05-10

코딩 테스트 팁 - 시간 복잡도

연산

- 알고리즘에서 시간 복잡도는 주어진 문제를 해결하기 위한 연산 횟수.
  - 일반적으로 **수행 시간은 1억 번의 연산을 1초의 시간으로 간주하여 예측**

시간 복잡도 정의하기

- 시간 복잡도 유형
  - 빅-오메가 `Ω` : 최선일 때의 연산 횟수를 나타낸 표기법
  - 빅-세타 `θ`: 보통일 때의 연산 횟수를 나타낸 표기법
  - 빅-오 `O`: 최악일 때의 연산 횟수를 나타낸 표기법

- 코딩 테스트에서는 빅-오 표기법을 기준으로 수행 시간을 계산

어떤 알고리즘을 사용할까?

- 예시 문제) https://www.acmicpc.net/problem/2750
- 백준 2750 문제 “수 정렬하기” (시간 제한 2초)
  - 여기서는 $1 \le N \le 1,000,000$ & 절댓값: 1,000,000 이라고 가정한다. (문제에서는 1,000 으로 표기)

문제

N개의 수가 주어졌을 때, 이를 오름차순으로 정렬하는 프로그램을 작성하시오.

입력

첫째 줄에 수의 개수 N(1 ≤ N ≤ 1,000)이 주어진다. 둘째 줄부터 N개의 줄에는 수 주어진다. 이 수는 절댓값이 1,000보다 작거나 같은 정수이다. 수는 중복되지 않는다.

출력

첫째 줄부터 N개의 줄에 오름차순으로 정렬한 결과를 한 줄에 하나씩 출력한다.

예제 입력 1 복사

```
5
5
2
3
4
1
```

예제 출력 1 복사

```
1
2
3
4
5
```

- 시간 제한이 2초 → 조건을 만족하기 위해 2억 번 이하의 연산 횟수로 문제를 해결해야 함
  - 연산 횟수는 1초에 1억 번의 연산 기준.
  - 시간 복잡도는 항상 최악일 때, 즉 데이터의 크기가 가장 클 때가 기준.
- 연산 횟수 계산 방법
  - **연산 횟수 = 알고리즘 시간 복잡도 X 데이터의 크기**
- 알고리즘 적합성 평가(버블 정렬, 병합 정렬 예시)
  - 버블 정렬 = (1,000,000)^2 = 1,000,000,000,000 > 200,000,000 → 부적합
  - 병합 정렬 = 1,000,000log(1,000,000) = 약 2,000,000,000 < 200,000,000 → 적합

시간 복잡도 도출 기준

- 상수는 시간 복잡도 계산에서 제외.
  - 연산 횟수가 N인 경우

```
class Example1 {
    public static void main(String[] args) {
        int N = 10_000;
        int cnt = 0;
        for(int i = 0; i < N; i++) {
            cnt++;
        }
        System.out.println("연산 횟수: " + cnt); // 연산 횟수: 10000
```

```
    }
}
```

- 연산 횟수가 3N인 경우

```
class Example2 {
    public static void main(String[] args) {
        int N = 10_000;
        int cnt = 0;
        for(int i = 0; i < N; i++) {
            cnt++;
        }
        for(int i = 0; i < N; i++) {
            cnt++;
        }
        for(int i = 0; i < N; i++) {
            cnt++;
        }
        System.out.println("연산 횟수: " + cnt); // 연산 횟수: 30000
    }
}
```

- 두 예제 코드의 연산 횟수의 차이 → 3배. 하짐나 코팅 테스트에서는 일반적으로 상수를 무시하므로 두 코드 모두 시간 복잡도는 O(n)

- 가장 많이 중첩된 반복문의 수행 횟수가 시간 복잡도의 기준.
  - 연산 횟수가 N^2인 경우

```
class Example3 {
    public static void main(String[] args) {
        int N = 10_000;
        int cnt = 0;

        for(int i = 0; i < N; i++) {
            for(int j = 0; j < N; j++) {
                cnt++;
            }
        }
        System.out.println("연산 횟수: " + cnt); // 연산 횟수: 100000000
    }
}
```

- 일반 for 문이 10개 있다 하더라도 중첩된 반복문을 기준으로 시간 복잡도가 도출

배열 & 리스트

배열

- 메모리의 연속 공간에 값이 채워져 있는 형태의 자료구조.
- 인덱스를 통해 배열의 값을 참조할 수 있음.
- 선언한 자료형의 값만 저장 가능.
- 특징 정리
    - 인덱스를 사용하여 값에 바로 접근.
    - 특정 값을 삽입하거나 삭제하기 어려움.(주변에 있는 값을 이동하는 과정 필요)
    - 구조가 간단하여 코딩 테스트에서 많이 사용.
    - 자바 → 선언할 때 배열의 크기가 정해져 늘리거나 줄일 수 없음.
    - 자바스크립트 → 배열의 크기가 유연. (엄밀한 의미의 배열이 아니고 흉내)

