시간 복잡도

- 알고리즘이 문제를 해결하는 데 걸리는 시간을 입력 크기(n)에 대한 함수로 표현한 것
- 입력 데이터의 크기가 커질수록 알고리즘이 얼마나 빨리 또는 느리게 실행되는지를 나타내는 척도

코딩 테스트 경향

- 알고리즘에 대한 딥한 내용보다, 시간 복잡도를 고려한 설계를 묻는 문제들이 많아짐
- 구현 + 시간 복잡도

시간 복잡도 종류

시간 복잡도의 종류 (예시 포함)

1. `O(1) – 상수 시간`
    - 입력 크기와 상관없이 항상 일정한 시간이 걸림.
    - 예: 배열의 첫 번째 요소 접근.

2. `O(log n) – 로그 시간`
    - 입력 크기가 커질수록 연산 횟수가 로그 함수로 증가. 주로 이진 탐색.
    - 예: 정렬된 배열에서 이진 탐색.
    - n = 1,000,000 → 약 20번 연산.

3. `O(n) – 선형 시간`
    - 입력 크기에 비례해 연산 횟수가 증가.
    - 예: 배열의 모든 요소를 한 번씩 확인.

4. `O(n log n) – 선형 로그 시간`
    - 효율적인 정렬 알고리즘(퀵 정렬, 병합 정렬 등)에서 자주 등장.
    - 예: n개의 요소를 log n번씩 처리.

5. `O(n²) – 제곱 시간`
    - 입력 크기의 제곱에 비례. 중첩 루프에서 흔함.
    - 예: 버블 정렬.

6. $O(2^n)$ - 지수 시간
    - 입력 크기가 조금만 커져도 실행 시간이 급격히 증가.
    - 예: 재귀적 부분 집합 생성.

시간 복잡도별 한계

> 시간 복잡도에 따른 알고리즘 고려 참고(자바 기준ㅍ)

- O(n)

  : n = 10^7 ~ 10^8일 때 1초 내 가능.

  - 예: 배열 순회 1억 번 → 약 1초.

- O(n log n)

  : n = 10^5 ~ 10^6일 때 가능.

  - log n ≈ 20이라 가정 → 10^6 × 20 = 2×10^7번 → 0.2~0.5초.

- O(n^2)

  : n = 10^4일 때 가능.

  - 10^4 × 10^4 = 10^8번 → 1초 근처.
  - n = 10^5라면 10^{10}번 → 100초로 시간 초과.

시간 복잡도 최적화를 위한 알고리즘

1. `구간합`
2. `이진탐색`
3. `DP`
4. `투 포인터`
5. `슬라이딩 윈도우`

1. 구간합

사용 사례

- 문제: 배열에서 여러 번 구간 합을 구해야 할 때.
- 예: "배열 [1, 2, 3, 4, 5]에서 인덱스 1~3의 합은?"
  - 단순 계산: O(n) → 2 + 3 + 4 = 9.
  - 구간 합: 누적 배열 [1, 3, 6, 10, 15] 생성 → sum[3] - sum[0] = 10 - 1 = 9 (O(1)).

특징

- 두 가지 작업이 필요
  - 전처리 (누적 배열 생성)
  - 쿼리 (필요한 값 추출)
- 배열에서 특정 구간의 합을 빠르게 계산하기 위해 **미리 누적된 합**을 저장.

