2장. 알고리즘 분석

알고리즘은 컴퓨터의 자원을 사용하여 실행해야 하는 프로그램의 전체적인 윤곽을 통제하는 일련의 단계적인 명령들의 집합이라고 정의할 수 있다. 실행은 R, 파이썬 그리고 자바 등 어떤 프로그래밍 언어라도 가능하다. 데이터는 모든 프로그램에 있어서 복잡한 구성요소이며, 데이터가 어떻게 구성되어 있는지(데이터 구조)에 따라 실행 시간은 매우 달라질 수 있다. 데이터 구조가 좋은 알고리즘 구현의 핵심인 이유가 여기에 있다. 이 책에서는 실행 시간 또는 시간 복잡성에 주로 집중할 것이며, 프로그램 실행 중 메모리 사용과 실행 시간과의 관계도 부분적으로 다룰 것이다. 2장에서는 다음 주제를 상세하게 다룬다.

* 최선, 최악, 그리고 평균의 경우
* 컴퓨터 대 알고리즘
* 알고리즘 점근 분석(asymptotic analysis)
  + 상한(upper bounds) 추정 = 빅 오 표기법(Big O notation)
  + 하한(lower bounds) 추정 = 빅 오메가 표기법(Big Ω notation)
  + 빅 세타 표기법 (Big Θ notation)
  + 단순화 규칙
  + 분류 규칙
* 프로그램의 성능 추정
* 문제 분석
* 시스템 공간 한계
* 경험적 분석

(## 분류 규칙 : 원서의 이 부분에서는 Classifying functions 이지만 아래쪽 해당 절 본문에서는 Classifying rules 임. ‘분류 규칙’으로 용어 통일함.)

<대> 데이터 구조로 시작하기

데이터 구조는 알고리즘의 핵심적인 부분이다. 자세한 설명을 하기 전에, 한 가지 예를 생각해보자. 사용자가 입력한 유한한 길이를 가진 양의 정수를 정렬하여 오름차순으로 출력하는 알고리즘을 프로그래밍해야 한다. 사용자가 정의한 입력과 사용자가 원하는 출력 사이의 연결 고리 역할을 하는 정렬 알고리즘은 다양한 방법으로 접근할 수 있다.

* 버블 정렬(bubble sort)과 셸 정렬(shell sort). 정렬의 단순한 형태이지만 매우 비효율적이다.
* 삽입 정렬(insertion sort)과 선택 정렬(selection sort). 주로 작은 데이터셋 정렬에 사용된다.
* 병합 정렬(merge sort), 힙 정렬(heap sort), 퀵 정렬(quick sort). 평균 시스템 런타임 시간 복잡성(빅 세타 표기법)에 기반한 효율적인 정렬 방법이다.
* 계수 정렬(counting sort), 버킷 정렬(bucket sort), 기수 정렬(radix sort) 등의 분산 정렬(distributed sort). 런타임과 메모리 사용량 모두 다룰 수 있다.

이 각각의 방법은, 결과적으로, 특정 인스턴스의 집합을 더 효과적으로 처리할 수 있다. 이렇게 말하면 좋은 알고리즘에 대한 개념을 흐리게 된다. 알고리즘은 다른 많은 특성 중에 다음과 같은 것을 가지고 있어야 좋은 알고리즘이라고 할 수 있다.

* 짧은 실행 시간
* 더 적은 메모리 사용량
* 읽기 쉬운 코드
* 입력값에 대한 일반성

일반적으로 한 문제는 다양한 알고리즘을 통해 접근할 수 있으며, 각각의 알고리즘은 다음과 같은 매개변수를 기초로 평가될 수 있다.

* 시스템 런타임
* 메모리 요구사항

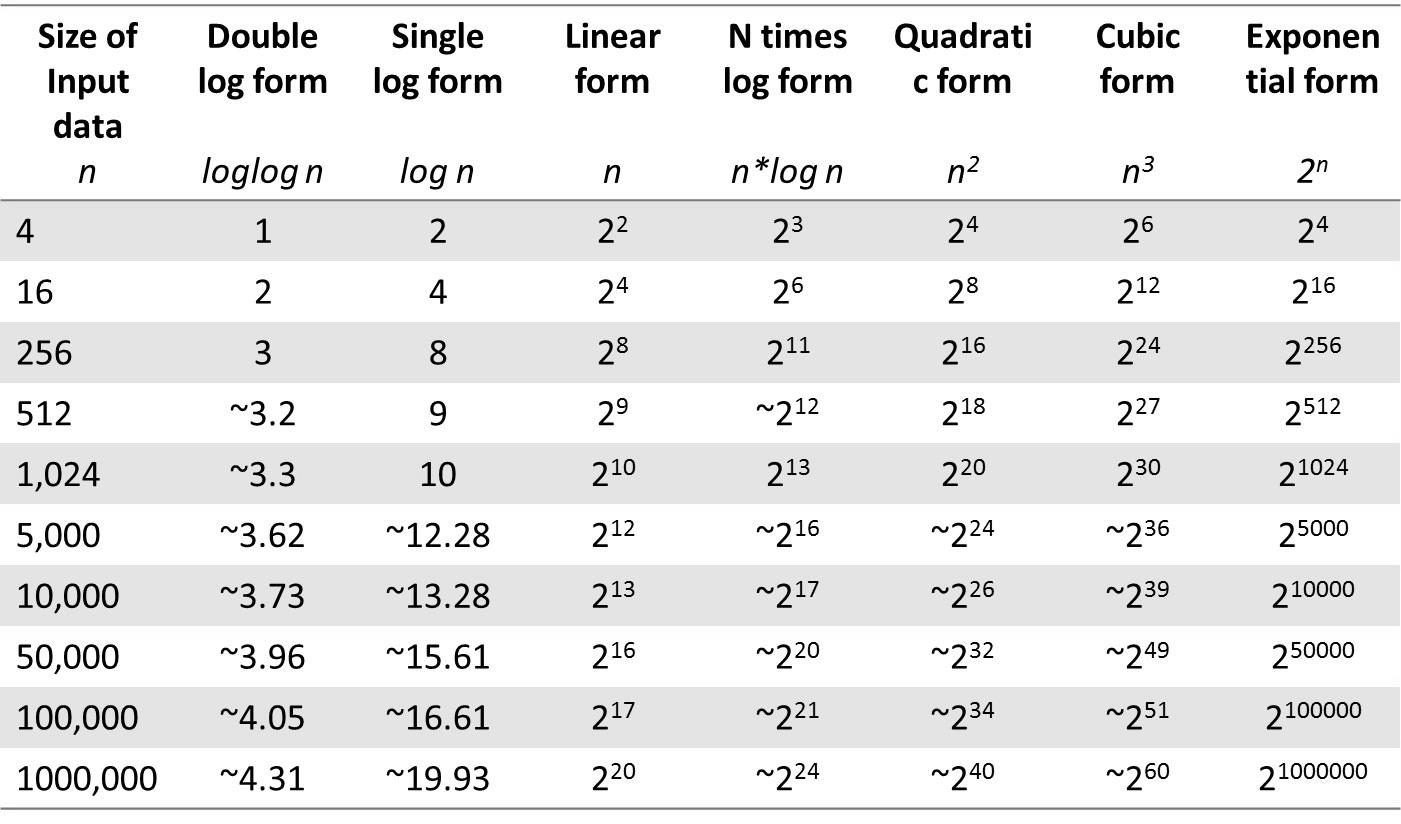
하지만, 이 매개변수들은 일반적으로 다음과 같은 외부 환경 요인에 의해 영향을 받는다.

