**<Asymptotic Notation(점근적 표기법)>**

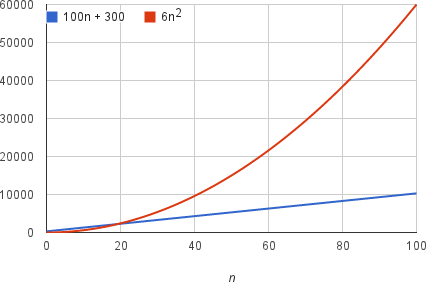
**1. Asymptotic notation**

But what we really want to know is how long these algorithms take.

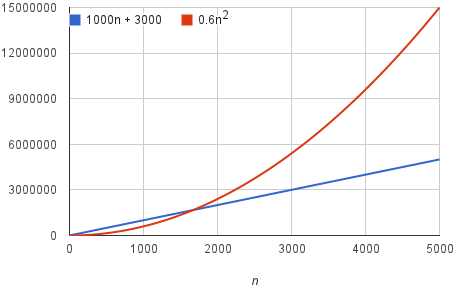
The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm.

Let's think about the running time of an algorithm more carefully. We can use a combination of two ideas. First, we need to determine how long the algorithm takes, in terms of the size of its input.

The second idea is that we must focus on how fast a function grows with the input size. We call this the **rate of growth** of the running time. To keep things manageable, we need to simplify the function to distill the most important part and cast aside the less important parts. For example, suppose that an algorithm, running on an input of size *n*, takes 6n^2 + 100n + 300 machine instructions. The 6n^2 term becomes larger than the remaining terms, 100 n + 300, once *n* becomes large enough, 20 in this case. Here's a chart showing values of 6n^2 and 100n + 300 for values of *n* from 0 to 100:



We would say that the running time of this algorithm grows as n^2, the coefficient 6 and the remaining terms 100n + 300 It doesn't really matter what coefficients we use; as long as the running time is an^2 + bn + c, for some numbers a > 0, is greater than, 0, *b*, and *c*, there will always be a value of *n* for which an^2 is greater than bn + c, and this difference increases as *n* increases. For example, here's a chart showing values of 0.6n^2 and 1000n + 3000 so that we've reduced the coefficient of n^ by a factor of 10 and increased the other two constants by a factor of 10:



The value of *n* at which 0.6n^2 becomes greater than 1000n + 3000 has increased, but there will always be such a crossover point, no matter what the constants.

By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth—without getting mired in details that complicate our understanding. When we drop the constant coefficients and the less significant terms, we use asymptotic notation. We'll see three forms of it: big-Θ notation, big-O notation, and big-Ω notation

**2. Big-θ (Big-Theta) notation**

선형 검색을 구현한 간단한 예를 살펴봅시다:

var doLinearSearch = function(array, targetValue) {

for (var guess = 0; guess < array.length; guess++) {

if (array[guess] === targetValue) {

return guess; // 찾은 경우

}

}

return -1; // 찾지 못한 경우

};

배열의 크기(array.length)를 *n*이라 합니다. for 문은 최대 *n*번 반복될 수 있고, 이런 최악의 경우는 배열에 찾는 값이 존재하지 않을 때 발생합니다.

for 반복문을 수행할 때마다 다음이 실행되어야 합니다:

* guess와 array.length를 비교합니다
* array[guess]와 targetValue를 비교합니다
* 가능하다면 guess의 값을 반환합니다
* guess를 증가시킵니다

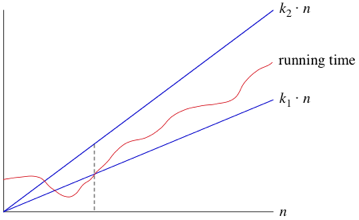
각각의 계산은 실행하는 데에 어떤 상수만큼의 시간이 걸립니다. for 문을 *n*번 반복하면 총 계산에 걸리는 시간은 c1

⋅*n*으로, c1은 반복 한 번에 걸리는 시간을 나타냅니다. 여기서 c\_1의 값은 알 수 없습니다. 컴퓨터, 사용한 언어, 소스 프로그램을 실행 가능한 코드로 바꾸는 컴파일러나 인터프리터 같은 것들에 영향을 받기 때문입니다.

이 코드에는 guess를 0으로 초기화하고, 필요할 때 -1을 반환하는 것처럼, for 문을 만드는 데에도 추가적으로 시간이 필요합니다. 이 추가적인 시간을 c2라고 하겠습니다. 이 또한 상수입니다. 따라서 최악의 경우 선형 검색에 걸리는 시간은 c1⋅n+c2입니다.

