06. 자료구조와 알고리즘

02. 알고리즘의 분석

학습목표

- 시간 복잡도와 공간 복잡도를 이해할 수 있다.
- 시간 복잡도와 공간 복잡도의 관계에 대해 이해할 수 있다.
- 알고리즘의 효율성을 판단할 수 있다.

들어가며

알고리즘(Algorithm)이란 문제를 해결하기 위한 절차를 말한다. 문제를 해결하는 방법에는 여러 가지가 존재할 수 있으며, 우리는 이 중 최선을 선택해야 한다. 다시 말해 알고리즘을 분석하고 비교할 줄 알아야 한다. 알고리즘을 분석하는 것은 컴퓨터 **자원**(Resource)*의 사용량을 분석하는 것이라고 볼 수 있는데, 자원을 적게 사용할수록 효율적인 알고리즘이라고 할 수 있다. 이번 시간에는 효율적인 알고리즘의 척도인 시간 복잡도와 공간 복잡도에 대해서 배워보도록 한다.

*자원이란 실행 시간, 메모리 사용량 등을 말한다.

시간 복잡도

알고리즘의 분석에 있어 가장 중요한 부분은 실행 시간이다. 알고리즘의 실제 실행은 하드웨어 및 소프트웨어 환경에 따라 천차만별이므로 단순 측정으로는 실행 시간을 분석할 수 없다. 다시 말해 하드웨어와 소프트웨어 환경을 배제한 객관적인 지표가 필요하다. 이를 위해서 시간 복잡도를 사용한다.

시간 복잡도(Time Complexity)는 알고리즘을 수행하는 데 필요한 연산이 몇 번 실행되는지를 숫자로 표기한다. 이 연산의 개수는 상수가 아니라 입력한 데이터의 개수를 나타내는 n에 따라 변하게 된다. 그래서 연산의 개수 n의 값에 따라 시간 복잡도를 나타낸 것을 시간 복잡도 함수라고 하며 수식으로는 $T(n)$이라고 표기한다. 일례로 양의 정수 n을 n번 더하는 알고리즘을 구현한다고 할 때, 이에 따른 시간 복잡도 함수를 확인해보자.

```
// Case 1
int sum = n * n;
```

```
// Case 2
int sum = 0;
for (int i = 0; i < n; ++i)
  sum += n;

// Case 3
int sum = 0;
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    sum += 1;
```

위 3개의 알고리즘의 연산 횟수를 비교해보면 아래와 같이 나타낼 수 있다. (간단히 나타내기 위해 루프 제어 연산은 제외한다.)

연산 종류	Case 1	Case 2	Case 3
대입 연산	1	$n + 1$	$n^2 + 1$
덧셈 연산		n	n^2
곱셈 연산	1		
전체 연산 횟수	2	$2n + 1$	$2n^2 + 1$

하나의 연산이 t만큼의 시간이 필요하다고 하면 이에 대한 시간 복잡도 함수를 각각 $2t$, $(2n + 1)t$, $(2n^2 + 1)t$로 나타낼 수 있다.

시간 복잡도를 구했으므로 알고리즘의 수행 시간을 측정할 수 있게 됐다. 위의 예시를 바탕으로 입력값에 따라 어느 정도의 시간이 걸리는 지를 계산해보자. 편의상 모든 연산에 걸리는 시간은 1ms라고 가정한다.

입력값	Case 1	Case 2	Case 3
$n = 1$	2ms	3ms	3ms
$n = 10$	2ms	21ms	201ms

$n = 100$	2ms	201ms	20,001ms = 약 20s
$n = 1000$	2ms	2,001ms = 약 2s	2,000,001ms = 약 33m
$n = 10000$	2ms	20,001ms = 약 20s	200,000,001ms = 약 55h

입력값이 작을 때는 그 차이가 크지 않았지만, 입력값이 점점 커질수록 엄청나게 차이가 벌어진다는 것을 볼 수 있다.* 그러므로 우리는 알고리즘을 효율적으로 구현하도록 늘 고민해야 한다.

