ChatGPT 알고리즘 스터디: 6주차

투 포인터와 슬라이딩 윈도우, 이분 탐색과 파라메트릭 서치

도입 (Introduction)

지난 주에는 **정렬 알고리즘**의 원리와 구현을 공부하여, 선택 정렬·퀵 정렬·병합 정렬 등의 방법론과 시간 복잡도 분석을 다루었습니다. 정렬은 데이터 처리를 위한 기본 단계로서, 알고리즘 효율성의 중요성을 체감할 수 있는 주제였습니다. 이번 주에는 **효율적인 탐색 기법**을 배우는 것을 목표로 합니다. 특히, 배열이나 리스트 상에서 부분적인 해답을 빠르게 찾는 **투 포인터(two pointers)** 및 **슬라이딩 윈도우(sliding window)** 기법과, **이분 탐색(binary search)** 및 이를 확장한 **파라메트릭 서치(parametric search)** 기법을 중점적으로 다룹니다. 이를 통해 이중 루프를 단일 루프로 줄이는 방법과 탐색 범위를 로그 수준으로 줄이는 기법을 익혀, **시간 복잡도를 획기적으로 개선**하는 방법을 학습할 것입니다. 또한 **Meet-in-the-Middle**, **삼분 탐색(ternary search)**, **슬라이딩 윈도우 + 자료구조** 및 **이분 탐색 + 그리디**와 같은 추가 개념들도 간단히 소개하여, 다양한 문제 상황에서 적용할 수 있는 알고리즘 전략을 넓혀봅니다. 이번 강의 노트를 통해 제공되는 예제 코드와 단계별 설명을 따라 실습하면서, 각 개념을 확실히 체득해 보세요.

목차

1. **투 포인터와 슬라이딩 윈도우**
2. 1.1 개념 설명 및 시각적 직관
3. 1.2 대표 문제 유형
4. 1.3 코드 예시 (C++ / Java / Python)
5. 1.4 시간 복잡도 및 적용 전략
6. **이분 탐색과 파라메트릭 서치**
7. 2.1 개념 설명 및 시각적 직관
8. 2.2 대표 문제 유형
9. 2.3 코드 예시 (C++ / Java / Python)
10. 2.4 시간 복잡도 및 적용 전략
11. **기타 알고리즘 개념**
12. 3.1 Meet in the Middle (중간에서 만나기)
13. 3.2 삼분 탐색 (Ternary Search)
14. 3.3 슬라이딩 윈도우 + 자료구조 활용
15. 3.4 이분 탐색 + 그리디 결합
16. **결론 및 다음 주 미리보기**

---

1. 투 포인터와 슬라이딩 윈도우

1.1 개념 설명 및 시각적 직관

투 포인터(two pointers) 기법은 1차원 배열에서 두 개의 포인터(인덱스)를 활용하여 원하는 결과를 효율적으로 찾는 알고리즘을 말합니다 [1]. 일반적으로 한 포인터는 **시작 지점(start)**을, 다른 하나는 **끝 지점(end)**을 가리키며, 이 두 포인터를 적절히 이동시키면서 문제의 조건을 만족하는 구간이나 원소들을 찾아냅니다. 투 포인터를 사용하면 이중 루프(`O(N^2)`)를 필요한 만큼의 단일 루프(`O(N)`)로 줄일 수 있어, **전체 탐색 대비 성능을 크게 향상**시킬 수 있습니다 [2].

투 포인터의 전형적인 예시로, **정렬된 배열에서 두 수의 합이 특정 값 X가 되는 원소 쌍을 찾는 문제**를 생각해봅시다. 한 포인터를 배열 **왼쪽 끝**(가장 작은 값)에서 시작하고, 다른 포인터를 **오른쪽 끝**(가장 큰 값)에서 시작합니다. 두 포인터가

가리키는 값의 합을 계산하여 X와 비교합니다. 만약 합이 X와 같다면 답을 찾은 것이고, 합이 X보다 작다면 왼쪽 포인터를 오른쪽으로 한 칸 이동시켜 합을 키웁니다. 반대로 합이 X보다 크다면 오른쪽 포인터를 왼쪽으로 한 칸 이동시켜 합을 줄입니다. 이렇게 하면 모든 경우를 일일이 비교하지 않고도 원하는 쌍을 효율적으로 찾아낼 수 있습니다. 포인터들이 배열 양쪽에서 가운데로 움직이는 과정을 **시각적으로 표현**하면, 마치 두 사람이 줄의 양 끝에서 시작해 서로를 향해 걸어가며 조건에 맞는 지점을 찾는 것과 비슷합니다 (왼쪽 사람이 합이 작으면 앞으로 한 걸음, 오른쪽 사람이 합이 크면 앞으로 한 걸음 다가오는 식입니다).

슬라이딩 윈도우(sliding window) 기법은 투 포인터와 유사한 맥락에서 사용되지만, **고정된 크기의 구간(window)**을 배열 위에서 왼쪽부터 오른쪽으로 **밀어가며(slide)** 이동하는 방법을 주로 가리킵니다. 예를 들어, 길이가 k인 부분 배열의 합을 구하는 문제에서는 처음 1번~k번 요소의 합을 계산한 뒤, 윈도우를 한 칸 오른쪽으로 움직여 2번~k+1번 요소의 합을 빠르게 구할 수 있습니다. 이때 이전 구간 합에서 빠지는 값과 새로 들어오는 값을 조정하여 **중복 계산을 피하는 것**이 슬라이딩 윈도우의 핵심입니다 [3]. 슬라이딩 윈도우는 구간의 크기가 **일정**할 때 주로 논하지만, 문제에 따라 윈도우 크기가 가변적일 수도 있습니다. 특히 **투 포인터를 활용한 가변 크기 구간** 처리 역시 흔히 슬라이딩 윈도우라고 부르기도 합니다. 예를 들어, 배열에서 **연속된 부분 합이 특정 조건을 만족하는** 가장 짧은 구간을 찾는 경우, 시작 포인터와 끝 포인터를 이동시켜 가며 현재 구간의 합을 유지하고 조절하는 방식이 쓰이는데, 이는 투 포인터이면서 동시에 윈도우를 슬라이딩하는 방법이라 볼 수 있습니다.

핵심은, 투 포인터/슬라이딩 윈도우 기법은 **배열을 한 번 순회**하면서 두 지점을 조절함으로써 문제를 해결한다는 점입니다. 한 포인터가 다른 포인터를 앞서 나가지 않도록 조건을 부여하여 (`start <= end` 등의 조건 유지) 올바른 탐색을 진행하며, 필요에 따라 두 포인터가 모두 한 방향(예: 오른쪽)으로 이동하기도 하고 반대 방향으로 이동하기도 합니다. 이러한 **시각적 직관**을 정리하면: 하나의 포인터가 다른 포인터를 쫓아가거나 서로를 향해 이동하면서, 그 사이의 구간 또는 조합을 지속적으로 평가하는 모습이라고 할 수 있습니다. 이러한 접근은 **연속된 구간 합, 두 수/세 수의 조합, 병합 과정** 등 다양한 상황에서 쓰이며, 뒤이어 대표 문제들과 코드 예시에서 구체적으로 확인해보겠습니다.