- 전처리(preprocessing)를 통해 이후 쿼리를 상수 시간(O(1))에 해결.
- 시간 복잡도:
    - 전처리: O(n)
    - 쿼리당: O(1)

```
int[] arr = {1, 2, 3, 4, 5};
int[] prefixSum = new int[arr.length + 1];
for (int i = 0; i < arr.length; i++) {
    prefixSum[i + 1] = prefixSum[i] + arr[i];
}
// 구간 합: prefixSum[j] - prefixSum[i-1]
int rangeSum = prefixSum[4] - prefixSum[1]; // 2~4 합 = 15 - 1 = 14
```

2. 이진 탐색

사용 사례

- 검색 문제, 최적화 문제(파라메트릭 서치).
- 예: "정렬된 배열 [1, 3, 5, 7, 9]에서 7 찾기" → O(log n)으로 위치 반환
- 중간 값을 찾음 -> 5
- 중간 값(5)이 찾는 값(7)보다 작기 때문에 중간 값 기준 우측만 탐색 [7, 9]
- 7, 9의 중간 값 -> 여기서는 7 -> 찾음

특징

- **정렬된 데이터**에서 특정 값을 빠르게 찾거나 조건을 만족하는 경계를 탐색.
- 시간 복잡도: O(log n).
- 범위를 절반씩 줄이며 탐색.

```
int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2; // 오버플로우 방지
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1; // 못 찾음
}
```

파라메트릭 탐색

> 이진 탐색을 응용한 알고리즘. 최적화 문제를 결정 문제로 변환해 해결하는 방법

- 조건을 만족하는 최대값 또는 최소값을 찾을 때 사용
- 특정 값을 찾는 대신 특정 조건의 경계를 찾아냄
- O(n log n)

문제. 입국 심사 문제 (프로그래머스)

제한사항

- 입국심사를 기다리는 사람은 1명 이상 1,000,000,000명 이하입니다.
- 각 심사관이 한 명을 심사하는데 걸리는 시간은 1분 이상 1,000,000,000분 이하입니다.
- 심사관은 1명 이상 100,000명 이하입니다.

입출력 예

n	times	return
6	[7, 10]	28

입출력 예 설명

가장 첫 두 사람은 바로 심사를 받으러 갑니다.

7분이 되었을 때, 첫 번째 심사대가 비고 3번째 사람이 심사를 받습니다.

10분이 되었을 때, 두 번째 심사대가 비고 4번째 사람이 심사를 받습니다.

14분이 되었을 때, 첫 번째 심사대가 비고 5번째 사람이 심사를 받습니다.

20분이 되었을 때, 두 번째 심사대가 비지만 6번째 사람이 그곳에서 심사를 받지 않고 1분을 더 기다린 후에 첫 번째 심사대에서 심사를 받으면 28분에 모든 사람의 심사가 끝납니다.

문제 개요

- n: 심사받아야 할 사람 수.
- times: 각 심사관이 한 사람을 심사하는 데 걸리는 시간 배열.
- 목표: 모든 사람(n명)을 심사하는 데 걸리는 **최소 시간**을 찾기.

풀이

- 이진 탐색 적용
- left = 0
- right = times.max * n (가장 느린 심사관이 모든 사람을 처리하는 경우) = 60
- mid = 30

- 30시간 안에 6명을 심사할 수 있는지 확인.
    - 6명이 불가능하면 시간을 늘림(우측 탐색).
    - 6명이 가능하면 시간을 점점 줄임(좌측 탐색)

파라메트릭 서치 적용

- **탐색 대상**: 모든 사람을 심사할 수 있는 최소 시간.
- **결정 문제**: "시간 mid 안에 n명 이상을 심사할 수 있는가?"

고민 사항

- 파라메트릭 서치(이진 탐색) 후 값을 반환할 때 탐색한 값을 mid를 통해 result를 업데이트 해서 값을 출력하는 경우가 많음
    - 대부분의 풀이를 보면 result라는 변수를 두고 mid를 통해 업데이트

```
int result = 0;
while (left <= right) {
  // 로직
  if (people >= n) {
    right = mid - 1;
  } else {
    left = mid + 1;
    result = mid;
  }
}

return result;
```

- 이진 탐색의 left, right, mid가 업데이트되는 과정과 단계를 정확하게 이해하면 mid를 통한 result 업데이트 단계는 대부분 필요하지 않음
- 위 예시에서 right는 조건을 만족하지 않는(people < n) 최대값
- 위 위예시에서 left는 조건을 만족하는(people >= n) 최소값