* 데이터 구조에 대한 처리
* 시스템의 소프트웨어 및 하드웨어 구성
* 코드를 작성하고 컴파일하는 스타일
* 프로그래밍 언어

모든 외부적인 요인을 제어하는 것은 거의 불가능하기 때문에 성능 비교를 위해 다양한 알고리즘의 시스템 런타임을 예측하는 것(이상적 시나리오 분석)은 어렵다. 이상적 시나리오 분석을 위해서는 알고리즘을 구현하고 실행시켜야 하기 때문이다. 그래서 알고리즘을 설계할 때는 알고리즘 성능을 추정하기 위해 점근 분석을 사용한다.

점근 분석은 전체 프로그램을 실제적으로 작성하고 컴파일하는 과정 없이 알고리즘의 효율성을 평가하는 방법이다. 점근 분석은 입력 데이터의 크기와 연산 작업의 갯수를 기초로 가상의 시스템 런타임을 나타내는 함수식이다. 이 함수식은 입력 데이터의 증가율은 시스템 런타임과 정비례한다는 원리를 바탕으로 한다. 예를 들어, 삽입 정렬의 경우 크기는 입력 벡터의 길이를 나타내고, 작업의 수는 정렬 연산의 복잡성을 나타낸다. 점근 분석은 알고리즘의 장점과 단점을 비교하기 보다는 알고리즘을 구현하는데 들어가는 수고의 양을 추정하기 위해 사용된다. 다음 표는 널리 사용되는 증가 함수를 보여준다. 더 자세한 내용은 뒷부분에서 설명한다.

(## 알고리즘 성능 공식에 대해 다양하게 사용되고 있는 growth rate functional forms / growth rate / growth functional form / growth rate function 등의 표현은 이후에 모두 ‘증가 함수’로 통일)



<그림시작>

입력 데이터의 크기

이중 로그 함수

로그 함수

선형 함수

n배 로그 함수

2차 함수

3차 함수

지수 함수

<그림끝>

<그림 2.1: 복잡성 추정을 위해 사용되는 다양한 증가 함수>

일반적으로 사용되는 증가 함수는 입력 데이터의 크기를 기반으로 하며, 알고리즘의 성능을 분석하는 데 사용된다. 또한 알고리즘의 시스템 런타임 추정을 위한 가상의 함수로도 사용된다.

<대> R에서의 메모리 관리

메모리 관리는 일차적으로 사용 가능한 메모리에 대한 관리와 함수를 더 유연하고 빠르게 실행하기 위해 요구되는 추가적인 메모리에 대한 예측을 포함한다. 여기서는 R 환경에서 객체의 저장과 관련된 메모리 할당의 개념을 다룰 것이다.

메모리 할당에 있어서 R은 객체에 따라 서로 다르게 메모리를 할당한다. 메모리 할당은 pryr 패키지의 object\_size 함수를 사용하여 확인할 수 있다. pryr 패키지는 install.packages("pryr") 명령으로 CRAN 저장소로부터 설치할 수 있다. 이 패키지는 R 버전 3.1.0 이상에서만 사용 가능하다. pryr 패키지의 object\_size 함수는 R 기본 패키지의 object.size 함수와 비슷하다. 하지만 다음과 같은 점에서 더 정확하다.

* 현재 객체와 연관된 환경의 크기도 포함
* 주어진 객체 내의 공유된 요소까지 고려됨

다음은 R에서 메모리 할당량을 보기 위해 object\_size 함수를 사용한 예제이다.

> object\_size(1) # 한 개의 숫자형 벡터에 할당된 메모리

48 B

> object\_size("R") # 한 개의 문자형 벡터에 할당된 메모리

96 B

> object\_size(TRUE) # 한 개의 논리형 벡터에 할당된 메모리

48 B

> object\_size(1i) # 한 개의 복소수형 벡터에 할당된 메모리

56 B

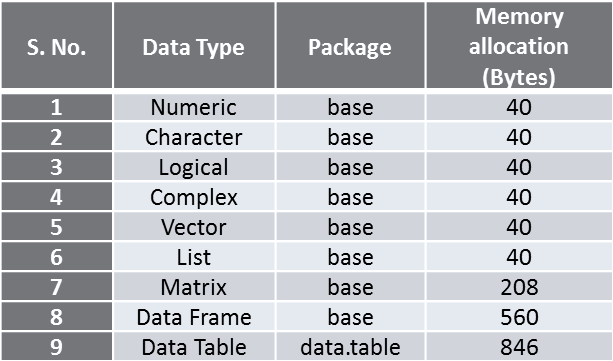
객체에 요구되는 저장 공간은 다음과 같은 요인들과 관련이 있을 수 있다.

* 메타데이터(metadata) : 객체의 메타데이터는 문자형, 정수형, 논리형 등과 같이 객체 타입에 따라 정의된다. 객체 타입은 디버깅할 때도 매우 유용하다.
* 노드 포인터(node pointer) : 노드 포인터는 서로 다른 노드들 간의 연결을 유지하며, 사용되는 노드 포인터의 갯수에 따라 메모리 필요량도 변한다. 예를 들어, 이중 링크드 리스트는 이전 노드와 다음 노드를 연결하는 두 개의 노드 포인터를 사용하기 때문에 단순 링크드 리스트보다 메모리가 더 많이 필요하다.
* 속성 포인터(attribute pointer) : 속성에 대한 참조를 유지하기 위한 포인터이다. 특히 변수에 의해 저장된 데이터에 대한 메모리 할당을 줄이는데 도움이 된다.
* 메모리 할당 : 현재 사용중인 시스템 공간을 나타내는 벡터의 길이
* 크기 : 벡터의 길이만큼 실제로 할당된 시스템 공간의 크기
* 메모리 패딩(memory padding) : 패딩은 구조체의 각 요소에 적용된다. 예를 들어, 각 요소는 8 바이트 경계 이후에 시작된다.

