

1-1. Introduction to big O

#_area/coding_test/leetcode/1_intro

Before we talk about big O, it's important that we first understand what exactly an "algorithm" is, especially in the context of LeetCode.

우리가 빅오(Big O)에 대해 이야기하기 전에, 먼저 '알고리즘'이 정확히 무엇인지 이해하는 것이 중요합니다. 특히 리트 코드(LeetCode)와 같은 상황에서는 더욱 그렇습니다.

An algorithm can be seen as a recipe for a computer to follow. It's a set of instructions that a computer will follow step-by-step to solve a problem.

알고리즘은 컴퓨터가 따라야 하는 '요리법'이라고 생각하면 됩니다. 어떤 문제를 해결하기 위해 컴퓨터가 단계별로 따라야 하는 일련의 지침입니다.

Algorithms take an input and produce an output. The output will be the answer to a question regarding the input. For example, let's say you had a non-empty array of positive integers called `nums`, and you wanted to answer the question: "What is the largest number in `nums`?"

알고리즘은 입력(input)을 받아서 출력(output)을 생성합니다. 이 출력은 입력과 관련된 질문에 대한 정답이 됩니다. 예를 들어, `nums` 라는 비어있지 않은 양의 정수 배열이 있을 때, "이 배열에서 가장 큰 숫자는 무엇인가?"라는 질문에 답하고 싶다고 가정해봅시다.

To answer this question, you would write an algorithm that takes an array called `nums` as **input**, and **outputs** the largest number in `nums`. Here is an example of such an algorithm:

1. Create a variable `maxNum` and initialize it to `0`.
2. Iterate over each element `num` in `nums`.
3. If `num` is greater than `maxNum`, update `maxNum = num`.
4. Output `maxNum`.

이 질문에 답하려면 `nums` 라는 배열을 **입력**으로 받고, 배열의 가장 큰 숫자를 **출력**하는 알고리즘을 작성할 것입니다. 다음은 이러한 알고리즘의 예시입니다.

1. `maxNum`이라는 변수를 만들고 0으로 초기화합니다.
2. 배열 `nums`의 각 요소 `num`을 하나씩 순회합니다.
3. 만약 현재의 `num`이 `maxNum`보다 크다면, `maxNum`의 값을 `num`으로 업데이트합니다.
4. 최종적으로 `maxNum`을 출력합니다.

Here, we have written down a set of instructions that when followed, will solve the problem. We can now implement these instructions in code so that a computer can quickly solve the problem. There are some important requirements for algorithms in the context of LeetCode:

- Algorithms should be **deterministic**. Given the same **input**, the algorithm should **always** produce the same **output**. Basically, there shouldn't be any randomness.
- The algorithm should be correct for any arbitrary valid **input**. In our example, we stated that `nums` is a non-empty array of positive integers. There are infinitely many such arrays, and our algorithm works for **all** of them. However, if `nums` contained negative numbers, the input would be invalid since we specifically required positive integers. In fact, our algorithm would break in such a case because we initialized `maxNum` to `0`. If `nums` were entirely negative, `maxNum` would remain `0` since no negative number would be greater than `0`, leading to an incorrect result. Instead, we should initialize `maxNum` to the first element of `nums` to ensure the maximum value is always selected from the array itself.

이렇게 하면 문제를 해결하기 위해 따라야 할 지침을 작성한 것입니다. 이제 이 지침을 코드로 구현하면 컴퓨터가 빠르게 문제를 해결할 수 있게 됩니다. 리트코드와 같은 환경에서는 알고리즘이 다음과 같은 중요한 조건을 충족해야 합니다.