`people` : mid 시간에 검사 받을 수 있는 사람의 수

- people이 주어진 n 보다 크다
    - 검사 시간을 더 줄여도 됨
    - 이진 탐색에서 mid 기준 왼쪽 배열만 탐색
- people이 주어진 n보다 작다
    - 검사 시간을 더 늘려야함
    - 이진 탐색에서 mid 기준 오른쪽 배열만 탐색
- 이때 종료 조건은 left <= right. 즉 종료 시점에 left는 right보다 큰 값으로 종료됨

접근

- [1, 2, 3, 4, 5, 6] 의 배열이 존재할 때 2가 최적의 시간이라고 가정해보자
    - 2 : n만큼의 수의 사람을 검사하는데 필요한 최소 시간
    - left = 1
    - right = 6
    - mid = 3
        - 3시간은 n명의 사람을 검사할 수 있기에 좌측 배열을 탐색해야함
    - left = 1
    - right = 2
    - mid = 1
        - 1시간은 n명의 사람을 검사할 수 없기에 우측 배열을 탐색해야함
    - left = 2
    - right = 2
    - mid = 2
        - 2시간은 n명의 사람을 검사할 수 있기에 좌측 배열을 탐색
    - left = 2
    - right = 1
        - right < left 이기에 탐색 종료
- 최종 결과
    - left = 2
    - right = 1
    - left는 조건을 만족하는 가장 작은 수
    - right는 조건을 만족하지 않는 가장 큰 수
    - 따라서 left를 반환하면 정답

```
while (left <= right) {
    long mid = (left + right) / 2;
    long people = 0;
    for (int time : times) {
        people += mid / time; // mid 시간 안에 각 심사관이 처리 가능한 사람 수
    }
    if (people >= n) { // n명 이상 처리 가능
        right = mid - 1; // 더 짧은 시간 탐색
    } else { // n명 처리 불가능
        left = mid + 1; // 더 긴 시간 탐색
    }
}
return left;
```

3. dp

사용 사례

- 최적화 문제, 점화식으로 표현 가능한 문제.
- 예: 피보나치 수열.
    - 단순 재귀: $O(2^n)$.
    - DP: $O(n)$.

특징

- 중복 계산을 피하기 위해 **부분 문제의 결과**를 저장하고 재활용.
- 시간 복잡도: 문제에 따라 다름 ($O(n)$, $O(n^2)$ 등).
- **메모이제이션** 또는 **테이블 작성** 방식 사용.

1. 상향식

- 작은 문제부터 시작해 점차 큰 문제로 나아가는 방식
- 반복문을 사용하여, 배열이나 테이블을 채워나감

```
public class FibonacciBottomUp {
  public static long fib(int n) {
    if (n <= 1) return n;

    long[] dp = new long[n + 1];
    dp[0] = 0; // 기저 상태
    dp[1] = 1; // 기저 상태

    for (int i = 2; i <=n; i++) {
      dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
  }
}
```

2. 하향식

- 큰 문제에서 시작해 필요한 작은 문제로 내려가는 방식
- 재귀와 메모이제이션 사용

```java
public class FibonacciTopDown {
  static long[] memo;

  public static long ifb(int n) {
    if (n <= 1) return n;
    if (memo[n] != 0) return memo[n];

    memo[n] = fib(n - 1) + fib(n - 2);
    return memo[n]
  }
}
```

비교

상향식(Bottom-Up)이 더 적합한 경우

- **피보나치처럼 단순하고 연속적인 문제**: F(0)부터 F(n)까지 모두 필요하거나, 계산 순서가 명확할 때.
- **성능 최적화 필요**: 재귀 오버헤드를 피하고, 상수 시간을 줄이고 싶을 때.
- **n이 매우 큼**: 스택 오버플로우 방지.
- **공간 효율성**: 변수만으로 해결 가능할 때.