\* 역자주 : 대부분의 컴파일러는 구조체의 각 요소를 메모리에 위치시킬 때 성능향상을 위해 CPU가 접근하기 쉬운 단위로 끊어서 배치한다. 64비트 CPU는 한번에 8바이트(=64비트)를 한 블럭으로 읽는다. 그래서 메모리 할당 후 메모리 블록의 맨 뒤에 남는 빈 공간을 패딩비트(padding bits)로 채우고, 다음 메모리 블럭에 그 다음 요소를 할당한다.

object\_size() 명령은 다음 그림에 보이는 것처럼 내재된 메모리 할당량을 보기 위해 사용된다.

(## 본문에는 table 로 되어 있고 도표에는 Figure로 되어 있는 경우에는 번호 순서 때문에 모두 ‘그림’으로 통일)



<그림시작>

데이터 타입

패키지

메모리 할당량 (바이트)

<그림끝>

<그림 2.2: R의 다양한 데이터 타입이 초기화 될 때 할당되는 메모리>

앞에서 각 데이터 구조/타입에 할당되는 메모리 크기를 확인했다. 이제 정수, 문자열, 불리언, 그리고 복소수 같은 여러 데이터 타입을 가진 벡터의 길이를 늘려가는 시나리오를 시뮬레이션 해보자. 시뮬레이션은 다음과 같이 벡터의 길이를 0에서 60까지 늘려가며 수행된다.

> vec\_length <- 0:60

> num\_vec\_size <- sapply(vec\_length, function(x) object\_size(seq(x)))

> char\_vec\_size <- sapply(vec\_length, function(x) object\_size(rep("a",x)))

> log\_vec\_size <- sapply(vec\_length, function(x) object\_size(rep(TRUE,x)))

> comp\_vec\_size <- sapply(vec\_length, function(x) object\_size(rep("2i",x)))

num\_vec\_size는 요소를 0부터 60개까지 가지고 있는 각 숫자형 벡터에 필요한 메모리 크기를 저장한 변수이다. 이 숫자형 벡터의 요소들은 함수에 선언한 것처럼 순차적으로 증가하는 정수이다. 마찬가지로 문자열 벡터, 논리형 벡터, 복소수형 벡터의 증가하는 메모리 필요량을 계산하여 char\_vec\_size, log\_vec\_size, 그리고 comp\_vec\_size에 저장한다. 이 시뮬레이션을 통해 얻은 결과는 다음 코드를 사용하여 시각화할 수 있다.

> par(mfrow = c(2,2))

> plot(num\_vec\_size ~ vec\_length, xlab = "Numeric seq vector",

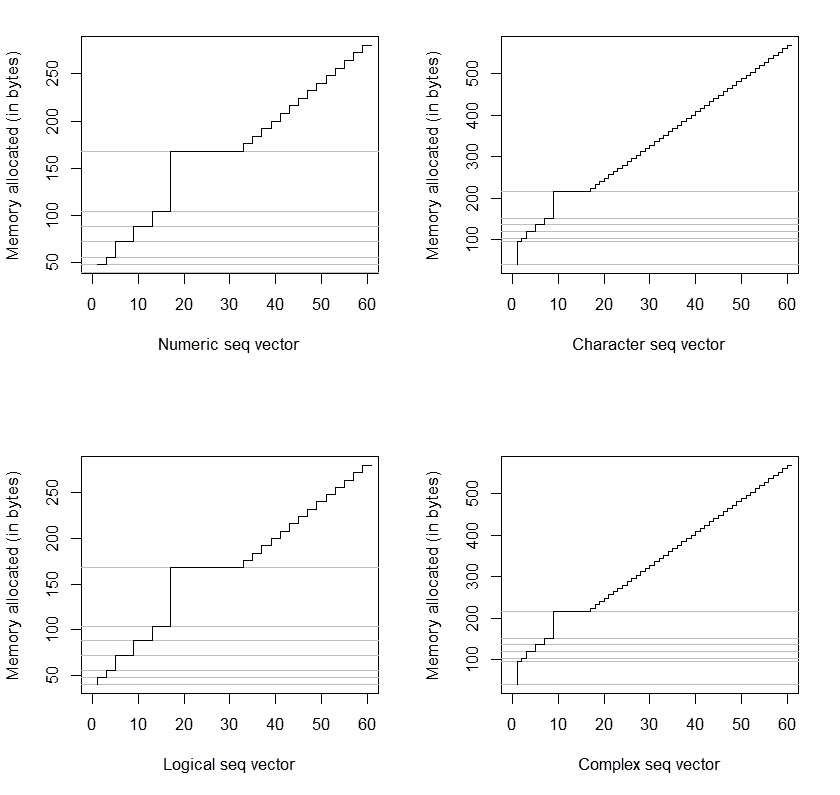
+ ylab = "Memory allocated (in bytes)", type = "n")

> abline(h = (c(0,8,16,32,48,64,128)+40), col = "grey")

> lines(num\_vec\_size, type = "S")

\* 역자주 : 위 코드는 생략된 부분이 있으므로 전체 코드를 다운로드 받아서 실행해 볼 것을 권장한다.

앞의 코드를 실행한 결과는 그림 2.3과 같다. 벡터에 할당되는 메모리는 벡터의 길이와 사용되는 객체 타입에 따르는 함수임을 알 수 있다. 그런데 그 관계가 선형적으로 보이지 않고 계단식으로 증가되는 것처럼 보인다. 이것은 더 좋고 일관된 성능을 제공하기 위해 R이 처음에 RAM에서 큰 블럭의 메모리를 할당하고 내부적으로 관리하기 때문이다. 이 메모리 블럭은 벡터의 데이터 타입과 그 내부의 요소 갯수를 기반으로 벡터에 개별적으로 할당된다. 초기에 특정 수준(숫자형/논리형 벡터는 128 바이트, 문자형/복소수형 벡터는 176 바이트)까지는 메모리 블럭이 불규칙적으로 증가하지만, 그 이후로는 8 바이트의 작은 크기로 증가하며 안정되는 것을 볼 수 있다.