1.2 대표 문제 유형

투 포인터와 슬라이딩 윈도우를 활용할 수 있는 대표적인 문제 유형은 다음과 같습니다:

- **두 수의 합 찾기**: 정렬된 배열에서 두 원소의 합으로 특정 값을 만들 수 있는지 찾는 문제. 투 포인터를 이용하면 양 끝에서 포인터를 움직이며 `O(N)`에 해결 가능합니다. (예: 합이 0이 되는 두 수 찾기 등)
- **연속 부분합 문제**: 배열에서 **특정 합**을 가지는 연속된 부분 배열을 찾거나, 특정 조건(예: 합이 M 이상, 혹은 정확히 M)이 되는 부분 배열의 개수/길이를 구하는 문제. 투 포인터로 현재 구간의 합을 유지하며 start와 end를 조절하여 탐색합니다 [2]. (예: 백준 2003 수들의 합 2, 백준 1806 부분합 등의 문제)
- **세 수의 합 (Three-sum) 및 네 수의 합**: 정렬된 배열에서 세 개 혹은 네 개의 수를 뽑아 특정 합을 만드는 문제. 두 포인터 기법을 응용하여, 첫 번째 숫자를 결정한 후 나머지 부분을 투 포인터로 해결하는 방식으로 효율 개선이 가능합니다. (예: LeetCode 3Sum 문제 등)
- **병합 과정 (Merge)**: 두 개의 **정렬된 배열을 하나로 병합**하는 문제는 각각의 배열에 대한 포인터 두 개를 이용해 작은 쪽부터 결과에 넣는 방식으로 구현됩니다. 이는 **병합 정렬**의 핵심 과정으로, 투 포인터가 대표적으로 활용되는 알고리즘입니다.
- **고정 길이 슬라이딩 윈도우 문제**: 윈도우 크기가 k로 고정된 상태에서 최댓값/최솟값, 합 등을 구하는 문제. 한 번 계산한 윈도우의 정보를 이용해 다음 윈도우의 값을 빠르게 갱신합니다. (예: 크기가 k인 부분 배열의 합 최대값 찾기, 이동 평균 계산 등)
- **문자열의 부분 패턴 탐색**: 문자열에서 특정 조건을 만족하는 부분 문자열을 찾는 문제에 슬라이딩 윈도우+투 포인터를 사용할 수 있습니다. 예를 들어, **중복 문자 없는 최장 부분 문자열** 문제에서는 윈도우 내의 문자 출현을 집합(Set)이나 맵(Map)으로 관리하면서, 윈도우를 확장/축소하여 해결합니다. 이 경우 윈도우에 자료구조를 결합하는 테크닉은 뒤에 별도로 언급될 예정입니다. (예: LeetCode Longest Substring Without Repeating Characters)

위의 문제 유형들은 투 포인터/슬라이딩 윈도우 기법이 **효율적인 대안**이 되는 상황들입니다. 공통적으로, **배열이나 리스트를 양 끝 또는 앞뒤 두 방향에서 좁혀가며 탐색**하거나, **구간을 조금씩 이동**시키며 해를 찾는 방식이 적용됩니다. 각 문

제에 맞게 포인터의 이동 규칙(언제 어느 포인터를 이동시키는지)을 정의하는 것이 중요하며, 이에 익숙해지면 많은 유사한 문제들을 자연스럽게 최적화된 방식으로 풀 수 있게 됩니다.

1.3 코드 예시 (C++ / Java / Python)

아래의 예시는 **투 포인터와 슬라이딩 윈도우**를 활용한 코드 예제입니다. **문제 시나리오**: 정수 배열에서 **합이 S 이상**이 되는 부분 배열 중 가장 짧은 길이를 구하는 문제를 가정합니다 (예: 백준 1806 부분합 문제와 유사). 이 문제는 배열의 연속 구간 합이 일정 값을 넘어서는 최소 구간을 찾는 것으로, 투 포인터를 사용하면 효율적으로 해결할 수 있습니다.

각 언어별로 동일한 로직을 구현하였으며, C++ -> Java -> Python 순으로 병렬로 코드를 제공합니다. 코드에서 **left**와 **right** 두 개의 포인터(인덱스)를 사용하여 현재 구간을 표현하고, **현재 합**이 조건 `>= S`를 만족할 때 left를 우측으로 이동(구간 축소)하고, 만족하지 않을 때 right를 우측으로 이동(구간 확장)하는 식으로 전체 배열을 탐색합니다. 이때 최소 구간 길이를 지속적으로 갱신하면서 답을 찾아냅니다.

C++ 코드:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main(){
  vector<int> arr = {1, 4, 45, 6, 10, 19};
  int S = 51;
  int n = arr.size();
  int minLen = INT_MAX;
  long long sum = 0;
  int left = 0;
  for(int right = 0; right < n; ++right){
    sum += arr[right];          // 오른쪽 포인터 확장
    while(sum >= S){            // 조건 만족: 구간 합이 S 이상
      minLen = min(minLen, right - left + 1);  // 길이 갱신
      sum -= arr[left++];       // 왼쪽 포인터 이동 (구간 축소)
    }
  }
  if(minLen == INT_MAX){
    cout << "해당 조건을 만족하는 부분 배열이 없습니다." << endl;
  } else {
    cout << "최소 부분 배열 길이: " << minLen << endl;
  }
}
```

Java 코드:

```java
import java.util.*;
class  Main {
  public static void main(String[] args){
    int[] arr = {1, 4, 45, 6, 10, 19};
    int S = 51;
    int n = arr.length;
    int minLen = Integer.MAX_VALUE;
```

```
        long sum = 0;
        int left = 0;
        for(int right = 0; right < n; right++){
            sum += arr[right];              // 오른쪽 포인터 확장
            while(sum >= S){                // 조건 만족 시
                minLen = Math.min(minLen, right - left + 1);  // 최소 길이 갱신
                sum -= arr[left++];         // 왼쪽 포인터 이동 (구간 축소)
            }
        }
        if(minLen == Integer.MAX_VALUE){
            System.out.println("해당 조건을 만족하는 부분 배열이 없습니다.");
        } else {
            System.out.println("최소 부분 배열 길이: " + minLen);
        }
    }
}
```

Python 코드:

```
arr = [1, 4, 45, 6, 10, 19]
S = 51
n = len(arr)
min_len = float('inf')
current_sum = 0
left = 0
for right in range(n):
    current_sum += arr[right]                    # 오른쪽 포인터 확장
    while current_sum >= S:                      # 조건 만족 시
        min_len = min(min_len, right - left + 1)    # 최소 길이 갱신
        current_sum -= arr[left]                 # 왼쪽 포인터 이동 (구간 축소)
        left += 1
if min_len == float('inf'):
    print("해당 조건을 만족하는 부분 배열이 없습니다.")
else:
    print("최소 부분 배열 길이:", min_len)
```

위 코드에서 볼 수 있듯이, **투 포인터를 이용한 윈도우 이동**을 통해 전체 배열을 한 번 순회하면서 원하는 조건을 만족하는 최소 구간을 찾아냅니다. `right` 포인터는 외부 루프에서 0부터 N-1까지 증가하며 배열을 끝까지 탐색하고, `left` 포인터는 내부 `while` 루프에서 조건이 충족될 때마다 오른쪽으로 움직입니다. 이런 구조 덕분에 `left`와 `right` 포인터 각각은 최대 N번씩 움직이고 멈추므로, 전체 알고리즘은 `O(N)`의 시간 복잡도로 동작합니다.

또한 세 언어의 구현을 비교해 보면, **로직은 동일**하지만 언어 특성에 따라 문법만 다를 뿐임을 알 수 있습니다. C++과 Java에서는 배열 길이를 변수로 저장하고 (`n`), Python에서는 `len(arr)`를 직접 사용하거나 코드가 간결합니다. C++에서는 `long long`을 사용해 합을 저장하고, Java에서는 `long`, Python에서는 별도의 타입 구분 없이 큰 정수를 다룹니다. 이러한 세부 차이에도 불구하고, 투 포인터 알고리즘의 본질은 언어에 상관없이 동일하므로, 알고리즘 자체에 대한 이해가 가장 중요합니다.

1.4 시간 복잡도 및 적용 전략

시간 복잡도: 투 포인터와 슬라이딩 윈도우를 적용한 알고리즘의 시간 복잡도는 일반적으로 **선형 시간 (**`O(N)`**)**입니다. 두 포인터가 각각 배열을 한 번씩 진척하면서 전체를 커버하기 때문입니다. 예를 들어 앞서 제시한 부분합 문제의 코드에서 `right` 포인터는 N번 이동하고, `left` 포인터도 총 N번 이내로 이동합니다. 이처럼 이중 루프를 사용하는 완전 탐색이 `O(N^2)` 걸릴 문제를 투 포인터 기법으로 `O(N)`에 해결할 수 있는 경우가 많습니다 [2]. 다만, 투 포인터 기법을 적용하기 위해서는 **문제의 특성**이 중요합니다. 배열이 정렬되어 있다든지 (두 수의 합 문제), 혹은 구간의 크거나 작음에 따라 포인터 이동 방향을 결정할 수 있는 **단조성(monotonicity)**이 존재해야 합니다. 예를 들어 모든 원소가 양수인 배열에서는 부분합을 늘릴 때 합이 단조 증가하므로 합이 목표치 이상/미만인지에 따라 포인터 움직임을 결정할 수 있었습니다. 하지만 배열에 음수 값이 섞여 있으면 합의 증감이 단조하지 않을 수 있어 이 방법이 바로 적용되지 않을 수 있습니다. 이러한 경우에는 투 포인터 대신 **누적 합(prefix sum)** 등의 다른 기법이 필요할 수 있습니다.

적용 전략: 언제 투 포인터/슬라이딩 윈도우를 떠올리면 좋을까요? 일반적으로 다음과 같은 경우 고려해볼 수 있습니다: - **정렬된 배열의 탐색 문제**: 투 포인터는 정렬된 리스트에서 특정 조건을 찾는데 매우 유용합니다. 투 포인터가 양 끝에서 출발하여 조건에 따라 한쪽을 움직이는 패턴을 쉽게 적용할 수 있습니다. - **연속 구간 문제**: 문제에서 "연속된 부분", "연속된 구간" 등의 키워드가 나오고, 그것을 모두 탐색하면 시간이 너무 큰 경우 투 포인터로 구간을 확장/축소하며 답을 찾는 방안을 고려해봅니다. 특히 부분합이나 부분곱, 부분 배열의 특성 등을 묻는 문제들입니다. - **두 개 이상의 배열 병합/비교 문제**: 병합 정렬의 병합 단계나, 두 리스트의 교집합 찾기 등 **두 배열을 동시에** 다루는 문제에서도 각 배열에 포인터를 두고 이동하며 해결할 수 있습니다. - **고정 윈도우의 이동 누적 계산**: 데이터 스트림이나 배열에서 일정 크기의 창을 이동시키며 합/평균/최댓값 등을 계산하는 경우, 이전 윈도우 결과를 활용하여 다음 윈도우 결과를 빠르게 계산하는 전략이 슬라이딩 윈도우입니다. 이 경우 링크드 리스트나 큐로 구현하는 패턴도 있습니다만, 핵심은 한 번의 순회로 모든 윈도우를 다 구하는 것입니다.

마지막으로, 투 포인터/슬라이딩 윈도우를 사용할 때 **경계 조건 관리**에 유의해야 합니다. 예를 들어 start와 end의 초기값 (보통 0에서 시작), 루프 종료 조건 (`end`가 배열 끝을 넘어갈 때 등), 그리고 윈도우 내에 유효한 상태를 유지하는 논리가 중요합니다. 이러한 부분은 연습을 통해 체득해야 하며, 실수로 무한 루프에 빠지지 않도록 `while` 문 조건 설정에도 신경을 써야 합니다. 하지만 익숙해진다면, 이 기법들은 많은 알고리즘 문제에서 효율적인 해법을 설계하는 강력한 도구가 될 것입니다.

---

2. 이분 탐색과 파라메트릭 서치

2.1 개념 설명 및 시각적 직관

이분 탐색(binary search)은 **정렬된 배열이나 범위**에서 탐색 범위를 절반씩 줄여가며 값을 찾는 알고리즘입니다. 이미 여러 번 언급된 바와 같이, 정렬된 데이터에서 특정 값을 찾을 때 **중간값을 확인하여** 목표 값이 중간값보다 작으면 왼쪽 절반을, 크면 오른쪽 절반을 탐색하는 식으로 진행합니다. 이렇게 하면 탐색 범위가 기하급수적으로 줄어들기 때문에 시간 복잡도가 `O(log N)`으로 매우 효율적입니다. 간단한 예로 숫자 맞추기 게임을 떠올려 보면 이해하기 쉽습니다. 1부터 100까지 범위 중 어떤 비밀 숫자가 있을 때, 무작위로 하나씩 찍어 물어보는 대신 항상 남은 범위의 **정가운데** 숫자를 질문하면, 매 번 질문할 때마다 범위가 절반으로 줄어듭니다. 이러한 방식으로는 많아야 7번 (왜냐하면 `2^7 = 128 > 100`)만에 답을 찾을 수 있지요. 이처럼 이분 탐색은 **분할 정복(divide and conquer)** 원리를 탐색에 적용한 대표적인 알고리즘입니다.

이분 탐색이 동작하려면 **데이터가 정렬되어 있어야 함**을 강조합니다. 정렬되지 않은 상태에서는 중간값과 크고 작음을 비교해 절반을 버리는 논리가 성립하지 않기 때문입니다. 또한 배열이 아니라도, 탐색 대상이 되는 값의 조건이 **단조 증가나 감소** 형태로 참/거짓이 나뉠 수 있다면 이분 탐색을 적용할 수 있습니다. 예를 들어 "어떤 값 X 이상인지 아닌지"에 대한 판단이 가능하고 X에 따라 참/거짓 구간이 나뉜다면, 순서대로 정렬된 배열이 없어도 범위 탐색에 이분 탐색을 응용할 수 있습니다. 이러한 아이디어를 확장한 것이 바로 **파라메트릭 서치(parametric search)**입니다.

파라메트릭 서치(parametric search)란, **최적화 문제**를 **결정 문제(예 또는 아니오로 답하는 형태)**로 바꾸어 이분 탐색으로 푸는 기법을 말합니다 [4]. 말이 어렵게 느껴질 수 있는데, 핵심은 어떤 문제에서 구해야 하는 값이 있을 때 그 값을 임의로 하나 가정하고 "이 값으로 조건을 만족시킬 수 있는가?"를 물어보는 함수를 만들어 이분 탐색을 적용하는 것입니다. 예를 들어 보겠습니다. 전형적인 **나무 절단 문제**를 생각해봅시다 (이 문제는 백준 2805 나무 자르기로 잘 알려져 있습니다): 여러 나무가 있고 톱날 높이 H를 정하면 H 위의 나무 부분이 잘려 나옵니다. 목표는 필요한 목재량 M이 확보되도록 톱날 높이 H를 설정하는 것인데, H가 낮을수록 많이 자르고 높을수록 적게 자르겠죠. 이 문제는 "H를 얼마로 해야 원하는 목재량을 얻는가"라는 **최적화 문제**입니다. 이를 "H로 필요한 목재를 얻을 수 있는가?"라는 **결정 문제**로 바꿔 생각해볼 수 있습니다. 즉, 어떤 H값에 대해 나무들을 잘랐을 때 합계가 M 이상이 되는지를 빠르게 계산하고 (모든 나무 높이에 대해 합산) **예/아니오**로 판단할 수 있다면, H를 이분 탐색으로 조정하면서 답을 찾을 수 있습니다. 이 경우, 목재량은 H가 작아질수록 (더 낮게 자를수록) 증가 모노톤이고, H가 커질수록 감소 모노톤입니다. 우리가 원하는 건 **목재량 ≥ M을 만족하는 최대 H**일 테니, 결정 함수는 "H로 목재 M 이상 얻을 수 있는가?"가 되고 이 함수의 참/거짓이 H에 대해 단조하게 변합니다 (H를 낮추면 참→거짓으로 변할 수 있고, 반대로 높이면 거짓→참으로 변할 수 있음). 따라서 범위 [0, maxHeight]에서 이분 탐색을 적용하면 답을 찾을 수 있습니다.