- 알고리즘은 **결정적(deterministic)**이어야 합니다. 즉, 동일한 **입력**이 주어지면 항상 동일한 **출력**이 나와야 합니다. 다시 말해, 무작위성(randomness)이 포함되면 안 됩니다.
- 알고리즘은 임의의 유효한 **입력**에 대해 항상 올바른 결과를 내야 합니다. 위의 예제에서 우리는 `nums`가 비어있지 않은 양의 정수 배열이라고 명시했습니다. 이러한 배열은 무한히 많이 존재할 수 있으며, 우리의 알고리즘은 그런 배열들에 대해 모두 정상 작동합니다. 하지만 만약 `nums`에 음수가 포함된다면, 우리가 애초에 요구한 '양의 정수'라는 조건을 위배하기 때문에 입력이 무효해집니다. 사실 이 경우 우리의 알고리즘은 잘못된 결과를 냅니다. 왜냐하면 우리는 `maxNum`을 `0`으로 초기화했기 때문입니다. 배열의 모든 요소가 음수일 경우, 어떤 음수도 `0`보다 크지 않기 때문에 `maxNum`은 계속 `0`으로 남게 되고, 결국 틀린 결과가 나옵니다. 따라서, 이런 문제를 방지하기 위해서는 `maxNum`을 배열의 첫 번째 원소로 초기화해야 합니다. 이렇게 하면 항상 배열 내의 값 중에서 최대값이 선택되게 됩니다.

Big O

Big O is a notation used to describe the computational complexity of an algorithm. The computational complexity of an algorithm is split into two parts: time complexity and space complexity. The time complexity of an algorithm is the amount of time the algorithm needs to run relative to the input size. The space complexity of an algorithm is the amount of memory allocated by the algorithm when run relative to the input size.

Big O는 알고리즘의 **계산 복잡도**(computational complexity)를 설명하는 데 사용되는 표기법입니다. 알고리즘의 계산 복잡도는 크게 **시간 복잡도**(time complexity)와 **공간 복잡도**(space complexity)로 나누어집니다. **시간 복잡도**란 알고리즘이 입력 크기에 비례하여 실행되는 데 필요한 시간을 의미합니다. 반면, **공간 복잡도**란 입력 크기에 비례하여 알고리즘이 실행될 때 할당하는 메모리의 양을 의미합니다.

Typically, people care about the time complexity more than the space complexity, but both are important to know.

일반적으로 사람들은 공간 복잡도보다 시간 복잡도를 더 중요하게 여기지만, 두 가지 모두 알고 있는 것이 중요합니다.

Time complexity: as the input size grows, how much longer does the algorithm take to complete?

Space complexity: as the input size grows, how much more memory does the algorithm use?

시간 복잡도: 입력 크기가 커질수록 알고리즘의 실행 시간이 얼마나 더 길어지는가?

공간 복잡도: 입력 크기가 커질수록 알고리즘이 얼마나 더 많은 메모리를 사용하는가?

How complexity works

Complexity is described by a function (math formula). What should the arguments to this function be?

복잡도는 함수(수학적 공식)로 표현됩니다. 그렇다면 이 함수의 인자(arguments)는 무엇이어야 할까요?

The arguments are variables defined by the programmer, but they should represent values that change between different inputs, and these values should affect the algorithm. For example, the most common variable you'll see is n , which usually denotes the length of an input array or string. In the example above, we could say that n is equal to the length of `nums`.


함수의 인자는 프로그래머가 정의하는 변수들이지만, 이 변수들은 입력에 따라 변하는 값이어야 하며, 그 값들이 알고리즘에 영향을 미쳐야 합니다. 예를 들어, 가장 흔히 볼 수 있는 변수는 보통 입력 배열이나 문자열의 길이를 나타내는 n 입니다. 위에서 살펴본 예시에서는, n 을 배열 `nums`의 길이라고 말할 수 있습니다.

Here, "the length of `nums`" is a value that changes between inputs, and it directly affects the algorithm. The longer `nums` is, the more elements we need to iterate through, and thus the longer our algorithm will take to complete.

여기서 "배열 `nums`의 길이"라는 값은 입력에 따라 변하며, 알고리즘에 직접적인 영향을 줍니다. `nums`가 길어질수록 우리가 순회해야 할 요소가 많아지고, 따라서 알고리즘이 완료되는 데 더 오랜 시간이 걸리게 됩니다.