<그림시작>

할당된 메모리 (bytes)

숫자형 벡터

할당된 메모리 (bytes)

문자형 벡터

할당된 메모리 (bytes)

논리형 벡터

할당된 메모리 (bytes)

복소수형 벡터

<그림끝>

<그림 2.3: 벡터의 길이에 따른 메모리 할당량>

초기에 할당된 메모리 크기의 차이 때문에 숫자형과 논리형 벡터는 비슷한 메모리 할당 패턴을 보이며, 복소수형 벡터는 문자형 벡터와 비슷하다. 메모리 관리는 알고리즘이 효율적으로 동작하는 데 도움이 되지만, 알고리즘은 프로그램 실행 전에 런타임을 기준으로 평가해야 한다. 다음 절에서는 함수의 런타임을 얻고 그것을 다른 비슷한 함수들과 비교하는 것과 관련된 기본 개념을 이야기할 것이다.

<중> R에서의 시스템 런타임

시스템 런타임은 서로 다른 알고리즘들을 비교하는 데 핵심이다. 이 과정은 여러가지 옵션들을 비교하고 최선의 알고리즘을 선택하는데 도움이 된다. 다양한 알고리즘을 벤치마킹하는 것은 이후 장에서 자세히 다룰 것이다.

CRAN에 있는 microbenchmark 패키지는 어떤 구문, 함수, 코드의 런타임을 밀리세컨드 이하의 단위까지 측정할 때 사용된다. 그 정확성은 system.time() 함수를 대치할 만하다. 또한 모든 측정이 C 코드로 수행되기 때문에 오버헤드를 최소화한다. 다음 방법들은 경과 시간을 측정하는 데 사용된다.

* 윈도우즈 OS의 QueryPerformanceCounter 인터페이스
* 리눅스 OS의 clock\_gettime API
* 맥 OS의 mach\_absolute\_time 함수
* 솔라리스 OS의 gethrtime 함수

이 책에서는 예제로 기본 데이터셋인 mtcars 데이터를 사용한다. 이 데이터는 1974년 Motor Trend US 잡지에서 얻은 것이며, 32개 자동차의 연료 소비량과 10가지 특성을 비교한 내용으로 구성되어 있다. (1973-74년 모델)

이제 한 정수형 속성(carb, 카뷰레터)의 고유한 값에 따라서 특정한 숫자형 속성(mpg, 갤런 당 마일)의 평균을 계산하는 연산을 수행하려고 한다. 이 작업은 aggregate, group\_by, by, split, ddply(plyr), tapply, data.table, dplyr, sqldf 등의 다양한 방법으로 수행할 수 있다. 설명을 위해 다음 네 가지 방법을 사용했다.

* aggregate 함수:

aggregate(mpg~carb, data=mtcars, mean)

* plyr 패키지의 ddply:

ddply(mtcars, .(carb), function(x) mean(x$mpg))

* data.table 형식:

library(data.table)

mtcars\_tb = data.table(mtcars)

mtcars\_tb[, mean(mpg), by=carb]

* group\_by 함수:

library(dplyr)

summarize(group\_by(mtcars, carb), mean(mpg))

위에 제시한 네 가지 방법의 성능을 평가하기 위해 microbenchmark 패키지를 사용한다. 각 방법을 1,000번 반복한 결과를 평가해 보자.

(## 원서 오류 : 100 times)

> library(microbenchmark)

> MB\_res <- microbenchmark(

+ Aggregate\_func = aggregate(mpg~carb, data=mtcars, mean),

+ Ddply\_func = ddply(mtcars, .(carb), function(x) mean(x$mpg)),

+ Data\_table\_func = mtcars\_tb[, mean(mpg), by=carb],

+ Group\_by\_func = summarize(group\_by(mtcars, carb), mean(mpg)),

+ times=1000

+ )

그 결과는 다음과 같다.

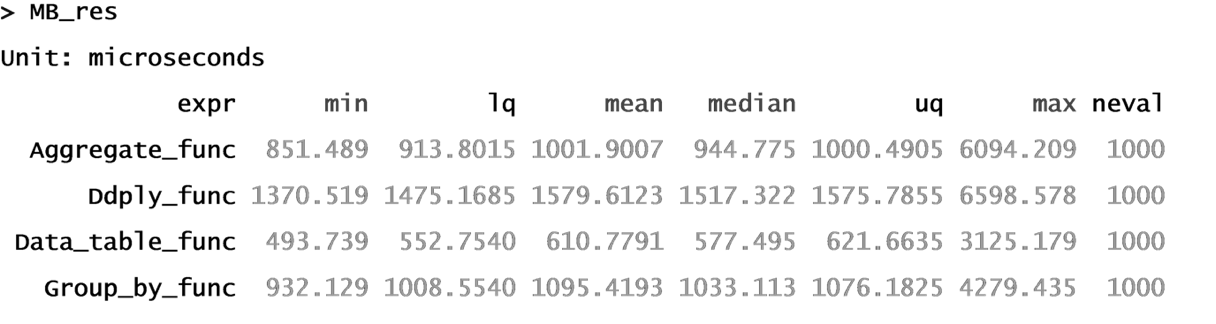
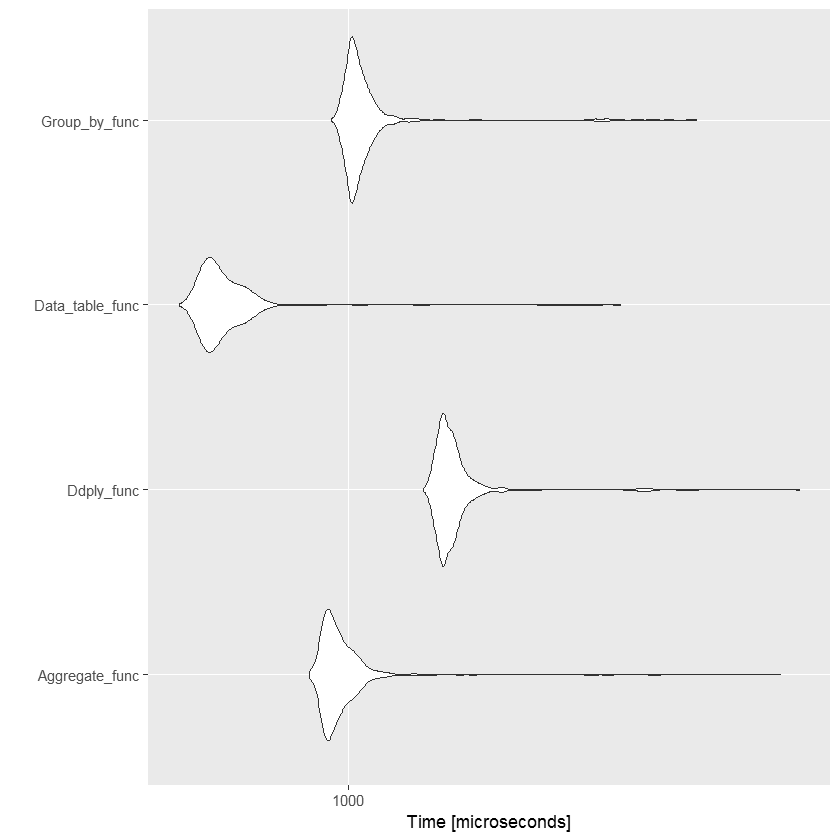


그림 2.4는 각 접근법으로부터 얻은 실행 시간의 분산을 그린 플롯이다.