보다 일반적으로, 파라메트릭 서치를 쓰려면 다음 조건들이 성립해야 합니다 [5]: 1. 답이 될 수 있는 후보 값을 정했을 때 **결정 문제(Yes/No)**로 쉽게 판단할 수 있어야 합니다. (위 나무 문제에서는 주어진 H로 M 이상 얻는지를 합산하여 판단) 2. 결정 문제의 결과가 **해답 공간에서 단조성**을 띠어야 합니다. 즉 어떤 값 X에서 조건이 참이라면 X보다 더 큰 값들은 모두 참 (혹은 그 반대)과 같이, 참/거짓 구간이 연속적으로 나타나야 합니다. 그래야만 이분 탐색으로 범위를 좁힐 근거가 생깁니다. 3. 해답 공간의 범위가 명확히 정의되고 유한해야 합니다. 대부분의 경우 문제에서 주어지는 최소~최대 범위를 그대로 활용하거나, 없을 경우 논리적으로 최소 0 또는 1부터 최대 어떤 큰 값까지로 설정합니다.

파라메트릭 서치를 **시각적**으로 생각해 보면, 이분 탐색과 거의 유사합니다. 다른 점은 우리가 탐색하는 대상이 배열의 인덱스가 아니라 **값 공간**이라는 점입니다. 예를 들어, 1부터 1,000,000까지의 값 중 답이 있다고 예상되면 그 범위를 반씩 줄여가며 중간값을 시험해보는 것입니다. 각 시험(결정 문제 판단)에서 **그 값이 가능한지 확인하기 위해 별도의 알고리즘**이 들어갈 수 있으며, 흔히 그 안에 **그리디 알고리즘**이나 **그래프 탐색/Dynamic Programming** 등이 들어가기도 합니다. 하지만 외부에서는 그 판단을 일종의 블랙박스 함수로 보고 이분 탐색은 그 결과 (True/False)에 따라 범위를 움직일 뿐입니다. 파라메트릭 서치를 통해 문제를 풀면, 보통 시간 복잡도는 **결정 문제 판단 비용 × 로그(범위)**로 표현됩니다. 범위가 큰 경우 (예: 10억 등)라도 로그 규모로 탐색하므로 결정 함수가 충분히 빠르면 전체도 효율적입니다.

2.2 대표 문제 유형

이분 탐색과 파라메트릭 서치는 매우 널리 쓰이는 개념이라, 다양한 문제 유형에서 등장합니다. 대표적인 사례들을 살펴보겠습니다:

- **기본 이분 탐색 문제**: **정렬된 배열에서 값 찾기**는 가장 기본입니다. 주어진 값이 배열에 존재하는지 탐색하거나, **Lower Bound / Upper Bound** (특정 값 이상이 처음 나오는 위치, 특정 값을 초과하는 첫 위치 등)를 구하는 것도 이분 탐색으로 구현합니다. 많은 언어의 표준 라이브러리에서 `binary_search`, `lower_bound` 등의 함수로 제공하기도 합니다.
- **특정 조건을 만족하는 경계 찾기**: 정렬은 되어 있지만 값이 정확히 일치하지 않아도 되는 경우, 예컨대 "X 이상인 첫 번째 원소 찾기" 또는 "X보다 큰 첫 번째 원소 찾기", "정렬된 데이터에서 구간 [L, R]에 속하는 원소 개수 구하기" 등의 문제도 이분 탐색으로 양 끝 경계를 찾아 해결합니다.
- **파라메트릭 서치 활용 문제**: 앞서 설명한 **나무 자르기** (목표 길이 확보를 위한 톱날 높이 찾기)나 **랜선 자르기** (랜선을 일정 길이로 잘라 최대 몇 개 만드는지, 목표 개수 이상 만들 수 있는 최대 길이 찾기)가 전형적입니다 [6]. 이런 문제들은 결정 함수가 "mid 길이로 잘랐을 때 목표 개수 이상 얻을 수 있는가"가 되고, 길이가 길수록 덜 얻어지므로 결정 함수의 결과가 단조 변합니다. 또 다른 예로 **공유기 설치** 문제가 있습니다. N개의 집에 최대한 멀리 떨어지게 공유기 C개를 설치할 때, 공유기 사이 최소 거리의 최댓값을 구하는 문제인데, 이 역시 "D 거리 이상으로 C개 설치 가능한가?"를 결정 문제로 하여 D를 이분 탐색으로 찾아냅니다 (거리 D가 크면 설치하기 어려워지고, 작으면 설치하기 쉬워지니 단조 조건 성립).
- **스케줄링/배분 문제의 최적화**: 예를 들어 M개의 작업을 N명의 사람에게 분배할 때 **걸리는 최소 시간**을 구하는 문제 등이 있습니다. X 시간을 주었을 때 모든 작업을 처리할 수 있는지를 (그리디하게 작업을 할당하며) 결정하

고, 가능/불가능 여부로 이분 탐색으로 시간을 좁혀가는 식입니다. 이러한 문제는 결정 함수에 **그리디 알고리즘**이 들어가고, 이분 탐색과 결합하여 최적 해를 찾는 전형적인 패턴입니다.

- **기타: K번째 수 찾기** 같은 문제에서도 이분 탐색+결정 문제 아이디어가 쓰입니다. 예를 들어 정렬되지 않은 매우 큰 배열에서 K번째로 작은 수를 찾는 문제를, "X 이하의 수가 K개 이상 있는가?"라는 결정 문제로 바꾸어 X값 범위를 이분 탐색하는 식입니다. 또는 2D 행렬에서 K번째 작은 원소 찾기 등 다양한 응용 변형들이 있습니다.
- **실수 범위에서의 탐색**: 삼분 탐색과도 겹치는 영역이지만, 가끔 **실수 범위**에서 어떤 조건을 만족하는 최적점을 찾아야 할 때, 오차 허용 범위 내에서 **이분 탐색**을 사용하는 경우도 있습니다. 예를 들어 "어떤 함수가 0이 되는 지점"을 이분법으로 근 접근하거나, 확률/통계적인 임계값을 찾는 경우 등이 있습니다. 이 경우는 무한 또는 연속 범위지만, 일정 precision까지 반복하여 수렴시키는 방식으로 `O(log(1/ε))` 번 반복하게 됩니다.

정리하면, **이분 탐색은 정렬이나 단조성이 있는 경우에 범용적으로 쓰이고, 파라메트릭 서치는 결정 조건과 단조성만 확보되면 최적화 문제 전반에 활용**됩니다. 알고리즘 문제를 풀다 보면 "최대한 ~하게 한다" 또는 "최소/최대 ~값은?"처럼 최적화를 요구하는 문장에서 이러한 접근을 생각해볼 수 있습니다. 이후의 코드 예시에서 한 가지 사례를 직접 구현해보겠습니다.

2.3 코드 예시 (C++ / Java / Python)

이번에는 **파라메트릭 서치**의 구체적인 예시로, 흔히 볼 수 있는 **랜선 자르기 문제**를 코드로 구현해 보겠습니다. 문제 정의를 간략히 설명하면 다음과 같습니다:

> 이미 가지고 있는 여러 개의 랜선이 있고, 이들을 잘라서 일정 개수 이상의 랜선을 만들고자 한다. 만들고자 하는 랜선의 **최대 길이**를 구하라.

예컨대, 4개의 랜선 길이가 `[802, 743, 457, 539]`이고 최소 11개의 랜선을 만들고 싶다면, 랜선을 자를 수 있는 최대 길이는 200이 됩니다 (802를 4개로 자르고, 743을 3개, 457을 2개, 539를 2개로 잘라 총 11개 확보 가능). 이 문제를 해결하기 위해, **결정 문제**를 다음과 같이 설정합니다: "임의의 길이 L로 모든 랜선을 잘랐을 때, K개 이상 얻을 수 있는가?". L에 대한 이 조건은 L이 길어질수록 만들 수 있는 조각 수가 줄어들어 **단조 감소**하므로, 참/거짓 구간이 명확합니다. 이제 L의 범위를 1부터 최대 랜선 길이까지로 놓고 이분 탐색을 진행하면서, 결정 함수를 이용해 조건을 만족하는지 확인하면 됩니다. 최종적으로 조건을 만족하는 최대 L을 찾으면 그것이 답입니다.

아래 C++, Java, Python 코드에서는 위 개념을 구현하고 있습니다. 코드 구조를 보면, **이분 탐색 루프** 안에서 각 랜선을 `mid` 길이로 잘라 몇 조각 얻어지는지 합산(`count`)하고, 그 합이 목표 개수 `K` 이상인지 판별하여 `lo`나 `hi`를 조정합니다. `result` 변수에 조건을 만족한 (Yes) 경우의 `mid` 값을 기록해 두었다가 최종 답으로 사용합니다.

C++ 코드:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
  vector<int> lengths = {802, 743, 457, 539};
  int K = 11;
  long long lo = 1;
  long long hi = *max_element(lengths.begin(), lengths.end());
  long long result = 0;
  while (lo <= hi) {
    long long mid = (lo + hi) / 2;
    long long count = 0;
    for (int len : lengths) {
      count += len / mid;   // 각 랜선을 mid 길이로 잘라 몇 조각 나오는지
```

```
    }
    if (count >= K) {
      result = mid;      // mid 길이로 K개 이상 만들 수 있음 (조건 만족)
      lo = mid + 1;      // 더 큰 길이도 가능한지 오른쪽 구간 탐색
    } else {
      hi = mid - 1;      // K개 못 만들면 길이를 줄여야 함, 왼쪽 구간 탐색
    }
  }
  cout << "최대 길이: " << result << endl;
}
```

