param data: 원본 데이터 배열입니다.
        :param func: 구간에 대해 계산할 함수입니다. 예: 합계, 최소값,
        :param default: 기본 값으로, 트리의 리프 노드에 해당하는 값이
        """
        self.n = n
        self.func = func  # 트리의 각 노드에서 사용할 함수
        self.default = T  # 트리의 기본 값
        self.tree = [T] * (2 * n)  # 세그먼트 트리 배열 초기화



    def update(self, index: int, value: T):
        """
        특정 인덱스의 값을 업데이트하고 세그먼트 트리를 갱신합니다.

        :param index: 업데이트할 데이터의 인덱스 (0-based index).
        :param value: 새롭게 설정할 값.
        """
```

```
        # 리프 노드에서 값을 업데이트합니다.
        index += self.n
        self.tree[index] += value

        # 트리의 상위 노드들을 업데이트합니다
        while index > 1:
            index //= 2
            self.tree[index] = self.func(self.tree[2 * index]

    def find_kth(self, k: int) -> int:
        index = 1
        while index < self.n:
            if self.tree[index * 2] >= k:
                index = index * 2
            else:
                k -= self.tree[index * 2]
                index = index * 2 + 1
        return index - self.n

    def query(self, left: int, right: int) -> T:
        """
        주어진 범위 [left, right) 내의 구간 값을 계산합니다.

        :param left: 구간의 시작 인덱스 (포함, 0-based index).
        :param right: 구간의 끝 인덱스 (미포함, 0-based index).
        :return: 구간 [left, right) 에 대한 계산 결과.
        """
        result = self.default

        # 인덱스를 리프 노드에 맞춥니다.
        left += self.n
        right += self.n

        while left < right:
            # left가 홀수라면, 현재 노드를 결과에 포함시키고, 다음 구간
            if left % 2 == 1:
                result = self.func(result, self.tree[left])
                left += 1
```

```
            # right가 홀수라면, 현재 노드를 결과에 포함시키고, 다음 구
            if right % 2 == 1:
                right -= 1
                result = self.func(result, self.tree[right])

            # 상위 노드로 이동합니다.
            left //= 2
            right //= 2

        return result

    def update_17408(self, index: int, value: T):
        """
        특정 인덱스의 값을 업데이트하고 세그먼트 트리를 갱신합니다.

        :param index: 업데이트할 데이터의 인덱스 (0-based index).
        :param value: 새롭게 설정할 값.
        """
        # 리프 노드에서 값을 업데이트합니다.
        index += self.n
        self.tree[index] = value

        # 트리의 상위 노드들을 업데이트합니다
        while index > 1:
            index //= 2
            self.tree[index] = self.func(self.tree[2 * index]
```

2243 사탕상자

trie 구조에 대한 이해를 바탕으로 A가 1인 경우는 사탕상자에서 사탕을 꺼내는 경우, A가 2인 경우는 사탕을 넣는 경우에 따른 노드 탐색 및 업데이트를 공부하였습니다.

```
import sys


"""
```

```
TODO:
- 일단 SegmentTree부터 구현하기
- main 구현하기
"""

def main() -> None:
    # 입력을 처리합니다.
    input = sys.stdin.read
    data = input().split()
    n = int(data[0])  # 수정이가 사탕상자에 손을 댄 횟수

    # 세그먼트 트리 초기화
    # 세그먼트 트리에서 각 인덱스는 특정 맛의 사탕 개수를 나타냄
    size = 2**20
    seg_tree: SegmentTree[int, int] = SegmentTree(size, lambd
    index = 1


    for _ in range(n):
        query = int(data[index])

        if query == 1:
            # 사탕 꺼내기: B번째로 맛있는 사탕을 꺼냄
            B = int(data[index + 1])
            kth_candy = seg_tree.find_kth(B)
            print(kth_candy)
            seg_tree.update(kth_candy, -1)
            index += 2


        elif query == 2:
            # 사탕 넣기 또는 빼기: 맛이 B인 사탕을 C개 추가 또는 제거
            B, C = int(data[index + 1]), int(data[index + 2])
            seg_tree.update(B, C)
            index +=3
```

```
if __name__ == "__main__":
    main()
```

3653 영화 수집

SegmentTree에 데이터를 추가하고 이를 stack처럼 관리하고 추가하는 것을 공부하였습니다.