Note that choosing the letter n to denote this value is arbitrary. There is no requirement that we use n , it's just that n is the commonly accepted standard that everyone uses. If you wanted, you could use a banana (🍌) to represent the length of `nums`. The programmer is the one who decides which variables represent what values.

변수 이름으로 n 을 사용하는 것은 임의적입니다. 반드시 n 을 사용할 필요는 없으며, 단지 n 이 보편적으로 널리 사

용되는 표준일 뿐입니다. 원한다면, 배열 `nums`의 길이를 바나나()로 표현해도 됩니다. 어떤 변수가 어떤 값을 나타내는지는 프로그래머가 결정하는 것입니다.

In the context of LeetCode, there are some common assumptions that we make. When dealing with integers, the larger the integer, the more time operations like addition, multiplication, or printing will take. While this **is** relevant in theory, we typically ignore this fact because the difference is practically very small, and treat all integers the same. If you are given an array of integers as an input, the only variable you would use is n to denote the length of the array. Technically, you *could* introduce another variable, let's say k which denotes the average value of the integers in the array. However, nobody does this.

리트코드(LeetCode)의 맥락에서는 일반적으로 몇 가지 가정을 합니다. 예를 들어, 정수를 다룰 때 정수 값이 커질수록 덧셈, 곱셈 또는 출력과 같은 연산에 시간이 더 걸립니다. 이 사실이 **이론적으로는** 맞지만, 현실에서는 그 차이가 매우 작기 때문에 보통 무시하고 모든 정수를 동일하게 취급합니다. 만약 입력이 정수 배열이라면, 사용할 유일한 변수는 배열의 길이를 나타내는 n 입니다. 기술적으로는 배열에 있는 정수의 평균값을 나타내는 k 라는 또 다른 변수를 도입할 수도 있겠지만, 아무도 그렇게 하지 않습니다.

When written, the function is wrapped by a capital O. Here are some example complexities: 복잡도를 표기할 때는 함수가 대문자 O로 둘러싸여 표현됩니다. 다음은 복잡도의 예시들입니다:

- $O(n)$
- $O(n^2)$
- $O(2^n)$
- $O(\log n)$
- $O(n \cdot m)$

You might be thinking, what is m ? Remember: we define the variables. As these are simply examples with no associated problem, m could denote any arbitrary variable. For example, we could have a problem where the input is two arrays. n could denote the length of one while m denotes the length of the other.

이때 "도대체 m 은 뭔가요?"라고 생각할 수도 있습니다. 기억하세요: 변수를 정의하는 것은 우리입니다. 위의 예시들은 특정 문제와 관련 없이 단지 예시일 뿐이므로, m 은 임의의 변수가 될 수 있습니다. 예를 들어 입력으로 배열이 두 개 주어지는 문제가 있다면, n 은 한 배열의 길이를 나타내고, m 은 다른 배열의 길이를 나타낼 수 있습니다.

These functions represent the complexity. For example, you would say "The time complexity of my algorithm is $O(n)$ " or "The space complexity of my algorithm is $O(n^2)$ ".

이러한 함수들이 복잡도를 표현하는 것입니다. 예를 들어, "내 알고리즘의 시간 복잡도는 $O(n)$ 이야." 혹은 "내 알고리즘의 공간 복잡도는 $O(n^2)$ 이야."라고 말할 수 있습니다.

Calculating complexity

Roughly, your function calculates the number of operations or amount of memory (depending on if you're analyzing time or space complexity, respectively) your algorithm consumes relative to the input size. Using the example from above (find the largest number in `nums`), we have a time complexity of $O(n)$. The algorithm involves iterating over each element in `nums`, so if we define n as the length of `nums`, our algorithm uses approximately n steps. If we pass an array with a length of `10`, it will perform approximately `10` steps. If we pass an array with a length of `10,000,000,000`, it will perform approximately `10,000,000,000` steps.