Java 코드:

```
import java.util.*;
class  Main {
  public static void main(String[] args) {
    int[] lengths = {802, 743, 457, 539};
    int K = 11;
    long lo = 1;
    long hi = 0;
    for (int len : lengths) {
      if (len > hi) hi = len;
    }
    long result = 0;
    while (lo <= hi) {
      long mid = (lo + hi) / 2;
      long count = 0;
      for (int len : lengths) {
        count += (len / mid);
      }
      if (count >= K) {
        result = mid;   // 조건 만족, 더 큰 값 시도
        lo = mid + 1;
      } else {
        hi = mid - 1;   // 조건 불만족, 값 줄임
      }
    }
    System.out.println("최대 길이: " + result);
  }
}
```

Python 코드:

```
lengths = [802, 743, 457, 539]
K = 11
lo, hi = 1, max(lengths)
result = 0
while lo <= hi:
```

```
        mid = (lo + hi) // 2
        count = 0
        for length in lengths:
            count += length // mid
        if count >= K:
            result = mid    # 조건 만족 시
            lo = mid + 1    # 더 큰 길이로 시도
        else:
            hi = mid - 1    # 불만족 시 길이 줄이기
    print("최대 길이:", result)
```

위 코드들은 파라메트릭 서치의 전형적인 구조를 보여줍니다. **이분 탐색 루프** 내부에서 `mid` 계산, 조건 체크, `lo`/`hi` 이동, 그리고 필요 시 현재 `mid` 값을 결과로 저장하는 형태입니다. `lo <= hi` 조건을 쓰는 이유는 `lo`가 `hi`를 넘어서면 탐색 구간이 유효하지 않게 되므로 종료한다는 의미이고, C++/Java에서는 `long long` 또는 `long`을 사용한 반면 Python에서는 큰 정수도 기본 지원되므로 별도 자료형 구분이 없습니다.

시각적으로 이해해보면, 처음 `mid`가 약 중간 값으로 시도되고, 그 길이로 자르면 몇 개 나오나 검사합니다. 예제의 초기 `lo=1, hi=802`라면 `mid=401`로 시작할 것입니다. 이 길이로 자르면 `802/401=2개`, `743/401=1개`, `457/401=1개`, `539/401=1개` 합쳐 5개밖에 못 만듭니다. 5 < 11이므로 조건 불만족, 따라서 `hi`를 줄여 `mid`를 더 작게 만들어야 합니다. 이런 과정을 거치며 `mid`가 점점 줄어들다가 조건을 만족하는 구간에 들어설 것입니다. 최종적으로 200을 시도하면 `802/200=4`, `743/200=3`, `457/200=2`, `539/200=2` 합 11개로 딱 만족하고, 더 높이 (`201`)로는 10개가 되어 안 되니까 200이 답이 되는 원리입니다. 이 전체 탐색은 로그(802) ≈ 9번 정도 루프 돌며 결정 함수를 계산하니, 각 루프에서 배열 길이 4를 순회하는 것은 무시할 정도로 빨라, 전체적으로 매우 빠르게 답을 찾습니다.