```
from lib import SegmentTree
import sys


"""
TODO:
- 일단 SegmentTree부터 구현하기
- main 구현하기
"""

def main() -> None:
    input = sys.stdin.read
    data = input().split()

    index = 0
    test_cases = int(data[index])
    index += 1

    results = []

    for _ in range(test_cases):
        n = int(data[index])
        m = int(data[index + 1])
        index += 2

        movies_to_watch = list(map(int, data[index:index + m]
        index += m
```

```
        # DVD의 초기 상태를 설정 (1부터 n까지의 DVD를 스택처럼 관리)
        total_size = n + m

        segment_tree: SegmentTree = SegmentTree(total_size + 
                                            lambda a, b: a + 
        place = [0] * (n + 1)

        # DVD의 번호를 인덱스와 연결하여 관리
        for i in range(1, n + 1):
            place[i] = m + i
            segment_tree.update(place[i], 1)

        current_top = m
        result = []


        for movie in movies_to_watch:
            movie_index = place[movie]  # 0-based index로 변환

            # 꺼낼 때, 영화의 위에 몇 개의 DVD가 있는지 쿼리
            num_above = segment_tree.query(1, movie_index)
            result.append(str(num_above))

            # 꺼낸 DVD를 가장 위로 이동
            # 현재 DVD의 개수를 감소시키고, 제일 위로 이동
            segment_tree.update(movie_index, -1)  # 현재 DVD의
            current_top -= 1
            segment_tree.update(current_top, 1)
            place[movie] = current_top

        results.append(" ".join(result))

    print("\n".join(results))

if __name__ == "__main__":
    main()
```

17408 수열과 쿼리

기본적인 SegmentTree와 pair에 대한 SegmentTree에 대한 구성요소를 공부하였습니다.

```
from lib import SegmentTree
import sys


"""
TODO:
- 일단 SegmentTree부터 구현하기
- main 구현하기
"""


class Pair(tuple[int, int]):
    """
    힌트: 2243, 3653에서 int에 대한 세그먼트 트리를 만들었다면 여기서는
    """
    def __new__(cls, a: int, b: int) -> 'Pair':
        return super().__new__(cls, (a, b))

    @staticmethod
    def default() -> 'Pair':
        """
        기본값
        이게 왜 필요할까...?

        기본값을 반환합니다. 초기화 시에 사용됩니다.
        """
        return Pair(0, 0)

    @staticmethod
    def f_conv(w: int) -> 'Pair':
        """
        원본 수열의 값을 대응되는 Pair 값으로 변환하는 연산
```

```
        이게 왜 필요할까...?

        주어진 값을 Pair 형식으로 변환합니다. 두 번째 값은 항상 0입니다.
        """
        return Pair(w, 0)

    @staticmethod
    def f_merge(a: 'Pair', b: 'Pair') -> 'Pair':
        """
        두 Pair를 하나의 Pair로 합치는 연산
        이게 왜 필요할까...?

        두 Pair 객체를 병합하여 가장 큰 두 값을 가진 Pair를 반환합니다.
        """
        return Pair(*sorted([*a, *b], reverse=True)[:2])

    def sum(self) -> int:
        return self[0] + self[1]


def main() -> None:
    input = sys.stdin.read().strip()

    data = input.split()

    index = 0

    # 수열의 크기 N 읽기
    N = int(data[index])
    index += 1

    # 수열 A 읽기
    A = list(map(int, data[index:index + N]))
    index += N

    # 쿼리의 개수 M 읽기
    M = int(data[index])
    index += 1
```

```
    # SegmentTree를 초기화
    tree: SegmentTree[Pair, int] = SegmentTree(N, Pair.f_merg

    for i in range(N):
        tree.update_17408(i, Pair.f_conv(A[i]))

    results = []

    # 쿼리 처리
    for i in range(M):
        q_type = int(data[index])
        if q_type == 1:
            # 1 i v: Ai를 v로 바꾼다.
            i = int(data[index+1]) - 1
            v = int(data[index+2])
            tree.update_17408(i, Pair.f_conv(v))
            index += 3
        elif q_type == 2:
            # 2 left right: left ≤ i < j ≤ right 만족하는 모든 A
            left = int(data[index+1]) - 1
            right = int(data[index+2])
            result = str(tree.query(left, right).sum())
            results.append(result)
            index += 3

    # 결과 출력
    print("\n".join(results))


if __name__ == "__main__":
    main()
```