대략적으로 말하면, 여러분의 함수는 입력 크기에 따라 얼마나 많은 연산(시간 복잡도를 분석할 때)이나 메모리(공간 복잡도를 분석할 때)를 사용하는지를 계산합니다. 예를 들어 앞에서 살펴본 알고리즘(`nums` 배열의 최대값 찾기)을 예로 들어보겠습니다. 만약 배열의 길이를 나타내는 n 이 있다면, 이 알고리즘은 대략 n 개의 단계를 수행합니다. 즉, 배열의 길이가 `10` 이라면 약 `10` 번의 연산을 수행하고, 배열의 길이가 `10,000,000,000` 이라면 약 `10,000,000,000` 번의 연산을 수행하게 됩니다.

Time complexity is not meant to be an **exact** representation of the number of operations. For example, we needed to initialize `maxNum = 0` and we also needed to output `maxNum` at the end. Thus, you could argue that for an array of length `10`, we need `12` operations. This **is not the point** of time complexity. The point of time complexity is to describe how the number of operations changes as the input changes. The number of iterations we do depends on `nums`, but initializing `maxNum = 0` doesn't.

시간 복잡도의 목적은 정확한 연산 횟수를 세는 것이 아닙니다. 예를 들어 길이가 `10` 인 배열에 대해 실제로는 `12` 번의 연산이 필요하다고 주장할 수도 있지만, 이것이 시간 복잡도의 핵심이 아닙니다. 시간 복잡도의 핵심은 입력 크기가 바뀔 때 연산의 수가 **어떻게 변하는지**를 설명하는 것입니다. 즉, 정확한 연산 횟수보다 입력 크기 변화에 따른 연산 수의 증가 양상이 중요합니다.

Being able to analyze an algorithm and calculate its time and space complexity is a crucial skill.

Interviewers will **almost always** ask you for your algorithm's complexity to check that you actually understand your algorithm and didn't just memorize/copy the code. Being able to analyze an algorithm also enables you to determine what parts of it can be improved.

알고리즘의 복잡도를 분석하는 능력은 매우 중요합니다. 특히 리트코드 같은 환경에서는, 복잡도 분석 능력이 코드를 단순히 암기한 것이 아니라 실제로 이해하고 있는지를 보여주는 지표가 됩니다. 또한 알고리즘을 분석할 수 있다면, 어떤 부분이 개선 가능한지 알 수 있습니다.

Rules

There are a few rules when it comes to calculating complexity. First, **we ignore constants**. That means $O(9999999n) = O(8n) = O(n) = O(n/500)$.

복잡도를 계산할 때 몇 가지 규칙이 있습니다. 첫째, **상수(constant)는 무시합니다**. 즉, $O(9999999n) = O(8n) = O(n)$ 입니다.

Why do we do this? Imagine you had two algorithms. Algorithm A uses approximately n operations and algorithm B uses approximately $5n$ operations.

When $n = 100$ algorithm A uses 100 operations and algorithm B uses 500 operations. What happens if we double n ? Then algorithm A uses 200 operations and algorithm B uses 1000 operations. As you can see, when we double the value of n , both algorithms require double the amount of operations. If we were to 10x the value of n , then both algorithms would require 10x more operations.

우리는 왜 이런 분석을 할까요? 두 알고리즘이 있다고 상상해 봅시다. 알고리즘 A는 약 n 번의 연산을 수행하고, 알고리즘 B는 약 $5n$ 번의 연산을 수행합니다.

$n = 100$ 일 때, 알고리즘 A는 100번의 연산을 수행하고 알고리즘 B는 500번의 연산을 수행합니다. 만약 n 을 두 배로 늘리면 어떻게 될까요? 알고리즘 A는 200번의 연산을, 알고리즘 B는 1000번의 연산을 수행합니다. 보시는 것처럼, n 의 값을 두 배로 늘릴 때 두 알고리즘 모두 연산 횟수가 두 배로 늘어납니다. 만약 n 의 값을 10배로 늘린다면, 두 알고리즘 모두 10배 더 많은 연산을 필요로 하게 됩니다.