이처럼 **파라메트릭 서치의 시간 복잡도**는 결정 함수 계산에 걸리는 시간 `O(N)`과 이분 탐색의 `O(log M)` (M은 탐색 범위 크기) 곱으로 `O(N log M)`가 됩니다. 위 예시에선 N=4, M≈800이었지만, 일반적으로 N이 수십만, M이 수억 이상 될 수도 있으므로 이러한 효율 개선이 없으면 풀기 힘든 문제들이 많습니다.

2.4 시간 복잡도 및 적용 전략

시간 복잡도: 기본적인 이분 탐색은 `O(log N)`입니다. 여기서 N은 탐색 대상의 크기(예를 들어 배열의 길이)입니다. 1000만 개의 정렬된 배열도 약 `log2(10000000) ≈ 24` 단계면 찾을 수 있을 정도로 효율적입니다. 파라메트릭 서치는 앞서 언급한 대로 `O(f(N) * log R)` 꼴로 분석되는데, `R`은 해답 범위의 크기, `f(N)`은 한 번 결정 함수를 계산하는 데 걸리는 시간입니다. 많은 경우 `f(N)`이 선형 (`O(N)`)이므로 전체는 `O(N log R)`가 됩니다. 예를 들어, N=100만개의 랜선이 있고 R (최대 랜선 길이)이 2^31 ≈ 21억인 경우에도, $\log_2(21억) \approx 31$ 이므로 31 * 100만 ≈ 3100만 연산으로 충분히 답을 찾을 수 있습니다. 이는 N*R (≈ 10^14) 크기의 완전 탐색에 비하면 어마어마한 개선입니다.

공간 복잡도: 이분 탐색 자체는 탐색 범위의 인덱스나 값 몇 개만 저장하면 되므로 `O(1)`입니다. 파라메트릭 서치도 별도로 필요한 공간은 크지 않지만, 결정 함수를 구현하는 방식에 따라 자료구조를 쓸 수는 있습니다. 그러나 대부분은 입력 데이터를 읽고 저장하는 정도의 공간만 사용합니다.

적용 전략: 이분 탐색/파라메트릭 서치를 문제 해결에 적용하기 위해선, 다음 전략을 고려합니다: - **문제가 정렬된 입력을 제공하는가?** 만약 그렇다면 대상이 되는 배열이나 리스트에 이분 탐색 적용을 우선 떠올려볼 수 있습니다. 정렬은 많은 문제에서 암시적으로 주어지며, 이 경우 선형 탐색 대신 이분 탐색으로 시간을 단축하는 것이 기본적인 최적화입니다. - **탐색 범위를 정의할 수 있는가?** 파라메트릭 서치를 위해서는 답이 존재할 수 있는 범위를 정해야 합니다. 문제에서 명시적으로 범위를 주기도 하고 (예: ~이하의 정수, ~이하의 값 등), 그렇지 않다면 논리적으로 최소/최대치를 유추해야 합니다. 일반적으로 **최소 0 (또는 1)**부터 **최대 어떤 값**(예: 배열 내 최대값, 모든 원소 합, 등)으로 설정합니다. - **결정 문**

제로 변형 가능한가? 답을 검증하는 방법을 "예/아니오" 문제로 바꿀 수 있어야 합니다. 즉 임의의 후보 답 X에 대해 "조건을 만족하는가?"를 체크할 수 있는가를 생각해야 합니다. 이 때 해당 검사 알고리즘을 어떻게 짤지 구상합니다. 예를 들어 X 시간 내에 일을 끝낼 수 있는가 -> 작업을 그리디하게 배정해본다, X 길이로 잘라 K개 만들 수 있는가 -> 길이를 나누어본다 등. - **결정 조건의 단조성**: 결정 함수가 X값에 따라 True/False가 **한 방향으로만 변하는지** 확인해야 합니다. 대부분 직관적으로 그러하지만, 혹시라도 중간에 불규칙하게 참/거짓이 섞여 있으면 이분 탐색이 실패합니다. 예를 들어, X값이 너무 작으면 안 되고 어느 범위 이상부터 가능하다가, 너무 커지면 다시 불가능해지는 형태라면 단조성이 없습니다 (다행히 이런 경우는 드뭅니다). - **오차 범위 (실수 탐색의 경우)**: 만약 연속적인 값(실수 범위)을 탐색한다면 언제 멈출지 결정해야 합니다. 문제에서 소수점 몇째 자리까지 요구한다면 그에 맞춰 반복을 제한하거나, 원하는 정확도가 나올 때까지 루프를 돌립니다. 실수 이분 탐색은 수렴 과정에서 무한 루프를 방지하도록 적절한 종료 조건을 넣어야 합니다.

구현 팁: 이분 탐색을 구현할 때 흔히 발생하는 실수는 **무한 루프**나 **off-by-one 오류**입니다. 이를 피하기 위해 일반적으로 `lo <= hi`를 조건으로 쓰고, `mid = (lo + hi) / 2`로 계산한 후 `lo` 또는 `hi`를 `mid` ± 1로 옮기는 패턴을 사용합니다 (위 코드 예시도 동일). 또한 `mid` 계산 시 자료형 범overflow에 주의해야 합니다. C++에서 `(lo + hi) / 2`는 lo, hi가 int면 int 오버플로우 위험이 있어, 64비트로 변환하거나 `lo + (hi - lo) / 2` 형태를 쓰기도 합니다. 파이썬은 큰 정수 지원으로 상관없지만, 부동소수점 이분 탐색 시에는 `(lo + hi) / 2.0`이 적절합니다. 마지막으로, **결과값이 최댓값/최솟값 경계일 때** 제대로 잡히는지도 생각해야 합니다. 예를 들어 위 코드에선 `result`를 조건 만족 시 갱신했는데, 끝날 때 `lo`와 `hi`가 교차되며 루프가 종료되므로 `hi` (또는 `result`)가 정답이 됩니다. 구현 방식에 따라 `lo`가 답+1로 끝나기도 하고, `hi`가 답으로 끝나기도 하므로, 루프 이후 어느 변수를 출력할지 혼동하지 않도록 해야 합니다.

---

3. 기타 알고리즘 개념

메인 개념에서 다룬 투 포인터/슬라이딩 윈도우, 이분 탐색/파라메트릭 서치 외에도, 이번 주 주제와 연관하여 알아두면 좋은 몇 가지 알고리즘 기법들이 있습니다. 여기에선 **간략히 개념**을 설명하거나 언급하는 정도로 짚고 넘어가겠습니다.

3.1 Meet in the Middle (중간에서 만나기)

Meet-in-the-Middle은 흔히 **브루트포스(완전 탐색)** 방법을 최적화할 때 사용되는 테크닉입니다. 해결해야 할 문제의 입력 크기가 커서 순전히 완전 탐색하면 `O(2^N)`과 같이 너무 느릴 때, 입력을 **반으로 나눠서 각각 완전 탐색을 수행**한 뒤 그 결과를 합치는 방식으로 복잡도를 줄입니다 [7]. 말 그대로 "중간에서 만난다"는 의미로, 절반씩 구한 부분해들을 조합하여 원래 문제의 해답을 찾아내는 방법입니다.