*이를 실행 시간의 **성장률**(rate of growth)이라고 한다.

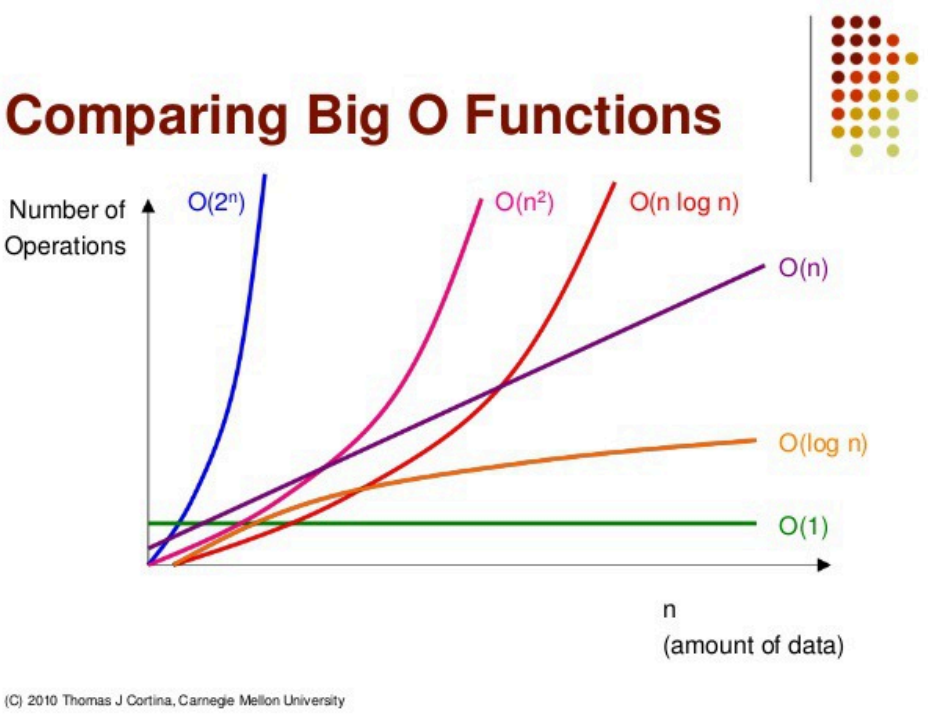
Big-O 표기법

앞의 결과값을 다시 한번 관찰해보자. 다항식의 시간 복잡도 함수에서 입력값이 커져감에 따라 각 항의 결과값에 관여하는 정도가 달라지게 됨을 알 수 있다. 일례로 Case3의 시간 복잡도 함수 $T(n) = 2n^2 + 1$에서 입력값을 10000을 넣었을 때, 각 항의 계산값은 $2(10,000)^2 = 200,000,000$과 1 이므로 $2n^2$대비 1의 영향이 매우 작음을 알 수 있다.

그래서 시간 복잡도와 뒤에 배울 공간 복잡도를 표시할 때는 각 함수의 모든 항을 표시하지 않으며 상수 계수와 중요하지 않은 항목을 제거한 **점근적 표기법**(Asymptotic Notation)을 사용한다. 여기에는 Big-θ, Big-O, Big-Ω 등이 있지만 이중 가장 많이 사용되는 것은 Big-O 표기법이다. Big-O 표기법은 점근적 상한선을 제공한다.

앞선 시간 복잡도 함수를 Big-O 표기법으로 나타내면 각각 $O(1)$, $O(n)$, $O(n^2)$으로 나타낼 수 있다. 많이 쓰이는 Big-O 표기법은 아래와 같으며 오른쪽으로 갈 수록 상한선이 높아진다. 다시 말해 왼쪽에 위치한 시간 복잡도일수록 해당 알고리즘은 빠르다고 할 수 있다.

$$O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$



유의할 점은 한 알고리즘이 다른 알고리즘보다 실제로 빠르다고 하더라도 같은 방식으로 표현이 될 수도 있다는 것이다. 따라서 같은 카테고리라면 어떤 알고리즘이 더 빠른지 면밀한 분석이 필요할 때가 있다.