하향식(Top-Down)이 더 적합한 경우

- 복잡한 문제 구조

  : 하위 문제가 불연속적이거나, 조건에 따라 계산 여부가 달라질 때.

  - 예: "최소 비용 경로"에서 특정 경로만 탐색.

    ```
    격자 크기: 3x3
    비용 배열:
    [1, 2, 3]
    [4, 5, 6]
    [2, 3, 1]
    조건: 비용이 5 이상인 칸은 이동 불가
    ```

- **탐색 기반 문제**: 모든 값을 미리 계산하기보다 필요한 값만 "탐색"하며 구할 때.
- **코드 작성 편의**: 점화식이 재귀적으로 표현하기 쉬울

4. 투 포인터

사용 사례: "합이 S인 연속 부분 배열 찾기"

- 배열: [1, 2, 3, 4, 5], 목표 합: S = 7.
- 목표: 합이 7인 연속 구간 찾기.

- left = 0, right = 0
- left ~ right 까지 합을 구함
    - 찾는 값보다 합이 작으면 right를 우측으로 이동
    - 찾는 값보다 합이 크면 left를 좌측으로 이동

```java
public static void findSubarray(int[] arr, int S) {
    int left = 0, sum = 0;
    for (int right = 0; right < arr.length; right++) {
        sum += arr[right]; // 오른쪽 포인터 확장
        while (sum > S && left <= right) { // 조건 초과 시 왼쪽 포인터 이동
            sum -= arr[left];
            left++;
        }
        if (sum == S) {
            System.out.println("구간: " + left + " ~ " + right);
        }
    }
}
```

개념

- 두 개의 포인터(인덱스)를 사용해 배열이나 리스트를 탐색.
- 보통 **두 포인터가 서로 다른 방향에서 시작**하거나, **조건에 따라 이동하며 구간을 조정**.
- 주로 "특정 조건을 만족하는 구간"을 찾는 데 사용.
- O(n) 시간에 배열을 탐색.

5. 슬라이딩 윈도우

사용 사례 : "길이 k인 구간의 최대 합 구하기"

- 배열: [1, 2, 3, 4, 5], 창 크기: k = 3.
- 목표: 길이가 3인 연속 구간의 최대 합 찾기.

```java
public static int maxSum(int[] arr, int k) {
    if (arr.length < k) return -1;

    int maxSum = 0;
    // 초기 창 설정
    for (int i = 0; i < k; i++) {
        maxSum += arr[i];
    }

    int windowSum = maxSum;
```

```
    // 창 슬라이딩
    for (int i = k; i < arr.length; i++) {
        windowSum = windowSum + arr[i] - arr[i - k]; // 새 값 추가, 첫 값 제거
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}
```

개념

- **고정된 크기의 창(window)**을 배열 위에서 슬라이딩하며 탐색.
- 창의 크기가 고정되어 있고, 창을 한 칸씩 이동시키며 조건을 확인.
- 주로 "고정 길이 구간"의 최대/최소 값을 찾는 데 사용.
- O(n) 시간에 배열을 탐색.

문제

1. [1, 3, 7, 8, 9]에서 '7 이상인 첫 위치를 찾을 때 이진 탐색과 파라메트릭 탐색 중 어느 알고리즘이 효율(Big O 기준 시간 복잡도만 따짐)이 더 좋을까?
    - 이진 탐색 : 주어진 배열에서 값을 찾는 것
    - 파라메트릭 탐색 : 주어진 배열의 값을 통해 특정 조건을 만족시키는 값을 찾는 것
        - 결정 함수 -> if (arr[mid -1] < arr[mid] && arr[mid-1] < 7);

2. [1, 2, 3, 4, 5]에서 길이 3인 모든 구간 합을 구하라고 하면, 누적합과 슬라이딩 윈도우 중 어느 것이 알고리즘 효율(Big O 기준 시간 복자도만 따짐)이 더 좋을까?