예를 들어, N개의 원소로 이루어진 집합에서 부분집합의 합으로 특정 S 값을 만들 수 있는지 확인하는 문제를 생각해봅시다. 완전 탐색을 하면 가능한 부분집합이 `2^N`개라 N이 40만 되어도 `2^40 ≈ 1조`로 불가능합니다. Meet-in-the-Middle 기법을 쓰면, 첫 절반 N/2 = 20개의 부분집합 합 모두와 나머지 절반 20개의 부분집합 합 모두를 구합니다. 첫 절반에서 나올 수 있는 합이 예컨대 X라고 하면, 두 번째 절반에서 S-X 값을 만들 수 있는지를 이분 탐색이나 투 포인터로 찾는 식으로 전체 해를 구성합니다. 이렇게 하면 첫 절반 완전 탐색 `O(2^(N/2))`, 두 번째 절반 `O(2^(N/2))`, 그리고 정렬 및 탐색 비용 몇 배가 추가되지만, 총합으로는 대략 `O(2^(N/2) * 2)` 수준으로 떨어집니다. N=40이면 `2^20 ≈ 100만` 정도로 현실적인 숫자가 됩니다. 실제로 유명한 백준 1208 부분수열의 합 2 문제 등이 이 기법으로 해결 가능합니다.

Meet-in-the-Middle은 **분할 정복**과 유사해 보이지만, 조금 다르게 **부분 결과를 생성 후 합치는 추가 과정**이 있는 것이 특징입니다 [7]. 이때 합치는 과정에서 이분 탐색이나 투 포인터를 함께 사용하여 원하는 결과를 효율적으로 찾습니다 (예: 두 리스트에서 하나씩 선택하여 합이 S가 되는 경우 찾기). 보통 N이 30~40 정도로 애매하게 큰 완전 탐색 문제에서 자주 등장하는 아이디어이므로, 이러한 크기의 문제가 보이면 "절반으로 쪼개 볼까?"를 고려해볼 만 합니다.

3.2 삼분 탐색 (Ternary Search)

삼분 탐색(ternary search)은 **이분 탐색과 유사**하지만, 탐색 범위를 한 번에 **3등분**하여 범위를 줄여나가는 탐색 방법입니다. 이는 배열이 아닌 **함수의 극값(최대나 최소)을 찾는 문제**에 주로 사용됩니다. 특히 함수가 **unimodal (단봉)** 특성을 가질 때, 즉 하나의 봉우리나 골만 존재할 때 적용할 수 있습니다 [8]. 예를 들어, 어떤 함수 f(x)가 x=a에서 최대값을 갖고 양쪽으로는 감소한다고 합시다. 이 범위에서 최대값을 찾기 위해 삼분 탐색을 쓰면, 매번 두 개의 중간점 (1/3 지점, 2/3 지점)을 선택하여 f값을 비교합니다. f(1/3)과 f(2/3)을 비교하면, 어느 쪽이 더 큰지에 따라 봉우리가 왼쪽 구간에 있는지 오른쪽 구간에 있는지 알 수 있고, 그 아닌 쪽 구간을 버립니다. 이렇게 하면 매 단계마다 남는 범위가 기존의 2/3로 줄어들므로, **시간 복잡도는 이분 탐색과 마찬가지로** `O(log N)` (밑이 1.5 정도지만 상수배 차이)입니다 [8].

삼분 탐색은 주로 **함수 최적화 문제**에서 사용됩니다. 예를 들어, "어떤 포인트에서의 값이 최소인지를 찾아라" 같은 문제는 이분 탐색으로 풀긴 어렵고 삼분 탐색이 적합합니다. 실전 문제로는 **백준 8986 전봇대** 문제가 삼분 탐색을 활용하는 예로 알려져 있습니다. 다만, 삼분 탐색이 적용되려면 함수가 **한 번 감소하고 그 후 증가 (최소값의 경우)** 혹은 **한번 증가 후 감소 (최대값의 경우)** 형태여야 합니다. 여러 번 증가/감소가 반복되는 함수 (멀티모달)에는 사용할 수 없습니다. 또한 이론적으로는 삼분 탐색이 이분 탐색과 같은 복잡도지만, 한 번에 두 번 계산해야 하므로 상수 시간은 조금 더 들 수 있습니다. 따라서 **정수 범위**에서의 단조한 결정 문제는 굳이 삼분 탐색을 쓰기보다 이분 탐색/파라메트릭 서치를 쓰는 편입니다. 삼분 탐색은 주로 **실수 범위**에서의 함수 최적값 찾기나, **정수 범위이지만 오직 극값 찾기 목적일 때** 활용됩니다.

3.3 슬라이딩 윈도우 + 자료구조 활용

슬라이딩 윈도우 기법을 기본적으로는 투 포인터로 구현하지만, 때로는 **윈도우 내부의 정보를 효율적으로 관리**하기 위해 보조 자료구조를 함께 사용합니다. 윈도우가 이동하면서 제외되는 원소와 새로 포함되는 원소가 생기는데, 단순한 합이나 개수는 앞서처럼 변수로 관리하면 충분합니다. 하지만 다음과 같은 경우 자료구조가 필요해집니다: - **윈도우 내 최댓값/최솟값**을 매번 구해야 하는 경우: 윈도우 이동 시 값이 빠지고 들어오므로, 정렬 상태가 변할 수 있습니다. 이때 **데크(Deque)**를 이용하면 상수 시간에 윈도우 최댓값을 업데이트 할 수 있는 테크닉이 있습니다 (이 알고리즘은 별도로 "슬라이딩 윈도우 최댓값 문제"로 유명합니다). - **윈도우 내 원소의 빈도나 종류**를 추적해야 하는 경우: 예를 들어 문자열에서 "윈도우 내 서로 다른 문자 개수", "윈도우 내 각 문자 빈도" 등 조건을 관리하려면, **해시맵(HashMap)**이나 **카운터(counter)**를 사용합니다. 윈도우가 이동할 때 빠지는 문자 빈도 감소, 들어오는 문자 빈도 증가 처리를 해서 현재 윈도우 상태를 파악합니다. (예: 아나그램 찾기 문제에서 고정 길이 윈도우의 문자 빈도를 비교하거나, 앞서 언급한 중복 없는 부분 문자열 문제에서 윈도우 내 문자 존재 여부를 체크할 때 등) - **복합 자료구조 활용**: 윈도우 내에서 빠르게 중간값을 찾거나 특정 순위의 값을 찾아야 하는 경우에는 **Balanced BST(균형 이진 탐색 트리)**나 **힙(heap)**을 윈도우와 함께 사용하기도 합니다. 예를 들어, 정렬되지 않은 스트림에서 크기 k 윈도우별 중간값을 출력하는 문제는 두 개의 힙을 이용해 윈도우를 구성하는 유명한 해결법이 있습니다.