공간 복잡도

알고리즘의 성능을 측정할 때는 실행 시간만 따지지 않는다. 컴퓨터 내의 자원 공간을 얼마나 사용하는지도 따져봐야 한다. **공간 복잡도**(Space Complexity)는 알고리즘을 수행하는 데 필요한 자원 공간의 양을 말한다. 공간 복잡도 함수는 고정 공간* 요구 + 가변 공간** 요구로 나타낼 수 있으며 수식으로는 $S(P) = c + S_p(n)$으로 표기한다.

*고정 공간 : 입력과 출력의 횟수나 크기와 관계 없는 공간

**가변 공간 : 문제를 해결하기 위해 요구되는 추가 공간

팩토리얼의 구현을 통해 공간 복잡도를 살펴보자.

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

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

```
int factorial(n) {
    int sum = 1;
    for (int i = n; i > 1; --i) {
        sum *= i;
    }

    return sum;
}
```

재귀적으로 구현한 함수의 공간 복잡도는 $O(n)$이며, 아래는 $O(1)$이다.

분석하기

$O(n^2)$까지의 코드를 분석해보자.

```
void Foo(int n)
{
  // 어떤 입력이 들어와도 연산의 횟수 동일
  Console.WriteLine("O(1)");
}



void Foo(int n)
{
  if (n == 1)
  {
    return;
  }

  // 입력에 대해 로그를 따름
  // Ex.
  // 2 <= N < 4 => 1회 실행
  // 4 <= N < 8 => 2회 실행
```

```
  // 8 <= N < 16 => 3회 실행
  // 16 <= N < 32 => 4회 실행
  // 32 <= N < 64 => 5회 실행
  // 1024 <= N < 2048 => 10회 실행
  Console.WriteLine("O(logn)");

  Foo(n / 2);
}

void Foo(int n)
{
  for (int i = 0; i < n; ++i)
  {
    // 입력에 따라 늘어남
    Console.WriteLine("O(n)");
  }
}



// 위의 logn이 n번 반복됨. 그래서 O(nlogn)
for (int i = 0; i < n; ++i)
{
  Foo(n);
}



void Foo(int n)
{
  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < n; ++j)
    {
      // 입력에 대해 제곱으로 진행 됨
      Console.WriteLine("O(n^2)");
```

```
    }
  }
}
```

시간 복잡도 vs 공간 복잡도

여태까지 배운 내용을 토대로 정리하자면 효율적인 알고리즘은 실행 시간이 적게 걸리며, 자원을 적게 소모하는 것이라고 할 수 있다. 하지만 시간과 공간은 반비례적인 경향이 있어 보통 알고리즘의 척도는 시간 복잡도를 위주로 판단한다. 즉, 시간 복잡도를 위해 공간 복잡도를 희생하는 경우가 많다.

최선, 최악, 평균의 경우

알고리즘을 분석할 때는 **최선의 경우**(Best Case)와 **최악의 경우**(Worst Case), **평균의 경우**(Average Case)로 나눠 파악해야 한다. 각각의 경우는 알고리즘의 효율성이 가장 좋을 때, 가장 나쁠 때, 평균적일 때를 말한다. 알고리즘 선택 및 개발 시 최선의 경우 혹은 평균의 경우는 매우 드물게 고려하며, 일반적으로는 최악의 경우를 고려한다.*

* 그래서 점근적 상한선인 Big-O 표기법을 사용하는 것이기도 하다.

더해보기

1. 시간 복잡도와 공간 복잡도는 어떤 상관 관계를 갖고 있나요?
2. 병합 정렬의 최선, 최악, 평균의 경우 시간 복잡도 및 공간 복잡도는 어떻게 되나요? 또 왜 그런가요?

참고자료

- C로 배우는 쉬운 자료구조
- 누구나 자료 구조와 알고리즘
- [무료] C로 배우는 자료구조 및 여러가지 예제 실습
- Worst, Average and Best Case Analysis of Algorithms - GeeksforGeeks