리스트

- 값과 포인터를 묶은 노드라는 것을 포인터로 연결한 자료구조.
    - 노드 → 컴퓨터 과학에서 값, 포인터를 쌍으로 갖는 기초 단위를 부르는 말.
- 특징 정리
    - 인덱스가 없어 값에 접근하려면 Head 포인터부터 순서대로 접근(접근 속도 느림)
    - 포인터로 연결되어 있어 값 삽입, 삭제 속도 빠름.
    - 크기가 정해져 있지 않아 크기가 변하기 쉬운 데이터를 다룰 때 적합.
    - 포인터를 저장할 공간이 필요하므로 배열보다 구조가 복잡.

배열 합 & 구간의 합 구하기

정의

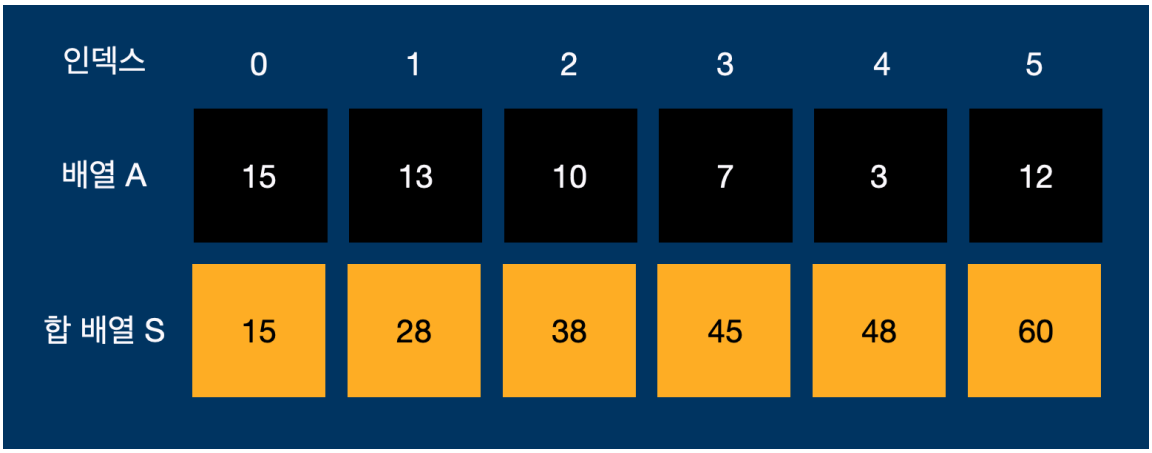
- 구간 합은 합 배열을 이용하여 시간 복잡도를 더 줄이기 위해 사용하는 알고리즘.

합 배열

- 구간 합 알고리즘을 활용하려면 합 배열을 구해야 함.
- 배열 A가 있을 때 합 배열 S는 다음과 같이 정의.
  - 합 배열 S 정의

```
// A[0] 부터 A[i] 까지의 합
S[i] = A[0] + A[1] + A[2] + ... + A[i-1] + A[i]
```

- 합 배열을 미리 구해 놓으면, 일정 범위의 합을 구하는 시간 복잡도가 O(n) → O(1) 로 감소.



- 합 배열 S를 만드는 공식

```
S[0] = A[0]
// i > 0
S[i] = S[i-1] + A[i]
```

```
function makeSumArr(arr) {
  const s = new Array(arr.length);

  for (let i = 0; i < arr.length; i++) {
    if (i === 0) {
      s[i] = arr[i];
    } else {
      s[i] = s[i - 1] + arr[i];
    }
  }

  return s;
```

```
}

const a = [15, 13, 10, 7, 3, 12];

console.log(makeSumArr(a)); // [ 15, 28, 38, 45, 48, 60 ]
```

- 구간 합을 구하기 공식

```
// i에서 j까지 합
// i가 0 이면 S[j] 만 리턴
// i, j 유효한 값이라 가정 (배열 길이를 넘어서거나 i 가 j보다 큰 경우 제외)
// i > 0
S[j] - S[i-1]
```

```
const S = makeSumArr(a); // [ 15, 28, 38, 45, 48, 60 ]

function outputRangeSum(i, j) {
  if (i === 0) {
    return S[j];
  }
  return S[j] - S[i - 1];
}

console.log(outputRangeSum(1, 3)); // 30
console.log(outputRangeSum(0, 5)); // 60
```