> library(ggplot2)

> autoplot(MB\_res)



<그림 2.4: 집계 연산의 각 방법을 1,000번 반복했을 때 실행 시간(microseconds)>

주어진 데이터셋에 대한 네 가지 방법 중 data.table이 다른 것에 비해 적은 시간에 효과적으로 수행되었다. 그러나 이 방법들은 대량의 관측치와 많은 수의 속성을 가진 데이터셋이 주어진 시나리오에서 검증해 볼 필요가 있다.

<중> 최선, 최악, 평균적인 경우

시스템 런타임 성능을 기준으로 특정 알고리즘에 대한 코드를 최선, 최악, 또는 평균의 범주로 분류할 수 있다. 자세한 이해를 위해 정렬 알고리즘을 생각해보자. 정렬 알고리즘은 숫자형 벡터를 오름차순으로 배열하기 위해 사용되며, 출력 벡터는 첫번째 요소로 가장 작은 숫자, 마지막 요소는 가장 큰 숫자를 가지고 있어야 하고, 그 가운데 요소들은 순차적으로 증가해야 한다. 5장, 정렬 알고리즘 부분에서 다양한 형태의 정렬 알고리즘을 자세히 다룰 것이지만, 여기서는 삽입 정렬로 구현할 것이다. 삽입 정렬에서 벡터 안의 요소들은 이동 위치를 기준으로 정렬된다. 최선, 최악, 평균의 경우는 데이터에 종속적이다. 이제 삽입 정렬 알고리즘에 대한 최선, 최악, 평균의 경우 시나리오를 정의해보자.

* 최선의 경우 : 최선의 경우는 실행 시간이 최소인 경우이다. 예를 들어, 벡터의 모든 요소들이 이미 오름차순으로 정렬되어 있으면 정렬에 걸리는 시간은 최소가 된다.
* 최악의 경우 : 최악의 경우는 벡터를 완전히 정렬하는데 가능한 최대의 시간이 필요한 경우이다. 예를 들어, 벡터 안의 모든 요소가 거꾸로 내림차순으로 배열되어 있다면 정렬을 하는데 가장 많은 시간이 필요하다.
* 평균의 경우 : 평균의 경우는 벡터의 정렬을 마치는데 중간 정도의 시간이 필요한 경우이다. 예를 들어, 벡터의 요소 중 절반은 오름차순으로 되어 있고 나머지 절반은 내림차순으로 되어 있는 경우이다. 평균의 경우는 서로 다르게 배열되어 있는 요소들을 가진 많은 벡터들을 사용하여 평가된다.

일반적으로 최선의 경우 시나리오는 알고리즘을 가장 낙관적으로 평가하기 때문에 알고리즘을 벤치마크 할 때 고려되지 않는다. 그러나 만약 최선의 경우가 발생할 확률이 높다면 최선의 경우 시나리오를 사용하여 알고리즘을 비교할 수 있다. 최선의 경우와 반대로 최악의 경우 시나리오는 알고리즘을 가장 비관적으로 평가한다. 이것은 주로 철도 네트워크 통제, 항공 교통 통제와 같은 실시간 애플리케이션에 쓰이는 알고리즘을 벤치마크 할 때 사용된다. 종종 입력 데이터의 분포를 잘 알 수 없는 경우가 있는데, 그럴 때는 최악의 경우 시나리오를 기초로 알고리즘 성능을 평가하는 것이 안전하다.

일반적으로 평균의 경우 시나리오가 알고리즘 성능의 대표적인 척도로 사용된다. 하지만 이것은 입력 데이터의 분포를 알고 있는 경우에만 유효하다. 평균의 경우 시나리오는 입력 데이터의 분포가 왜곡되어 있으면 알고리즘을 적절하게 평가하지 못할 수 있다. 정렬 작업에 있어서, 만약 대부분의 입력 벡터가 내림차순으로 정렬되어 있다면 평균의 경우 시나리오는 알고리즘을 평가하기 위한 최선이 아닐 수도 있다.

간단히 말해서 입력 데이터 분포와 함께 실시간 애플리케이션 시나리오는 최선, 최악, 평균의 경우를 기반으로 알고리즘을 분석하기 위한 주요 기준이다.