Remember: the point of complexity is to analyze the algorithm **as the input changes**. We don't care that algorithm B is 5x slower than algorithm A. For both algorithms, as the input size increases, the number of operations required increases **linearly**. That's what we care about. Thus, both algorithms are $O(n)$.

기억하세요: 복잡도를 분석하는 핵심은 알고리즘을 **입력이 변할 때 어떻게 변화하는지** 확인하는 것입니다. 알고리즘 B가 알고리즘 A보다 5배 느리다는 것은 중요한 게 아닙니다. 두 알고리즘 모두 입력 크기가 증가함에 따라 연산의 개수가 **선형적으로** 증가한다는 것이 중요합니다. 이것이 바로 우리가 주목하는 부분입니다. 따라서 두 알고리즘 모두 $O(n)$ 입니다.

The second rule is that we consider the complexity as the variables **tend to infinity**. When we have addition/subtraction between terms of the **same** variable, we ignore all terms except the most powerful one. For example, $O(2^n + n^2 - 500n) = O(2^n)$. Why? Because as n tends to infinity, 2^n becomes so large that the other two terms are effectively zero in comparison.

두 번째 규칙은 복잡도를 고려할 때 변수가 **무한대로 향할 때**의 상황을 생각한다는 것입니다. **같은 변수**의 항들 간에 덧셈 또는 뺄셈이 있을 때는, 가장 영향력이 큰 항을 제외하고는 모두 무시합니다. 예를 들어, $O(2^n + n^2 - 500n) = O(2^n)$ 이 됩니다. 이유가 뭘까요? n 이 무한대로 갈수록 2^n 항이 매우 커지기 때문에, 나머지 두 항은 상대적으로 거의 0에 가까워지기 때문입니다.

Let's say that we had an algorithm that required $n + 500$ operations. It has a time complexity of $O(n)$.

When n is small, let's say $n = 5$, the $+500$ term is very significant - but we don't care about that. We need to perform the analysis as if n is tending toward infinity, and in that scenario, the 500 is nothing. 예를 들어, 어떤 알고리즘이 $n + 500$ 번의 연산을 필요로 한다고 합시다. 이 알고리즘의 시간 복잡도는 $O(n)$ 입니다. n 이 작을 때, 예를 들어 $n = 5$ 라면, $+500$ 이라는 항이 매우 중요한 영향을 미치지만, 우리는 이것에 신경 쓰지 않습니다. 우리는 항상 n 이 무한대로 향한다고 가정하고 분석해야 합니다. 그런 상황에서는 상수인 500은 아무런 의미가 없어지기 때문입니다.

The best complexity possible is $O(1)$, called "constant time" or "constant space". It means that the algorithm ALWAYS uses the same amount of resources, regardless of the input.

가능한 최상의 복잡도는 $O(1)$ 이며, 이를 "상수 시간(constant time)" 또는 "상수 공간(constant space)"이라고 부릅니다. 이는 입력값과 무관하게 알고리즘이 항상 동일한 양의 자원을 사용한다는 의미입니다.

Note that a constant time complexity doesn't necessarily mean that an algorithm is fast ($O(5000000) = O(1)$), it just means that its runtime is independent of the input size.

상수 시간 복잡도라고 해서 반드시 빠른 알고리즘이라는 의미는 아닙니다. 점에 주의하세요($O(5000000)$ 도 결국 $O(1)$ 입니다). 이는 단지 알고리즘의 실행 시간이 입력 크기와 무관하다는 의미일 뿐입니다.

When talking about complexity, there are normally three cases:

- Best case scenario
- Average case
- Worst case scenario

복잡도를 논의할 때는 일반적으로 세 가지 경우를 고려합니다:

- 최선의 경우(Best case scenario)
- 평균적인 경우(Average case)
- 최악의 경우(Worst case scenario)

In most algorithms, all three of these will be equal, but some algorithms will have them differ. If you have to choose only one to represent the algorithm's time or space complexity, never choose the best case scenario. It is most correct to use the worst case scenario, but you should be able to talk about the difference between the cases.

