POINTER 알고리즘 세미나

제 1강 - 시간 복잡도 (Time complexity)와 분석 (Analysis)

Kim Jun Hyeok

Kangwon Science Highschool

August 15, 2020

1 시간 복잡도 (Time complexity)

시간 복잡도라는 것은 어떤 알고리즘의 수행 시간을 입력에 관한 크기를 독립 변수로 두고 하나의 실함수 (Real function)으로 나타내었을때 그 함수의 증가의 정도를 나타낸 것을 의미한다.
아무리 생각해도 직감적으로 감이 오지는 않으니 한번 예제를 통해서 살펴보자.

Listing 1: $1^2 + ... + n^2$을 반환하는 코드이고 시간복잡도는 $O(a)$이다.

```
1  int run(int n) {
2      int ans = 0;
3      for (int i = 1; i <= n; i++)
4          ans += i*i;
5      return ans;
6  }
```

Listing 2: $1^2 + ... + n^2$을 반환하는 코드이고 시간복잡도는 $O(a^2)$이다.

```
1  int run(int n) {
2      int ans = 0;
3      for (int i = 1; i <= n; i++)
4          for (int j = 1; j <= i; j++)
5              ans += i;
6      return ans;
7  }
```

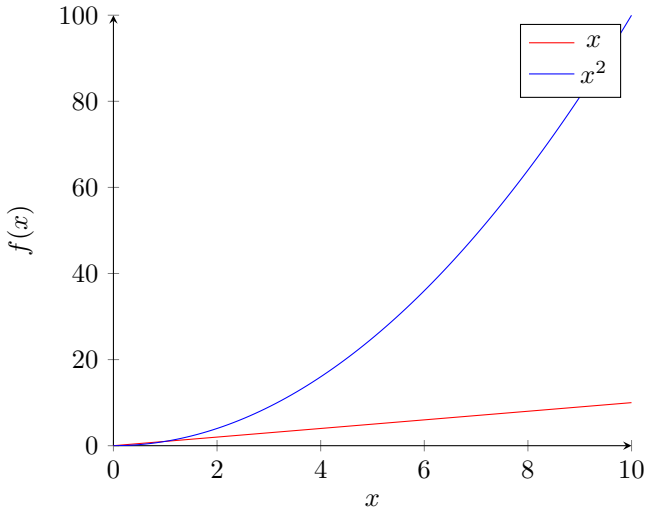

Figure 1: x, x^2의 증가의 정도를 표현하기 위한 그래프이다.

1, 2 모두 보았을때 직관적으로 1이 2보다 효율적임을 직관적으로 알 수 있다.
이제 각각을 분석해보자. 1같은 경우 입력의 크기 n에 관하여 덧셈 연산이 총 n번 수행되고, 2같은 경우 입력의 크기 n에 관하여 덧셈 연산이 총 $1 + ... + n = \frac{n(n+1)}{2}$번 수행된다.

이렇게 본다면, 별 차이 없을 수도 있지만 가령 $n = 10^8$이라고 해보자. 그러면 2는 약 $n^2 = 10^{16}$번의 덧셈 연산을 수행한다는 이야기인데, 1Ghz의 성능을 가진 CPU가 1초에 10^9개의 명령어 (Instruction)을 처리할 수 있는데, 2020년 기준 아직까지도 등장한 적이 없는 100Ghz짜리 CPU가 처리해도 매우 오랜시간이 걸릴 것이다.

또한, 앞서 이야기한 문제인 $1^2 + ... + n^2$을 구하는 1보다 더 빠른 방법은 있지만, 지금은 논외로 한다. 이제 이것을 시간 복잡도 (Time complexity)를 통해서 분석해보자.
처음에 이야기한 시간 복잡도에서 연산에 대한 수행시간은 직접 계산하지는 않고 간략하게 입력의 크기와 수행 시간이 얼마나 비례해서 증가하는지, 그것을 나타내어야 한다. 대개 연산 횟수랑 수행 시간은 비례 관계에 있으므로 연산 횟수를 각 알고리즘마다 구해야 한다.

