

# week01 time complexity, recursion, brute-force

2023-03-30

알고리즘이란? 문제를 해결하기 위한 절차

알고리즘은 구체적일수록 좋은 것인가?

- 알고리즘 ≠ 프로그래밍
  - 알고리즘을 설명할 때는 청자가 사람이라는 사실을 고려한다.
    - 프로그램도 결국 문제를 해결하는 과정을 기술한 것이지만 기계를 대상으로 하는 설명
    - 핵심 아이디어를 중심으로 어느 정도까지 세부적으로 설명할지 생각하기
  - 알고리즘을 이해하되 어떻게 구현할지 생각하기
    - 구현에 익숙하지 않다면 알고리즘의 각 단계를 어떻게 구현할지도 고려
    - 적합한 자료구조를 이용해 구현 → 알고리즘 공부와 자료구조 공부는 동시에 병행!



알고리즘의 성립 조건

1. 항상 올바른 답을 낼 것
2. 유한한 시간 안에 종료될 것



좋은 알고리즘이란?

- 누구나 쉽게 이해할 수 있으며 간결
- 주어진 자원의 한계 고려 (시간, 공간)

## 시간복잡도

성능을 분석하기 위한 방법

1. 시간 → 절대적인 시간을 기준으로 하면 코드가 작동하는 환경에 많은 영향을 받음
2. 연산 횟수 ✓✓

점근적 표기법 사용!

## Big-Ω

모든  $n > n_0$ 에 대해  $cg(n) \leq f(n)$ 인 양의 상수  $c$ ,  $n_0$ 가 존재하면  $f(n) = \Omega(g(n))$

## Big-θ

모든  $n > n_0$ 에 대해  $c_1g(n) \leq f(n) \leq c_2g(n)$ 인 양의 상수  $c_1$ ,  $c_2$ ,  $n_0$ 가 존재하면  $f(n) = \theta(g(n))$

## Big-O

| 자주 사용하는 시간 복잡도 표기 |                                 |
|-------------------|---------------------------------|
| $O(1)$            | 상수 시간 (Constant Time)           |
| $O(\log n)$       | 로그 시간 (대수 시간, Logarithmic time) |
| $O(n)$            | 선형 시간 (Linear time)             |
| $O(n \log n)$     | 로그 선형 시간 (Log-linear time)      |
| $O(n^2)$          | 제곱 시간 (Quadratic time)          |
| $O(n^3)$          | 세제곱 시간 (Cubic time)             |
| $O(2^n)$          | 지수 시간 (Exponential time)        |

### $O(1)$

- 입력값이 증가해도 시간이 늘어나지 않음 → 상수 시간 복잡도

- ex) 배열에 index로 직접 접근하는 경우

## **$O(\log n)$**

- ex) 이진 탐색 트리 계열의 자료구조 삽입, 삭제, 탐색 등

## **$O(n)$**

- 입력값이 증가함에 따라 시간도 선형적으로 증가
- ex) 연결리스트의 탐색, 연산

## **$O(n \log n)$**

- ex) 퀵 정렬, 병합 정렬

## **$O(n^2)$**

- ex) 삽입, 선택, 버블 정렬

## **$O(2^n)$**

- ex) 피보나치


```
int fibonacci(int n) {
    if (n <= 1)
        return 1;

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

# 재귀

호출된 함수가 자기 자신을 다시 호출하는 것

재귀함수를 구현하기 위해서 정해야 할 것

1. 언제 어떤 매개변수를 가지고 재귀를 호출할지
2. 호출을 정지시켜줄 base case  필수!

ex) 팩토리얼 재귀 코드

```
int factorial(int n) {
    //base case
    if (n <= 0)
        return 1;
```

```
    return n * factorial(n-1);  
}
```

모든 재귀함수는 재귀 구조 없이 반복문만으로도 구현 가능!

재귀를 잘 활용하면 코드가 간결해지지만, 메모리와 시간적 측면에서의 손해를 감수해야함

## brute-force

가능한 모든 경우의 수를 탐색하며 요구 조건에 충족되는 결과만을 가져오는 방식

- 선형 구조 → 순차 탐색
  - 비선형 구조 → BFS, DFS
- ➡ 그래프 구조 학습 후 다루기!

## 순차 탐색

순서대로 모든 경우를 탐색하는 것

1. 문제에서 주어진 자료를 선형 구조로 바꾼다
2. 구조화된 자료들을 구조에 맞는 방법으로 해를 구성할 때까지 탐색
3. 구성된 해 정리

ex) 10의 약수의 합 구하기

1. 10의 약수가 될 수 있는 자연수를 구조화  
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
2. 선형 구조이므로 반복문을 돌며 조건에 맞는 해를 처음부터 끝까지 탐색
3. 구성된 해를 모두 더하기