대부분의 알고리즘에서는 이 세 가지 경우가 모두 동일하지만, 일부 알고리즘에서는 서로 다를 수도 있습니다. 알고리즘의 시간이나 공간 복잡도를 표현할 때, 단 하나만 선택해야 한다면 절대로 최선의 경우를 선택하지 마세요. 가장 적절한 선택은 최악의 경우이며, 각 경우들 사이의 차이점에 대해서도 설명할 수 있어야 합니다.

Analyzing time complexity

Let's look at some example algorithms in pseudo-code and talk about their time complexities.

몇 가지 예시 알고리즘을 의사 코드(pseudo-code)로 살펴보고, 이들의 시간 복잡도에 대해 이야기해 보겠습니다.

```
// Given an integer array "arr" with length n
for (int num: arr) {
    print(num);
}
```

This algorithm has a time complexity of $O(n)$. In each for loop iteration, we are performing a print, which costs $O(1)$. The for loop iterates n times, which gives a time complexity of $O(1 \cdot n) = O(n)$.

이 알고리즘의 시간 복잡도는 $O(n)$ 입니다. 각 반복(iteration)에서 하나의 숫자를 출력하는데, 이는 매번 $O(1)$ 의 시간이 걸립니다. 배열의 길이가 n 이므로 반복문은 정확히 n 번 실행되며, 따라서 총 시간 복잡도는 $O(n)$ 이 됩니다.

```
// Given an integer array "arr" with length n
for (int num: arr) {
    for (int i = 0; i < 500000; i++) {
        print(num);
    }
}
```

This algorithm has a time complexity of $O(n)$. In each inner for loop iteration, we are performing a print, which costs $O(1)$. This for loop iterates 500,000 times, which means each outer for loop iteration costs $O(500000) = O(1)$. The outer for loop iterates n times, which gives a time complexity of $O(n)$.

이 알고리즘의 시간 복잡도는 $O(n)$ 입니다. 내부 반복문은 각 요소마다 정확히 500,000번 실행되지만, 이는 입력 크기 (n)와 무관한 상수이기 때문에 $O(1)$ 로 간주합니다. 외부 반복문은 n 번 실행되므로 전체 시간 복잡도는 $O(n)$ 이 됩니다.

Even though the first two algorithms technically have the same time complexity, in reality the second algorithm is much slower than the first one. It's correct to say that the time complexity is $O(n)$, but it's important to be able to discuss the differences between practicality and theory.

첫 번째 알고리즘과 이 두 번째 알고리즘은 기술적으로 같은 시간 복잡도($O(n)$)를 갖습니다. 하지만 실제 실행 속도는

두 번째 알고리즘이 훨씬 느립니다. 시간 복잡도는 정확한 실행 속도를 나타내는 것이 아니라, 입력 크기가 변함에 따라 연산 횟수가 어떻게 증가하는지 설명하는 데 목적이 있기 때문입니다.

```
// Given an integer array "arr" with length n
for (int num: arr) {
    for (int num2: arr) {
        print(num * num2);
    }
}
```

This algorithm has a time complexity of $O(n^2)$. In each inner for loop iteration, we are performing a multiplication and a print, which both cost $O(1)$. The inner for loop runs n times, which means each outer for loop iteration costs $O(n)$. The outer for loop runs n times, which gives a time complexity of $O(n \cdot n) = O(n^2)$.

이 알고리즘의 시간 복잡도는 $O(n^2)$ 입니다. 내부 반복문에서는 곱셈과 출력을 수행하는데, 이 연산들은 각각 $O(1)$ 의 시간이 걸립니다. 내부 반복문은 정확히 n 번 실행되므로, 외부 반복문의 각 반복당 $O(n)$ 의 시간이 소요됩니다. 외부 반복문이 n 번 반복되므로, 총 시간 복잡도는 $O(n^2)$ 이 됩니다.

```
// Given integer arrays "arr" with length n and "arr2" with length m
for (int num: arr) {
    print(num);
}

for (int num: arr) {
    print(num);
}

for (int num: arr2) {
    print(num);
}
```

This algorithm has a time complexity of $O(n + m)$. The first two for loops both cost $O(n)$, whereas the final for loop costs $O(m)$. This gives a time complexity of $O(2n + m) = O(n + m)$.