<중> 컴퓨터 대 알고리즘

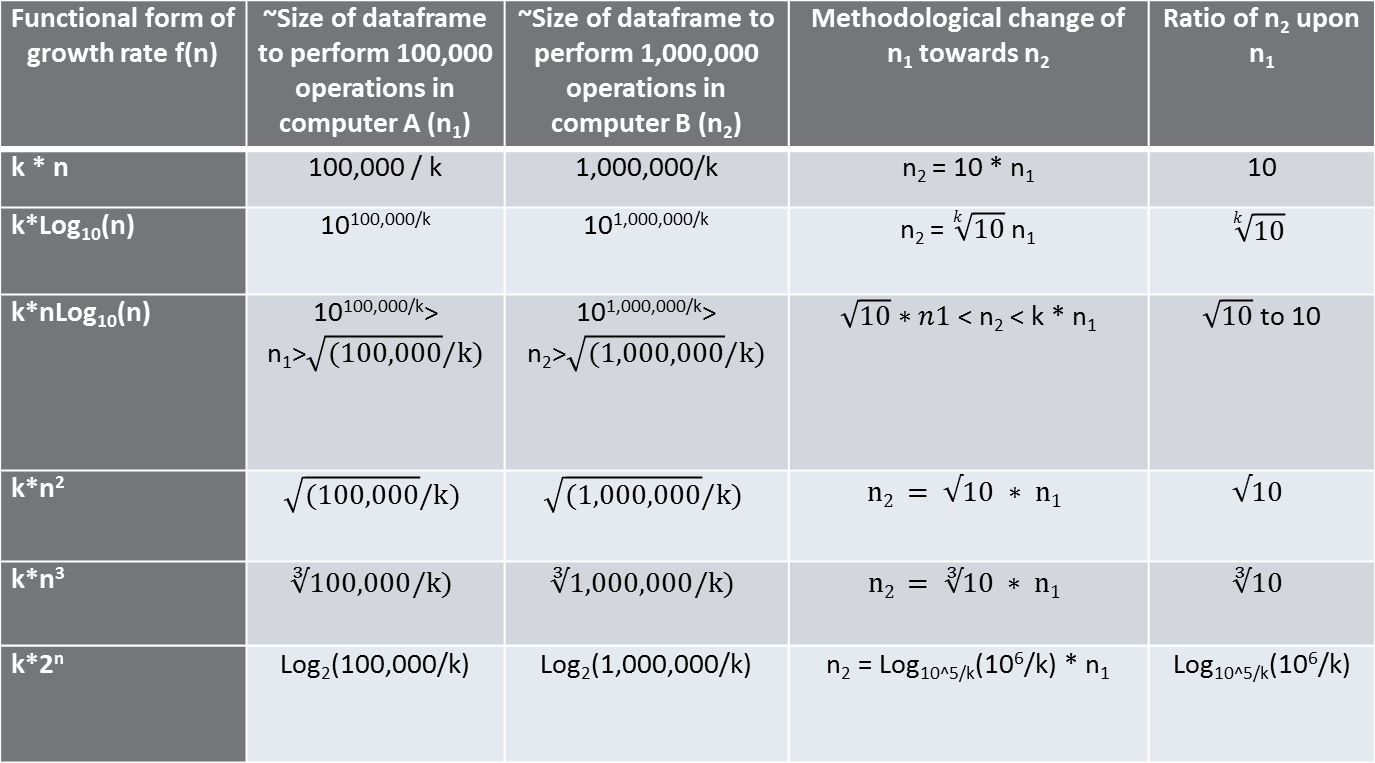
이 절에서는 주로 컴퓨터 구성과 알고리즘 런타임 간의 상호 절충 관계를 자세히 다룬다. 두 대의 컴퓨터 A와 B가 있고, B는 A보다 10배 빠르다. 100,000개의 관측치가 있는 데이터 프레임에 대한 알고리즘의 시스템 런타임이 컴퓨터 A에서는 약 60분이다. 알고리즘의 시스템 런타임 함수식은 n3이다. 그런데 이 함수식은 알고리즘의 실행을 완료하기 위해 필요한 연산 작업 수의 증가와 동등하다고 할 수 있다. 즉, 시스템 런타임의 함수식과 연산 작업 증가율은 같다. 다음 상황들은 이 상호 절충 관계를 이해하는데 도움이 될 것이다.

상황 1: 컴퓨터 A보다 10배 빠른 컴퓨터 B는 알고리즘의 시스템 런타임을 60분에서 6분으로 줄일 수 있을까?

컴퓨터 A와 B의 데이터셋 크기가 동일하다면 그 대답은 아마도 “예”일 것이다. 하지만 데이터 프레임의 크기를 10배 늘리면 다음과 같은 상황이 발생한다.

상황 2: 컴퓨터 A보다 10배 빠른 컴퓨터 B에서 1,000,000개로 관측치가 늘어난 데이터 프레임을 알고리즘이 60분 안에 처리할 수 있을까?

컴퓨터 구성의 변화 뿐만 아니라 입력 데이터의 크기 변화도 다루어야 하고, 이 경우는 알고리즘이 비선형적으로 (여기서는 세제곱 형태) 수행되기 때문에 까다로워진다. 다음 표는 주어진 알고리즘의 증가 함수에 대해 늘어난 입력 데이터 프레임을 정해진 시간 내에 처리하는, 컴퓨터 A보다 10배 빠른 컴퓨터 B의 가용성을 설명한다. 60분 동안 컴퓨터 A가 100,000번의 연산을 처리할 수 있고, 반면에 컴퓨터 B는 1,000,000번의 연산을 수행할 수 있다고 가정한다. k는 양의 실수인 상수 ~ 컴퓨터 A와 B에서 동일한 x분의 시간.



<그림시작>

증가 함수 f(n)

~컴퓨터 A에서 10만 번의 연산에 사용되는 데이터 프레임의 크기(n1)

~컴퓨터 B에서 100만 번의 연산에 사용되는 데이터 프레임의 크기(n2)

n1에서 n2로 방법론 변경

n1에 대한 n2의 비율

<그림끝>

<그림 2.5: 크기가 다른 데이터 프레임을 사용한 일반적인 증가 함수의 성능 비교>

각 알고리즘 증가 함수에 대해 알아보자.

* 선형 : 그림 2.5를 보면, 상수 k에 상관없이 컴퓨터 B는 동일한 60분 동안 10 배 더 큰 입력 데이터 프레임을 처리할 수 있다. 즉, 선형 런타임 함수를 가진 알고리즘의 처리 속도는 상수 k에 독립적이다. 입력 데이터의 절대적인 크기가 런타임에 영향을 미친다. 또한 런타임이 고정된 값으로 주어진 경우, 한 시스템이 다른 시스템보다 i배 빠르다면 빠른 시스템의 데이터 처리 가용성은 느린 시스템보다 i배 더 높다. 그러므로 두 컴퓨터의 상대적인 성능은 알고리즘의 증가율 상수 k에 독립적이다.
* 제곱, 세제곱 : 상수 k에 대해 컴퓨터 B는 60분의 동일한 시간 내에 입력 데이터 프레임을 10의 제곱근(3.16), 그리고 10의 세제곱근(2.15) 배 처리할 수 있다. 여기서도 역시 컴퓨터 B의 성능은 상수 k에 영향을 받지 않는다. 다시 말해서, 컴퓨터 A보다 10배 빠른 컴퓨터 B는 주어진 시간 내에 선형 함수의 경우 10배를 처리하는 것에 비해 (제곱근 함수식의 경우 제곱근의 성능인) 3.16배의 데이터만을 처리할 수 있다. 그러므로 i 제곱근(i는 2, 3, 4 등)의 내재적인 특성 때문에 컴퓨터의 성능이 빨라진다고 해도 그에 따르는 입력 데이터 크기에 대한 이익은 높은 비율로 줄어든다.
* 로그 : 이 함수는 보통 두 가지 형태로 사용된다.
  + Log(n) : 입력 데이터 프레임의 크기 증가는 두 가지 요소에 의존적이다. 하나는 시스템의 컴퓨팅 성능의 증가분이고, 다른 하나는 상수 k이다. 그러나 입력 데이터 크기의 증가는 시스템 성능 증가분의 k 제곱근 값에 정비례하기 때문에 시스템의 컴퓨팅 구성과 그 성능 간의 차이는 계속된다.
  + nLog(n) : 시스템의 컴퓨팅 성능 증가에 따른 처리 가능한 입력 데이터 크기의 증가량은 이차 함수를 사용하여 얻은 값보다 크지만 선형함수를 가진 알고리즘보다는 낮다.
* 거듭제곱 (지수) : 거듭제곱 함수에서 알고리즘의 시스템 런타임은 입력 데이터 크기가 증가할 때 지수함수적으로 증가한다. k = 1인 경우, 컴퓨터 A에서 100,000번의 연산을 수행하는 입력 데이터의 크기는 ~ 11이다. 마찬가지로, 컴퓨터 B에서 1,000,000번의 연산을 수행하는 입력 데이터의 크기는 ~ 14이다. 그러므로 n2 = n1 + 3 이라고 할 수 있다. 이것은 시스템의 성능이 10배 좋아져도 동일한 시간 내에 처리할 수 있는 데이터 사이즈의 증가는 미미하다는 것을 분명히 보여준다. 지수 함수 또는 거듭제곱 함수를 가진 알고리즘에 대한 입력 데이터 크기의 증가는 곱셈이라기 보다는 거의 덧셈이라고 할 수 있다. 즉, 컴퓨터 A에서 100,000건의 데이터를 처리할 때 시스템 런타임이 60분이라면, 그보다 10배 빠른 컴퓨터 B에서는 60분 동안 100,003건 크기의 입력 데이터만 처리하는 것이다. 그러므로 지수 함수를 갖는 알고리즘의 성능은 다른 증가 함수와 많이 다르다.

