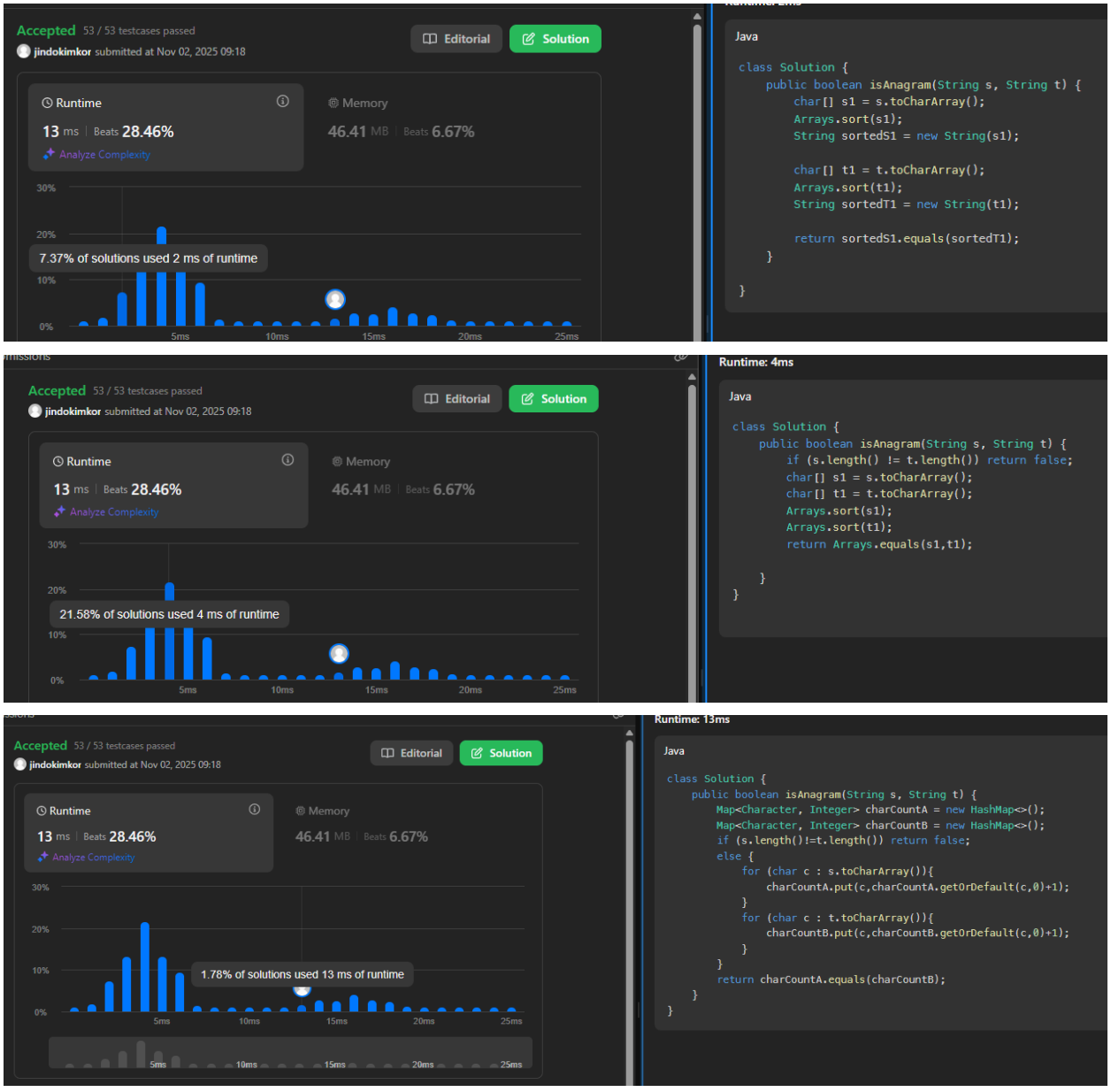

왜 **Hashmap**이 측정에서 더 **Runtime**이 오래 걸린것으로**??**

1. 작은 입력 크기**:**

LeetCode 테스트: 보통 n < 1000

O(n) vs O(n log n):
n = 100: 100 vs 664
n = 1000: 1000 vs 9966

→ 차이가 크지 않음

2. HashMap 오버헤드:

Map<Character, Integer> map = new HashMap<>();
map.put(c, map.getOrDefault(c, 0) + 1);

// 내부 동작:
- Hash function 계산
- Collision 처리
- Resizing (필요시)
- Object 생성 (Integer boxing)

Arrays.sort():

Arrays.sort(chars); // Primitive type
// - 최적화된 C/Assembly
// - No object creation
// - Cache-friendly

3. 실제 상수 계수:

HashMap: O(n) 이지만 상수가 큼
- 각 operation마다 hash, get, put

Sorting: O(n log n) 이지만 상수가 작음
- Highly optimized
- In-place

언제 **HashMap**이 더 빠를까**?**

입력이 클 때:

n = 1,000,000:
HashMap: ~1,000,000 operations
Sorting: ~20,000,000 operations

→ HashMap이 확실히 빠름

알파벳이 적을 때**:**

26개 문자만 → int[26] 배열 사용
→ 훨씬 빠름 (hash 계산 없음)

수정된 분석**:**

방법	**Big-O**	실제 **(**작은 입력**)**	실제 **(**큰 입력**)**
Sorting	O(n log n)	빠름 **(3-4ms)**	느림
HashMap	O(n)	느림 (13ms)	빠름
int[26]	O(n)	제일 빠름 **(1-2ms)**	제일 빠름

가장 빠른 방법 **(**알파벳만**):**

```
class Solution {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) return false;

        int[] count = new int[26];  // HashMap 대신

        for (char c : s.toCharArray()) {
            count[c - 'a']++;
        }

        for (char c : t.toCharArray()) {
            count[c - 'a']--;
        }

        for (int n : count) {
            if (n != 0) return false;
        }

        return true;
    }
}
```

왜 빠른가:

- No hash function
- No object creation
- Direct array access
- Cache-friendly

실전 전략 (수정):

Assessment에서:

```
// 1순위: Sorting (구현 빠름, 성능 좋음)
Arrays.sort(s1);
Arrays.sort(t1);
return Arrays.equals(s1, t1);

// 2순위: int[26] (제일 빠름, 알파벳만)
int[] count = new int[26];

// 3순위: HashMap (범용적, 이론적으로 optimal)
Map<Character, Integer> map = new HashMap<>();
```

정리:

네 분석이 맞아:

- 실제로 Sorting이 더 빠름 (작은 입력)
- Big-O는 이론, 실제는 다를 수 있음

내 실수:

- 상수 계수 무시
- 입력 크기 고려 안 함
- 실제 구현 최적화 간과

배울 점:

- Big-O ≠ 실제 성능
- 벤치마크 중요
- 작은 입력: 간단한 게 빠름
- 큰 입력: Big-O가 중요