이 알고리즘의 시간 복잡도는 $O(n + m)$ 입니다.

첫 번째 배열(arr)을 두 번 반복하지만, 상수 배수는 무시하므로 $O(2n + m)$ 에서 최종적으로 $O(n + m)$ 으로 단순화됩니다.

```
// Given an integer array "arr" with length n
for (int i = 0; i < arr.length; i++) {
    for (int j = i; j < arr.length; j++) {
        print(arr[i] + arr[j]);
    }
}
```

This algorithm has a time complexity of $O(n^2)$. The inner for loop is dependent on what iteration the outer for loop is on. The first time the inner for loop runs, it runs n times. The second time, it runs $(n - 1)$ times, then $(n - 2)$, $(n - 3)$ and so on.

이 알고리즘의 시간 복잡도는 $O(n^2)$ 입니다. 안쪽 반복문은 바깥 반복문의 현재 값에 따라 실행 횟수가 달라집니다. 안쪽 반복문은 첫 번째 실행 시 n 번, 두 번째는 $(n - 1)$ 번, 다음엔 $(n - 2)$ 번, $(n - 3)$ 번, ... 이런 식으로 줄어들며 실행됩니다.

That means the total iterations is $1 + 2 + 3 + \dots + n$, which is the partial sum of this series: $n(n + 1) / 2 = (n^2 + n) / 2$. In big O, this is $O(n^2)$ because the addition term in the numerator and the constant term in the denominator are both ignored.

즉, 총 반복 횟수는 $1 + 2 + 3 + \dots + n$ 이 되며, 이는 등차수열의 합 공식에 의해 $n(n + 1) / 2 = (n^2 + n) / 2$ 로 표현됩니다. Big O 표기법에서는 분자에서 덧셈항과 분모의 상수 항을 무시하기 때문에 이 알고리즘의 시간 복잡도는 최종적으로 $O(n^2)$ 이 됩니다.

Logarithmic time

A logarithm is the inverse operation to exponents. The time complexity $O(\log n)$ is called logarithmic time and is **extremely** fast. A common time complexity is $O(n \cdot \log n)$, which is reasonably fast for

most problems and also the time complexity of efficient sorting algorithms.

로그(logarithm)는 지수(exponent)의 역 연산(inverse operation)입니다. 시간 복잡도 $O(\log n)$ 을 로그 시간(logarithmic time)이라고 부르며, **매우 빠른** 속도를 나타냅니다. 또 흔히 등장하는 시간 복잡도로는 $O(n \cdot \log n)$ 이 있는데, 이는 대부분의 문제를 풀기에 충분히 빠르며 효율적인 정렬 알고리즘들이 가지는 시간 복잡도이기도 합니다.

Typically, the base of the logarithm will be 2. This means that if your input is size n , then the algorithm will perform x operations, where $2^x = n$. However, the base of the logarithm doesn't actually matter for big O, since all logarithms are related by a constant factor.

일반적으로 로그의 밑(base)은 2를 사용합니다. 이는 입력 크기가 n 일 때, 알고리즘이 수행하는 연산 횟수가 대략 x 회라 하면, $2^x = n$ 의 관계가 성립함을 의미합니다. 하지만 빅오(Big O) 표기법에서는 로그의 밑(base)이 무엇인지는 중요하지 않습니다. 왜냐하면 모든 로그는 서로 상수배(constant factor)의 차이만 있기 때문입니다.