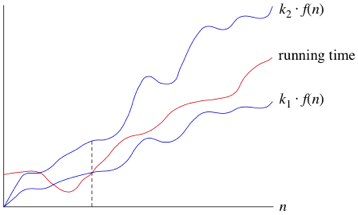
이제까지 설명했듯이 상수 인자인 c1과 c2을 안다고 해서 실행 시간이 커지는 비율을 알 수는 없습니다. 중요한 것은 선형 검색의 최악의 경우의 실행 시간은 배열 크기인 *n*에 따라 커진다는 것입니다. 여기서 실행시간을 표시하기 위해 사용하는 표기법은 Θ(n) 입니다. 이는 그리스어인 "세타"로 "*n*의 빅 세타 " 또는 그냥 "*n*의 세타"라고 읽습니다.

특정 실행 시간이 Θ(n) 이라고 하는 것은 *n*이 충분히 크다면 실행 시간이 어떤 상수 k1와 k2에 대하여 최소 k1⋅n이며 최대 k2⋅n이 된다는 뜻입니다. 아래 그림을 보면 Θ(n)에 대해 이해할 수 있습니다.



*n*의 작은 값에 대해서는 k1⋅n 과 k2⋅n의 실행 시간을 어떻게 다른지는 고려하지 않습니다. 그렇지만 *n*값이 충분히 커지면 점선에서 오른쪽 실행 시간은 반드시 k1⋅n와 k2⋅n 사이에 있게 됩니다. k1과 k2라는 상수가 있다면 실행 시간은 \Theta(n)Θ(n)\Theta, left parenthesis, n, right parenthesis라고 할 수 있습니다.

big-Θ 표기법이 단지 *n*에만 제한되지는 않습니다. n^2이나 nlog\_2같이 *n*에 관한 어떤 함수에서나 이를 이용할 수 있습니다. 아래 그림을 보면 어떤 함수 f(n)에 대하여 실행 시간이 Θ(f(n))이라는 의미가 무엇인지 알 수 있습니다:



*n*값이 충분히 커지면 실행 시간은 k1⋅f(n)과 k2⋅f(n) 사이에 있게 됩니다.

보통 상수 인자와 낮은 차원의 항목은 생략하고 사용합니다. big-Θ 표기법을 사용하는 또다른 이점은 시간 단위를 고려할 필요가 없다는 것입니다. 예를 들어 실행시간이 6n^2 + 100n + 300μs라고 가정해 봅시다. 아니면 ms일 수도 있을 니다. big-Θ 표기법에서는 이를 언급하지 않습니다. 또한 계수인 6과 저차원 항목인 100n + 300을 생략하고 그냥 실행 시간이 Θ(n^2)라고 할 수 있습니다.

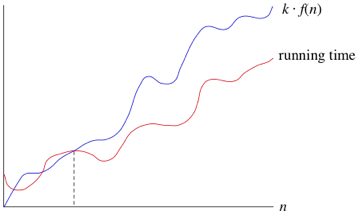
big-Θ 표기법을 사용하는 것은 실행 시간에 대해 **점근적으로 근접한 한계값(asymptotically tight bound)**이 있다고 표현하는 것입니다. "점근적으로"라는 말을 쓰는 이유는 큰 값의 *n*에 대해서만 적용되기 때문입니다. "근접한 한계값"이라는 말은 위, 아래로 상수값 내에서 실행 시간을 좁힐 수 있다는 뜻입니다.

**3. Big-O notation**

Big-Θ 표기법은 실행 시간에 대하여 위아래에 점근적으로 근접한 한계가 있습니다. 하지만 한계를 위에만 둘 때도 있습니다. 예를 들어 이진 검색 실행 시간 최악의 경우는 Θ(log2n)이지만, 이진 검색이 모든 경우에 Θ(log2n)이라고 할 수는 없습니다. 찾고자 하는 값을 첫 번째 추측에 찾으면 어떻게 될까요? 그러면 그 경우 실행 시간은 Θ(1) 입니다. 이진 검색의 실행 시간은 절대 Θ(log2n) 이상이진 않지만 더 좋을 때도 있습니다.

"실행 시간은 최대한 이만큼 커지지만 더 천천히 커질 수도 있다"는 의미인 점근적 표기법 형태를 사용하는 것이 편리할 수도 있습니다. 이런 경우를 위해 "big-O" 표기법을 사용합니다.

실행 시간이 O(f(n))이라면 충분히 큰 값인 n와 상수 k에 대해 실행 시간은 최대 k⋅f(n)가 됩니다. 실행 시간이 O(f(n)) 인 경우에 대해 이렇게 생각할 수 있습니다:



여기서는 실행 시간이 "f(n)의 big-O"거나, 그냥 "f(n)의 O"라고 표현합니다. **점근적 상한선(asymptotic upper bounds)**에 대해서는 big-O 표기법을 사용하는데 이는 충분히 큰 입력 크기에 대하여 실행 시간에 상한값을 두고 제한하기 때문입니다.