여기에서 **연산**은, 무조건 상수 시간에 수행되는 연산으로 잡아야 한다. 상수 시간이라는 것은 입력의 크기에 비례해서 연산 수행시간이 증가하지 않는 연산을 의미한다. 앞서 말한 덧셈 연산은 물론 컴퓨터 내부적으로는 가산기 (adder)라는 것으로 비트 수에 비례해서 연산이 이루어지지만, 비트 수가 컴퓨터 시스템에 따라서 고정되어있으므로 (우리가 일상적으로 볼 수 있는 32bits, 64bits 컴퓨터가 그런 의미이다) 상수 시간으로 볼 수 있다.

먼저 1를 다시 분석해보자. 입력 크기 n에 따른 수행되는 덧셈 연산의 횟수를 $T(n)$이라고 잡자. 그러면 3번째 줄의 for 반복은 a번 수행되고 4번째 줄에 for 반복 내에서 덧셈 연산이 1번 이루어지니 (곱셈도 있지만 우리가 중요한건 그게 아니다) $T(n) = n$이라고 할 수 있다. 그러나 우리는 증가의 정도를 나타내기로 했으니, 이것을 잘 표현할 수 있는 Big-O 표기법을 사용하도록 한다.

Definition 1 (Big-O 표기법의 정의)**.** $O(f(x)) = g(x)$ 라는 것은 $g(x)$의 정의역의 임의의 원소 x에 관하여 $g(x) \leq c \times f(x)$를 만족하는 상수 c가 존재한다는 것이다.

1를 이제 써먹을 준비가 되었다. 1의 덧셈 연산 횟수 $T(n) = n$이니 $T(n) = O(n)$이다. 왜냐하면 $n \leq c \times n$은 매우 자명하기 때문이다. 이제 2를 분석해보자. 덧셈 연산 횟수 $T(n) = 1 + ... + n = \frac{n(n+1)}{2}$이다. 우리는 $\frac{n(n+1)}{2} \leq c \times n^2$을 만족하는 상수 c가 존재하는지에 대해서 판별을 해야되는데, $n(n+1) \leq 2c \times n^2 \Rightarrow n \leq (2c-1)n^2 \Rightarrow 1 \leq (2c-1)n$이고 정의역은 결국 자연수 ($\mathbb{N}$)이니 $c = 1$이면 충분하다. 따라서 $T(n) = O(n^2)$이고 2의 시간 복잡도는 $O(n^2)$이다.

물론, 시간 복잡도가 다항 함수가 아닐 수도 있다. 예를 들어서 BOJ 1065 - 한수를 보자. 필자가 작성한 정답 코드는 다음과 같다.

Listing 3: BOJ 1065 - 한수를 푸는 코드, 브루트 포스 (Bruteforce)기법을 사용하였다.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(void) {
6     ios_base::sync_with_stdio(false);
7     cin.tie(nullptr);
8     int n = 0;
```

```
9       cin >> n;
10      int cnt = 0;
11      for (int i = 1; i <= n; i++) {
12          if (i <= 10) {
13              cnt++;
14          } else {
15              int i_t = i;
16              int delta = ((i_t / 10) % 10) - (i_t % 10);
17              bool failed = false;
18              while ((i_t/10) != 0) {
19                  int a = i_t % 10;
20                  int b = (i_t / 10) % 10;
21                  if (delta + a != b) {
22                      failed = true;
23                      break;
24                  }
25                  i_t /= 10;
26              }
27              if (!failed)
28                  cnt++;
29          }
30      }
31      cout << cnt;
32      return 0;
33  }
```

3의 18-26번째 줄에서는 while 루프를 도는데 조건이 특이하다. 바로 특정한 값 i를 계속 10으로 나누고 0이 될때까지 반복한다는 의미인데, 결국 끝날때 까지의 반복 횟수는 약 $\log i$이다. 즉, 시간 복잡도에 $\log i$가 들어간다는 말이다. 이제 나눗셈 연산 기준으로 분석해보자. 우선 비트수는 한정되어있으니 나눗셈 연산은 상수 시간 (constant time)으로 가정해도 문제는 없을 것이고, $T(n) = \log 1 + ... \log n = \log n!$ 이다. 여기에서 Stirling's approximation [1]을 적용시키면, $\log n! = O(\log n^n =)$

References

[1] "Stirling's approximation," Jul 2020. [Online]. Available: https://en.wikipedia.org/wiki/Stirling's_approximation