$O(\log n)$ means that somewhere in your algorithm, the input is being reduced by a percentage at every step. A good example of this is binary search, which is a searching algorithm that runs in $O(\log n)$ time. With binary search, we initially consider the entire input (n elements). After the first step, we only consider $n / 2$ elements. After the second step, we only consider $n / 4$ elements, and so on. At each step, we are reducing our search space by 50%, which gives us a logarithmic time complexity. $O(\log n)$ 은 알고리즘의 과정 중 어딘가에서 매 단계마다 입력 크기가 일정 비율(percentage)씩 줄어드는 것을 의미합니다. 대표적인 예로는 이진 탐색(binary search)이 있는데, 이진 탐색은 시간 복잡도가 $O(\log n)$ 인 탐색 알고리즘입니다. 이진 탐색에서는 처음에 전체 입력(n 개의 요소)을 고려합니다. 첫 번째 단계 이후에는 입력의 절반($n / 2$)만 고려하고, 두 번째 단계 이후에는 그 절반($n / 4$)만 고려합니다. 이렇게 각 단계마다 검색 범위를 50%씩 줄이므로, 로그 시간 복잡도를 갖게 됩니다.

Analyzing space complexity

When you initialize variables like arrays or strings, your algorithm is allocating memory. We never count the space used by the input (it is bad practice to modify the input), and usually don't count the space used by the output (the answer) unless an interviewer asks us to.

배열이나 문자열과 같은 변수를 초기화할 때, 알고리즘이 사용하는 메모리의 양을 분석할 수 있습니다. 공간 복잡도 분석 시, 알고리즘이 사용하는 배열이나 문자열과 같은 메모리 할당량을 측정합니다.

In the below examples, the code is only allocating memory so that we can analyze the space complexity, so we will consider everything we allocate as part of the space complexity (there is no "answer").

아래 예시에서는 공간 복잡도 분석을 위해 변수를 할당하는 것이므로, 할당하는 모든 메모리를 공간 복잡도에 포

함합니다(별도의 "정답"이 없습니다).

```
// Given an integer array "arr" with length n
for (int num: arr) {
    print(num);
}
```

This algorithm has a space complexity of $O(1)$. The only space allocated is an integer variable `num`, which is constant relative to n .

이 알고리즘의 공간 복잡도는 $O(1)$ 입니다. 알고리즘에서 별도의 메모리를 추가로 할당하지 않고, 입력된 배열(`num`) 외에는 고정된 상수 크기의 메모리만 사용하기 때문입니다.

```
// Given an integer array "arr" with length n
Array doubledNums = int[];

for (int num: arr) {
    doubledNums.add(num * 2);
}
```

This algorithm has a space complexity of $O(n)$. The array `doubledNums` stores n integers at the end of the algorithm.

이 알고리즘의 공간 복잡도는 $O(n)$ 입니다. 입력 배열(`arr`)의 길이가 n 일 때, 새롭게 만든 배열(`doubledNums`) 역시 각 원소를 한 번씩 추가하므로 총 n 개의 공간을 추가로 할당하기 때문입니다.

```
// Given an integer array "arr" with length n
Array nums = int[];
int oneHundredth = n / 100;

for (int i = 0; i < oneHundredth; i++) {
    nums.add(arr[i]);
}
```

This algorithm has a space complexity of $O(n)$. The array `nums` stores the first 1% of numbers in `arr`. This gives a space complexity of $O(n/100) = O(n)$.

이 알고리즘의 공간 복잡도는 $O(n)$ 입니다. 배열 `nums`는 입력 배열 `arr`의 앞쪽 1%의 요소를 저장합니다. 이는 정확히는 $O(n/100)$ 이지만, 복잡도 분석에서는 상수 비율은 무시되므로 최종적으로 $O(n)$ 이 됩니다.

```
// Given integer arrays "arr" with length n and "arr2" with length m
Array grid = int[n][m];

for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr2.length; j++) {
        grid[i][j] = arr[i] * arr2[j];
    }
}
```

This algorithm has a space complexity of $O(n \cdot m)$. We are creating a `grid` that has dimensions $n \cdot m$. 이 알고리즘의 공간 복잡도는 $O(n \cdot m)$ 입니다. 여기서는 크기가 $n \cdot m$ 인 2차원 배열 `grid`를 생성하고 있기 때문입니다.