이제 모든 경우에 대해 이진 검색의 실행 시간을 알아낼 수 있는 방법이 생겼습니다. 이진 검색의 실행 시간은 항상 O(log2n) 라고 할 수 있습니다. 최악의 경우의 실행 시간에 대해는 더욱 상세한 식을 만들 수도 있습니다. 바로 Θ(log2n)이죠. 그렇지만 모든 경우를 포괄하는 일반적 표현으로는 이진 검색이 O(log2n) 시간 내에 실행된다고 하는 것이 가장 상세한 표현입니다.

big-Θ의 정의를 다시 보면 실행 시간에서 상한과 하한 둘 다 존재한다는 것을 제외하고 big-O 표기법과 상당히 비슷하다는 것을 알 수 있을 것입니다. 특정 상황에서 실행 시간이 Θ(f(n))이라면, 이는 또한 O(f(n))이기도 합니다. 예를 들어 이진 검색의 최악의 실행 시간은 Θ(log2n)이기 때문에 O(log2n)이라고도 할 수 있습니다.

그 반대가 항상 참이 되지는 않습니다. 앞에서도 봤듯이, 이진 검색이 항상 O(log2n) 시간 안에 실행된다고 할 수는 있지만, 항상 Θ(log2n) 시간에 실행되는 것은 아닙니다.

big-O 표기법이 점근적 상한선만 제공하고 점근적으로 근접한 한계를 주지 않기 때문에, 처음에는 틀린 것 같지만 따져보면 맞는 상황들이 있습니다. 예를 들면, 이진 검색의 실행 시간이 O(n)이라고 하는 것은 옳습니다. 실행 시간이 n에 상수를 곱한 것보다 느리게 커지기는 하기 때문입니다.

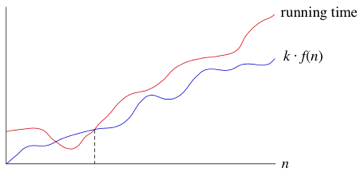
이렇게 생각해 보세요. 10달러를 가지고 있는데, 친구에게 가서 "내 주머니에 돈이 있는데 확실히 100만 달러보다는 적게 있어" 라고 말한다면, 정확하다고는 할 수 없지만 명백한 사실이긴 합니다.

100만 달러가 10달러의 상한선인 것처럼, O(n)도 이진 검색 실행 시간의 상한선입니다. 다른 정확하지 않은 상한선에는 O(n^2), O(n^3), O(2^n) 등이 있을 수 있습니다. 하지만 Θ(n), Θ(n^2), Θ(n^3), Θ(2^n)은 이진 검색의 어떤 경우도 정확히 묘사하지 않습니다.

**4. Big-Ω (Big-Omega) notation**

때로는 알고리즘이 상한선 없이 **최소한** 어느 정도 걸린다고 해야 할 때도 있을 것입니다. 그럴 때는 big-Ω 표기법을 사용합니다. 여기서 Ω는 그리스 문자 "오메가"입니다.

실행 시간이 Ω(f(n)) 라면 n이 충분히 클 때 실행 시간은 어떤 상수 k에 대해 최소 k⋅f(n)입니다. 실행 시간이 Ω(f(n))인 경우는 다음과 같이 생각할 수 있습니다.



여기서 실행 시간은 "f(n)의 big-Ω"라고 합니다. 점근적 하한선을 표현하기 위해선 big-Ω 표기법을 사용하는데, 그 이유는 점근적 하한선은 충분히 큰 입력 크기에서 실행 시간을 밑에서 제한하기 때문입니다.

Θ(f(n))이 자동적으로 O(f(n))을 의미하는 것처럼, 자동적으로 Ω(f(n))도 의미합니다. 따라서 이진 검색 실행 시간 최악의 경우는 Ω(log2n)이라고 할 수 있습니다.

맞긴 하지만 정확하지는 않은 big-Ω 표기법의 예를 들면, 주머니에 백만 달러가 있을 때, 주머니에 돈이 있긴 한데 적어도 10 달러는 된다고 말하는 것이 진실인 것처럼 이진 검색은 최소한 상수의 시간이 걸리므로 이진 검색 최악의 경우 실행 시간은 Ω(1)라고 말할 수 있습니다. 적어도 상수 시간 이상이 걸리는 것을 알기 때문입니다.

당연히 알고리즘을 다룰 때에는 가장 정확한 방법을 사용하는 것이 좋습니다. 여기서 위와 같은 예를 든 이유는 big- Ω, big-O, big-Θ의 이해를 돕기 위한 것뿐입니다.