```
int result[11];  
for (int i = 1; i <= 10; i++) {  
    if (10 % i == 0)  
        result[i] = 1;  
}  
  
int sum = 0;  
for (int i = 1; i <= 10; i++) {  
    if (result[i] == 1)  
        sum += i;  
}  
cout << sum;
```

## 연습문제

### boj 10870 피보나치수 5 (B2)

<https://www.acmicpc.net/problem/10870>

```
// boj 10870 피보나치 수 5
// recursion
#include <iostream>
using namespace std;

int n; // 0~20

int fibo(int x) {
    if (x < 2) return x;
    return fibo(x - 2) + fibo(x - 1);
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    cin >> n;
    cout << fibo(n);

    return 0;
}
```

### boj 1018 체스판 다시 칠하기 (S4)

<https://www.acmicpc.net/problem/1018>

```
// boj 1018 체스판 다시 칠하기
// brute-force
#include <iostream>
using namespace std;

int N, M;
char board[51][51];

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    cin >> N >> M;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) cin >> board[i][j];
    }

    int ans = 2500;
```

```

for (int i = 0; i <= N - 8; i++) { // 전체 체스판 탐색
    for (int j = 0; j <= M - 8; j++) {
        // 좌상단이 B인 경우: 맞을 때 cnt1 증가, 틀릴 때 cnt2 증가
        // 좌상단이 W인 경우: 맞을 때 cnt2 증가, 틀릴 때 cnt1 증가
        int cnt1 = 0, cnt2 = 0;

        for (int a = i; a < i + 8; a++) {
            for (int b = j; b < j + 8; b++) {
                // board[a][b]가 W인 경우 좌상단이 W
                if (board[a][b] == 'B') { // 한 칸 건너뛰며 반복되므로 행, 열의 합이 짝수/홀수인 경우로 나눔
                    if ((a + b) % 2 == 0)
                        cnt1++;
                    else
                        cnt2++;
                } else {
                    if ((a + b) % 2 == 0)
                        cnt2++;
                    else
                        cnt1++;
                }
            }
        }
        int temp = min(cnt1, cnt2);
        ans = min(ans, temp);
    }
}
cout << ans;
return 0;
}

```

## boj 9020 골드바흐의 추측 (S2)

<https://www.acmicpc.net/problem/9020>

### 1. 결과는 제대로 쓰지만 시간초과

```

// boj 9020 골드바흐의 추측
// brute-force
#include <iostream>
using namespace std;

int T, n; // n은 4~10000의 짝수

bool prime(int x) { // 소수 판별
    for (int i = 2; i < x; i++) {
        if (x % i == 0) return false;
    }
    return true;
}

void goldbach(int x) {
    int a = 0, b = 0;
    for (int i = 2; i < x / 2 + 1; i++) {
        if (prime(i) && prime(x - i)) {
            int temp = x;
            if (x - i - i < temp) {
                a = i;
                b = x - i;
            }
        }
    }
}

```

```

    }
}
cout << a << " " << b << "\n";
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    cin >> T;
    for (int i = 0; i < T; i++) {
        cin >> n;
        goldbach(n);
    }
    return 0;
}

```

## 2. 소수 판별 함수 prime의 for문을 수정함 ⇒ 시간초과 해결!

```

// boj 9020 골드바흐의 추측
// brute-force
#include <iostream>
using namespace std;

int T, n; // n은 4~10000의 짝수

bool prime(int x) { // 소수 판별
    for (int i = 2; i * i <= x; i++) {
        if (x % i == 0) return false;
    }
    return true;
}

void goldbach(int x) {
    int a = 0, b = 0;
    for (int i = 2; i < x / 2 + 1; i++) {
        if (prime(i) && prime(x - i)) {
            int temp = x;
            if (x - i - i < temp) {
                a = i;
                b = x - i;
            }
        }
    }
    cout << a << " " << b << "\n";
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    cin >> T;
    for (int i = 0; i < T; i++) {
        cin >> n;
        goldbach(n);
    }
    return 0;
}

```

- $n$ 보다 작거나 같은 수에 대해 소수인지 판별하는 경우  $\sqrt{n}$ 까지만 확인하는 방법!

- 약수의 중심을 구하는 원리

ex) 80의 약수는 1, 2, 4, 5, 8, 10, 16, 20, 40, 80

$$\sqrt{80} = 8.xx$$

- 절반 이상의 숫자는 확인할 필요가 없음. 자기 자신을 제외하고 절반을 초과하는 숫자에서 나뉘을 때 나머지가 0이 되는 숫자는 나올 수 없다!
- 절반까지의 숫자를 확인하는 경우  $O(n/2)=O(n)$
- root를 씌운 값까지만 확인하는 경우  $O(\sqrt{n})$