알고리즘이란?

문제를 푸는 잘 정의된 과정 -> input을 output으로 바꾸는 과정

Sequential Search

n개의 key가 있는 정렬되지 않은 array S에서 처음부터 마지막까지 하나씩 검사하여 key x를 찾는 것.

worst case -> n

best case -> 1

Binary Search

n개의 key가 있는 정렬된 array S에서 key x가 있는지 찾는 것

정렬된 리스트에서 검색 범위를 줄여나가면서 검색값을 찾는다.

검색이 반복될 때마다 검색 범위가 절반으로 줄어 속도가 빠르다.

worst -> logn+1

best -> 1

Fibonacci Sequence

피보나치 수열에서 n번째 항을 구하는 문제

해결법 1 – 재귀

```
int fib(int n) {
        if (n <= 1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

한 번 구한 값을 계속해서 구하게 하는 비효율을 야기함 -> 한 번 구한 값을 저장해 비효율을 개선하자!

해결법 2 - iterative

```
int fib2(int n) {
        index i;
        int f[0 ... n];
        f[0] = 0;
        if (n > 0) {
                f[1] = 1;
                for (i = 2; i <= n; i++) f[i] = f[i-1] + f[i-2];
        }
        return f[n];
}
```

재사용해 더 빠르다

알고리즘의 효율성 -> 얼마나 빠르게 실행되느냐(Time Complexity), 많은 공간을 연산하는데 사용하는가(Space Complexity)

Every-Case analysis

경우가 나뉘어지지 않고, 한 가지 경우만 존재. 입력크기 n에만 종속할 뿐, 결과가 항상 일정

ex) 배열값을 더하는 알고리즘 -> 배열 내용에 상관없이 for루프가 n번 반복

Worst-Case analysis

경우가 나뉘어지고 그중 최악의 경우. 단위 연산이 수행되는 횟수가 최대

문제를 풀기 위한 시간이 1, 2, 3, 4로 나왔다면 4의 시간으로 어떤 문제든 풀 수 있다.

즉, 모든 문제를 풀 수 있는 시간은 4이다.

Best-Case analysis

연산 수행 횟수가 최소인 경우

Average-Case analysis

모든 경우를 분석해 그 평균을 낸 것. 입력 크기와 입력 값(내용) 모두에 종속.

확률을 바탕으로 계산

Optimality

알고리즘이 최적이다 -> 이론 상으로는 더 이상 빠르게 문제를 풀 수 없다

Optimal한 지 어떻게 판단?

알고리즘의 complexity와 문제의 complexity가 만나는 지점이 optimal

점근 표기법(Asymptotic Notation)

시간 복잡도 또는 공간 복잡도 함수의 증가 양상을 구분하기 위해 사용하는 표기법

O(f(n))

점근적 상한, asymptotic upper bound

g(n) ∈ O(f(n))일 때, f(n)이 g(n)의 상항이 된다.

주어진 복잡도 함수 f(n)에 대해 g(n) <= c * f(n)을 만족하는 음이 아닌 정수 n과 상수 c가 존재한다.

즉, g(n)이 f(n)보다 빠르거나 같다.

Ω(f(n))

점근적 하한, asymptotic lower bound

g(n) ∈ Ω(f(n))일 때, f(n)이 g(n)의 하한이 된다.

주어진 복잡도 함수 f(n)에 대해 g(n) > c * f(n)을 만족하는 음이 아닌 정수 n과 상수 c가 존재한다.

즉, g(n)이 f(n)보다 느리거나 같다.

Θ(f(n))

Θ(n)= O(f(n)) ∩ Ω(f(n))일 때, Θ(n)은 가능한 모든 g(n)의 집합이다.

주어진 복잡도함수 f(n)에 대해 c * f(n) ≤ g(n) ≤d * f(n)을 만족 만족하는 음이 아닌 정수 n과 상수 c가 존재한다.

g(n) ∈ Θ(f(n))일 때, g(n)은 f(n)의 차수(order)라고 한다.