이제 상황 3으로 들어가 알고리즘과 컴퓨터 사이의 상호 절충 관계를 살펴보자.

상황 3 : n3 증가 함수를 가진 알고리즘에 대해서 컴퓨터의 성능 가용성을 높이는 것과 증가 함수 자체가 바뀌도록 알고리즘을 재구성하는 것 중 어떤 것이 더 좋을까?

시스템의 성능을 증가시키는 시나리오는 이미 상황 2에서 살펴봤으니, 알고리즘의 증가 함수를 재구성하는 상황을 분석해보자.

현재, 알고리즘이 가지고 있는 함수식은 n3이다. 입력 데이터의 크기가 1,000일 경우 전체 연산의 수는 1,0003이 된다. nLog10(n) 함수를 갖도록 이 알고리즘을 수정한다고 가정하면, 연산 작업의 수는 1,0003 보다 아주 낮은 3,000으로 줄어들 것이다. n > 2 이라면 n3 함수의 연산 작업 수는 nLog10(n) 함수의 연산 작업 수보다 10배 이상 많아진다. 이것은 컴퓨터의 성능을 10배 늘리는 것보다 알고리즘을 재구성하여 증가 함수를 바꾸는 것이 훨씬 바람직하다는 것을 말해준다.

이 상호 절충 관계를 요약하면

* 느린 증가율을 가진 알고리즘은 대량의 데이터를 처리할 때 컴퓨터의 구성을 업그레이드 하는 것보다 더 좋은 성능을 보여준다.
* 빠른 증가율을 가진 알고리즘은 대량의 데이터를 처리할 때 성능 증가율은 컴퓨터의 가용성을 높이는 것에 비례하지 않을 수도 있다.

<중> 알고리즘 점근 분석

알고리즘은 컴퓨터가 이해할 수 있는 언어로 주어진 문제를 분석하고 계산하기 위해 설계된 단계적인 절차임을 앞에서 배웠다. 알고리즘의 점근 분석(asymptotic analysis)은 필요한 한계 조건(boundary conditions)과 함께 런타임 성능 또는 증가율을 결정하기 위해 사용하는 수학적 표기법이다. 한계 조건은 컴퓨터의 구성, 입력 데이터 크기의 증가, 증가율 함수의 계수(앞의 “컴퓨터 대 알고리즘” 절에서 언급된 상수 k) 등의 요인에 의해 좌우된다. 하지만, 대량의 데이터셋을 처리하기 위한 능력은 증가율 함수식의 상수값보다 컴퓨터의 계산 능력에 좀 더 의존적이다. 또한, 다양한 증가율 함수의 곡선은 그 방정식에서 상수값과 상관없이 교차한다. 그러므로 증가율 또는 시스템 런타임 함수식의 상수는 컴퓨터 수준에서 또는 알고리즘 수준에서 비교할 때 일반적으로 무시된다. 그럼에도 불구하고 다음 상황에서는 상수를 고려하는 것이 바람직하다.

As we learned earlier, an algorithm is a step by step procedure designed to analyze and compute a given problem in a language understandable by a computer. Asymptotic analysis of an algorithm is a mathematical representation to determine its runtime performance or growth rate with the necessary boundary conditions. The boundary conditions depend on factors such as computer configurations, growth in the size of input data, coefficient of the growth rate function (also referred to as constant ( k ) in section Computer versus algorithm ), and others. However, the capability to handle larger data sets is more dependent on the increment in computational performance of computers rather than on the constant term in the growth rate functional form. Also, the curves of different growth rate functional forms do intersect irrespective of the value of the constant in those equations. Thus, the constants in the growth rate or system runtime functional forms are generally ignored while comparing performances at computer level or at the algorithm level. Nevertheless, it is desirable to consider constants in the following situations:

* 데이터 크기가 매우 작은데 알고리즘은 커다란 데이터셋에 최적화되도록 설계된 경우
* 매우 다양한 요인에 의해 상수가 서로 다른 알고리즘을 비교할 필요가 있을 때. 하지만 매우 느린 증가율을 가진 알고리즘은 일반적으로 고려되지 않기 때문에 매우 드물게 발생한다.
* If the data size is very small, and the algorithm is designed optimally for larger datasets.
* If we need to compare algorithms whose constants differ by a very large factor. However, this happens very rarely, since the algorithms with a very slow growth rate are generally not considered.

﻿

점근 분석은 또한 알고리즘의 런타임을 결정하는 입력 크기의 함수이기 때문에 알고리즘의 최선, 최악, 그리고 평균적인 경우를 결정하는데 사용된다. 예를 들어, 정렬 알고리즘의 성능은 입력 벡터의 증가분을 이용하여 평가될 수 있다. 다음은 표준적인 삽입 정렬과 병합 정렬의 점근 함수이다.