요컨대, 슬라이딩 윈도우는 **투 포인터로 기본 구현**하고, 그 안에서 필요한 부가 정보를 유지하기 위해 자료구조를 접목하는 것입니다. 이것은 결국 각 연산을 평균적으로 효율적으로 만드는 데 목적이 있습니다. 예를 들어, 윈도우 내 최댓값을 구하려고 매번 배열 구간을 선형 탐색하면 `O(N*k)`가 되지만, 데크를 쓰면 전체가 `O(N)`에 해결됩니다. 윈도우 내 빈도 수 업데이트도 맵을 쓰면 각 문자 당 `O(1)`에 빈도 변경이 가능하므로 전체가 `O(N)`이 됩니다. 이러한 응용은 투 포인터/슬라이딩 윈도우 기법의 활용 범위를 크게 넓혀주며, 다양한 **문자열 알고리즘**이나 **실시간 데이터 처리**에서 쓰입니다.

3.4 이분 탐색 + 그리디 결합

이 부분은 사실 앞서 설명한 **파라메트릭 서치**의 구현 양상과 많이 겹칩니다. 이분 탐색 + 그리디란, **결정 문제를 판단하는 방법으로 그리디 알고리즘을 사용하는 경우**를 뜻합니다. 많은 최적화 문제에서 "어떤 값 X로 가능한가?"를 검사할 때, 자연스럽게 떠오르는 방법이 **욕심쟁이 전략(그리디)**인 경우가 있습니다.

예를 들어, 다시 **공유기 설치** 문제를 생각해보겠습니다. 거리가 D 이상 떨어지게 공유기를 설치할 수 있는지를 검사하려면, 그리디하게 첫 집부터 차례로 **가능한 가장 이른 위치**에 공유기를 놓아보는 것이 합리적입니다. 첫 집에 하나 놓고,

그 다음에는 이전에 놓은 위치로부터 D 이상의 간격을 두고 가장 빠른 집에 놓고... 이런 식으로 진행해서 C개를 모두 설치해보는 것이죠. 이 방법은 탐욕적이지만, 공유기 설치 문제에서는 최적해를 검증하는 데 정확한 방법입니다. 그 이유는, 설치 간격을 벌리는 방향으로 일부러 늦게 설치할 이유가 없기 때문입니다. 이처럼 **"가능하면 최대한 앞으로 당겨서"** 하는 결정이 최적성을 해치지 않는 상황에서 그리디가 옳은 판단입니다. 결정 문제 판단에 그리디를 쓰면 구현도 간단하고 빠르며, 그 결과는 참/거짓으로 떨어집니다. 이후 이분 탐색으로 D를 조정하면 최적의 D를 찾게 됩니다.

또 다른 예시로, 작업 스케줄링 문제에서 X시간 안에 가능하냐를 볼 때, **가장 수행 시간이 긴 작업부터** 할당한다거나 **여유가 적은 기계부터** 채워나간다거나 하는 그리디 전략을 써볼 수 있습니다. 물론, 모든 경우에 그리디가 통하는 것은 아니어서, 만약 그리디가 잘못된 판단을 하게 되는 상황이면 결정 함수가 틀려질 수 있습니다. 그래서 실제로 "이분 탐색 + 그리디" 패턴이 쓰인다는 것은 해당 문제 자체가 **그리디로 결정 함수를 풀 수 있는 구조**임을 의미합니다. 이러한 문제들은 일반적으로 그리디 알고리즘으로도 풀리곤 하지만, 이분 탐색을 조합하면 탐색해야 할 범위를 줄여서 더 쉽게 답을 탐색하는 형태가 됩니다.

정리하면, **이분 탐색 + 그리디 = 파라메트릭 서치**의 한 형태입니다. 결정 문제의 판별에 그리디를 활용하는 것이지요. 실제 코딩 테스트나 알고리즘 문제에서도 이 조합은 흔히 등장하며, 잘 알려진 문제들 (예: 공유기 설치, 배낭 용량 결정 문제 등)에서 자주 보입니다. 이 접근을 위해서는 먼저 "이 값을 주었을 때 어떻게 배치/선택하면 가장 잘 해볼 수 있을까?"를 고민해보고, 그 방법이 최선이라는 확신이 들면 그리디를 써도 좋습니다. 그리고 나서 이분 탐색을 씌우면 전체 문제를 깔끔하게 해결할 수 있습니다.

---

4. 결론 및 다음 주 미리보기

이번 주는 **투 포인터/슬라이딩 윈도우**를 통해 배열과 구간을 효율적으로 탐색하는 법을, **이분 탐색/파라메트릭 서치**를 통해 정렬된 데이터 및 결정 문제를 빠르게 해결하는 법을 배우며, 알고리즘 문제 해결을 위한 중요한 **도구들(toolset)**을 확보했습니다. 지난 주의 정렬 알고리즘과 결합하여 생각해보면, 정렬로 데이터를 준비하고 이분 탐색으로 빠르게 찾거나, 정렬된 데이터에서 투 포인터로 원하는 쌍을 찾는 등, 실제 많은 문제들이 이러한 알고리즘들의 **조합으로 해결**될 수 있음을 알 수 있습니다. 더 나아가, Meet-in-the-Middle이나 삼분 탐색 같은 기법으로 완전 탐색이나 이분 탐색의 변형도 살펴보았고, 자료구조와의 융합 또는 그리디와의 융합으로 알고리즘을 한 단계 업그레이드하는 전략도 확인했습니다. 이처럼 알고리즘 기법들은 개별로도 중요하지만 **상황에 맞게 적절히 선택하고 조합하는 역량**이 문제 해결의 열쇠임을 알 수 있습니다.

다음 주에는 **동적 계획법(Dynamic Programming)**을 다룰 예정입니다. 동적 계획법은 **복잡한 문제를 작은 부분 문제로 나누어 푸는 전략**으로, 그동안 탐색이나 정렬, 그리디로 풀 수 없었던 다양한 문제들을 해결하는데 핵심이 되는 기법입니다. 특히 **메모이제이션**과 **상태 정의** 등을 통해 **중복 계산을 피하고 최적 부분 구조**를 활용하는 방법을 배울 것입니다. 이번 주까지 다룬 내용이 주로 **그리디하게 혹은 순차적으로 결정**하는 유형이었다면, 다음 주에는 **모든 가능성을 체계적으로 고려**하면서도 효율성을 놓치지 않는 테크닉을 연마하게 됩니다. 동적 계획법을 배우면 이전에 다룬 탐색이나 정렬 같은 개념들과 합쳐 훨씬 더 많은 문제들을 풀 수 있게 되니, 기대하는 마음으로 준비해 주시기 바랍니다!

끝으로, 이번 주 학습한 내용을 꼭 복습하고, 제공된 예제 코드들을 직접 실행/변형해 보면서 확실히 이해하시길 권장합니다. 알고리즘은 **직접 손으로 구현해보고 다양한 입력에 테스트**해볼 때 비로소 내 것이 됩니다. 궁금한 점이나 어려운 점은 서로 질의응답하며 해결해 나가고, 다음 주 동적 계획법 강의에서 다시 만나겠습니다. 수고하셨습니다!

---

1 2 3 [알고리즘] Two Pointers (투 포인터) / Sliding Window (슬라이딩 윈도우) :: 개발자는 아닙니다
https://void2017.tistory.com/333

4 5 6 [이분탐색] 파라메트릭 서치(개념)
https://sarah950716.tistory.com/16

7 Meet in the middle (MITM)
https://thinkmath2020.tistory.com/3737

8 [Algorithm/C++] 삼분 탐색(Ternary search) (백준 8986 전봇대)
https://movingbean.tistory.com/9