report

동작방식보다는 시행착오와 어려웠던 포인트 위주로 서술해봤습니다!

segmenttree

```
class SegmentTree(Generic[T, U]):
    def __init__(self,
                 arr: list[T],
                 default_value: T,
                 merger: Callable[[T, T], T],
                 ):
        self.merger: Callable[[T, T], T] = merger
        self.default_value: T = default_value
        self.tree: list[T] = [default_value for i in range(len(arr) * 4)]
        self.length: int = len(arr)

        # init tree
        for idx, val in enumerate(arr):
            self.insert(idx, val)

    def _query(self, start_idx: int, end_idx: int, node_start: int, node_end: int, node: int) -> T:
        # 겹치는 게 존재하지 않을때
        if node_end < start_idx or end_idx < node_start:
            return self.default_value
        # node범위가 query범위에 포함될때
        if start_idx <= node_start and node_end <= end_idx:
            return self.tree[node]

        node_mid: int = (node_start + node_end) // 2
        return self.merger(
            self._query(start_idx, end_idx, node_start, node_mid, 2 * node),
            self._query(start_idx, end_idx, node_mid + 1, node_end, 2 * node + 1)
        )

    # leaf node를 우선찍고
    # leafnode에서 default value반환
    # 그뒤로 다시 올라오며 modify 실행
    def _modify(self, target_idx: int, node_start: int, node_end: int, node: int, modify_val: T,
                modifier: Callable[[T, T], T]) -> None:
        if not (node_start <= target_idx <= node_end):
            return

        if node_start == node_end:
            self.tree[node] = modifier(self.tree[node], modify_val)
            return

        node_mid: int = (node_start + node_end) // 2
        self._modify(target_idx, node_start, node_mid, 2 * node, modify_val, modifier)
        self._modify(target_idx, node_mid + 1, node_end, 2 * node + 1, modify_val, modifier)

        self.tree[node] = self.merger(self.tree[2 * node], self.tree[2 * node + 1])
```

modify함수를 원래 root에서 시작해서 leaf로 내려가면서 값을 바꾸는 식으로 구현을 했었는 데 17408번에서 이런 방식으로는 구현이 어려워서 leaf에서 root으로 올라가면서 값을 바꾸는 식으로 구현을 바꿨습니다.

그리고 2243번에서 0값을 가지는 백만 길이의 리스트를 segment트리로 초기화 하는 데서 시간이 너무 오래걸려서 아래와 같이 우선 길일가 1인 리스트로 segment트리를 만든다음 주먹구구식으로 초기화 하는 방법을 선택했습니다.

```
MAX: int = 1000000 + 1
# 시간초과이슈로 생성자말고 수동으로 초기화
tree: SegmentTree[int, int] = SegmentTree(
    [0],
    0,
    lambda x, y: x + y
)
```

```
tree.tree = [0] * (MAX * 4)
tree.length = MAX
```

---

trie

```
class Trie(list[TrieNode[T]]):
    def __init__(self) -> None:
        super().__init__()
        self.append(TrieNode(body=None))

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

        action: trie에 seq을 저장하기
        """         cur: int = 0
        for val in seq:
            found: bool = False
            for child_idx in self[cur].children:
                if self[child_idx].body == val:
                    cur = child_idx
                    found = True
                    break
            if not found:
                new_idx = len(self)
                self.append(TrieNode(body=val))
                self[cur].children.append(new_idx)
                cur = new_idx

        self[cur].is_end = True
```

처음에는 아래와 같이 재귀적으로 구현했었습니다. 뒤의 어려웠던 점 파트에서 서술하겠지만 재귀적으로 구현한 것이 제 4시간을 날려버렸습니다...

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

        action: trie에 seq을 저장하기
        """
        self.retrieve(0, seq, 0)
        pass

    def retrieve(self, node_idx: int, seq: Iterable[T], idx: int):

        # retrieve body
        for child_idx in self[node_idx].children:
            if self[child_idx].body == seq[idx]:
                if len(seq) - 1 == idx:
                    self[node_idx].is_end = True
                else:
                    self.retrieve(child_idx, seq, idx + 1)
                return

        # 매칭되는 것이 없을 경우
        newNode = TrieNode()
        newNode.body = seq[idx]
        self.append(newNode)
        # node.children.append(len(super()))
        self[node_idx].children.append(len(self) - 1)

        # 끝일경우 처리
        if len(seq) - 1 == idx:
            self[len(self) - 1].is_end = True
        else:
            self.retrieve(len(self) - 1, seq, idx + 1)
```

10830은 1번만 거듭제곱할때도 1000으로 나눈나머지를 구해야한 다는 포인트를 챙기는 게 중요했습니다.

어려웠던 점

파이썬에서 재귀 성능 이슈(3080번)

평소에 재귀적으로 구현하는 것을 좋아해서 trie 트리에서 push연산도 재귀적으로 구현했었습니다. 근데 계속 시간초과가 떠서 삽질을 계속했습니다. 혹시 몰라서 테스트를 진행해봤습니다.

테스트 내용은 3080문제의 상황으로 랜덤한 3000길이 문자 1000개를 trie에 push하는 작업입니다.

```
start~~~
recur trie: 4.325122375
start~~~
trie : 0.4000921669999995
```

gc같은 외부적인 환경에 영향을 받는 테스트였지만 결과를 보면 재귀가 거의 10배 느린 것을 알 수 있었습니다. 따라서 이문제 이후로는 모든 문제에서 재귀를 최대한

3080 메모리 초과

찾아보니 3080이 c++에서도 메모리 부족이슈가 있는 타이트한 문제인데 안그래도 python으로 백준을 푸는 건 처음인데 많은 고생을 했습니다. 3080이 메모리가 부족한 것은 알고 있어서 여러 꼼수를 구현했습니다.

1. 문자열 대신 아스키코드로 trie 트리 구성
2. 연산에 필요없는 child 개수가 1개인 노드들을 제거하는 최적화
   를 해봤었는 데 일차적으로 맨붕이 왔었습니다.
   우선은 pypy로 실행하는 게 1차적인 문제점이였고 근본적인 문제는 앞서예기했던 재귀가 메모리도 많이 차지하는 것이 문제였습니다. 여기서 너무 시간을 많이 썻던 것 같습니다.
   마지막으로 위에서 2번의 방법보다 들어보지도 못한 LCP로 메모리 줄이는 방법이 있는 것 보고 경악을 금치 못했습니다...