Asymptotic analysis is also used to determine the best, worst, and average case of an algorithm, as it is a function of input size which evaluates the runtime of the algorithm. For example, the performance of a sorting algorithm can be evaluated using the incremental length of input vectors. The following are asymptotic functions for standard insertion sorting and merge sorting:

* 표준 삽입 정렬: f(n) = α+ c\*n 2
* 표준 병합 정렬 : f(n) = α + c\*n\*log 2 (n)

두 함수를 다루기에 앞서, α와 c는 상수이며, n은 입력 벡터의 길이를 나타낸다.

In the preceding two functions, α and c are constants and n is the length of the input vector.

ballpark estimation 근사치

여기서 점근 분석은 시스템 런타임 소비 시간 측면에서 알고리즘 성능의 근사값만 제공한다는 것을 명심해야 한다.

One needs to bear in mind that asymptotic analysis provides only a ballpark estimation of the algorithm's performance in terms of system runtime consumption.

다음에 나오는 점근 분석 표기법은 알고리즘의 런타임 계산의 복잡성을 결정하기 위해 일반적으로 사용되는 것이다.

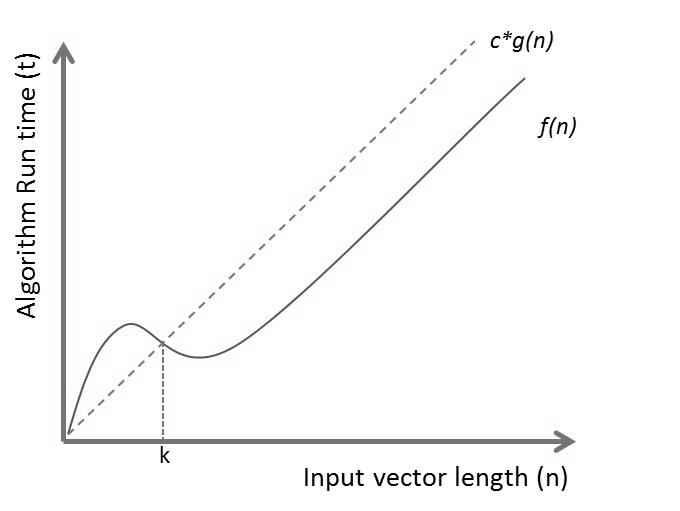
The following asymptotic notations are commonly used to determine the complexity in calculating the runtime of an algorithm.

<소> 상한 경계 또는 빅 오 표기법(Big O notation)

알고리즘 실행 시간의 상한 경계(upper bound)는 빅 오(O)로 표시한다. 이것은 최악의 경우 시나리오를 평가할 때 사용되며, 주어진 입력 벡터 길이에 대해 가장 긴 실행 시간을 결정한다. 즉, 이것은 알고리즘 증가율의 최대값이다.

The upper bound of an algorithm's running time is denoted as O. It is used in evaluating worst-case scenarios, and determines the longest running time for any given length of an input vector. In other words, it is the maximum growth rate of an algorithm.

\* 역자주 : 상한 경계(upper bound)는 찾고자 하는 값보다 큰 값이 처음으로 나타나는 위치를 말하며, 반대로 하한 경계(lower bound)는 찾고자 하는 값보다 작은 값이 처음 나타나는 위치를 말한다.



<그림시작>

알고리즘 런타임(t)

입력 벡터 길이(n)

<그림끝>

<그림 2.6: n>k 일 때 f(n)은 g(n)의 빅오이다.>

Figure 2.6: f(n) is Big O of g(n) for all n>k

다양한 입력 벡터 길이 n에 기초하여 알고리즘의 런타임 t를 결정하는 두 함수 f와 g를 생각해보자. 함수식 f(n)과 g(n)은 입력 벡터의 길이가 증가함에 따라 알고리즘의 실행 시간도 실질적으로 증가하기 때문에 음수이거나 감소하지 않는다. 이 함수식은 주어진 알고리즘의 최선, 최악, 평균의 경우 시나리오에 대한 실행 시간과 동일하다.

Let us consider two functions, f and g, which determine an algorithm's runtime t based on varying input vector length n . These functional forms f(n) and g(n) should be non-negative or non-decreasing, because as the length of the input vector increases, the running time of the algorithm practically increases. These functional forms are equivalent to the running time of best, average, and worst-case scenarios of any given algorithm.

그림을 통해 볼 수 있듯이, 초기에 n<k 인 경우 c\*g(n)은 f(n)보다 낮지만, 그 뒤에 n>k인 경우에는 c\*g(n)이 f(n)보다 높다. 그러므로 알고리즘의 상한 경계는 다음과 같이 나타낼 수 있다.

As we can see, initially c\*g(n) is lower than f(n) for values of n<k, and subsequently, c\*g(n) is higher than f(n) for n>k . Thus, the upper bound of the algorithm can be represented as follows:

f(n) = O(g(n)) – 즉, n>k>0 이고 c>0 이면 f(n) < c\*g(n) 이다.

f(n) = O(g(n)) that is – f(n) < c\*g(n) for n>k>0 and c>0 .

그러므로 모든 가능한 입력 n과 임의의 상수 c에 대해서 g(n)보다 f(n)이 더 빨리 실행될 때만 증가율이 f(n)인 알고리즘을 g(n)의 ‘빅 오’라고 한다.

Therefore an algorithm with a growth rate f(n) is known as Big O of g(n) only when f(n) executes faster than g(n) for all possible inputs n ( n>k>0 ) and any constant c ( c>0 ).

이제 실행 시간이 2차의 다항식 f(n)으로 표현되는 알고리즘을 생각해보고, f(n)의 상한 경계를 나타내는 g(n)을 결정해보자.

Now, let's consider an algorithm whose running time can be expressed as f(n) of polynomial order 2, and we need to determine g(n), which represents the upper bound for f(n) :

f(n) = 25 + 12n + 32n2 + 4\*log(n)

n>0 인 경우:

f(n) < 25n2 + 12n2 + 32n2 + 4n2

f(n) < (25 + 12 + 32 + 4)n2

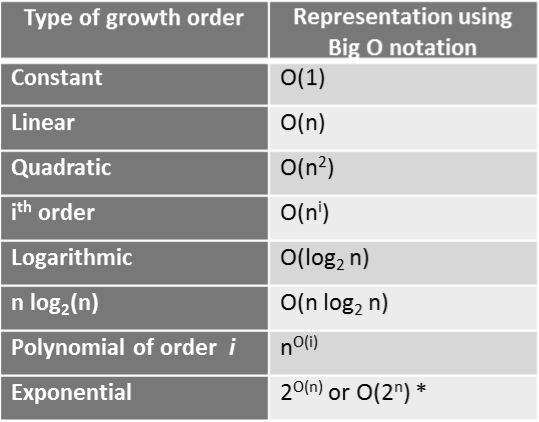
f(n) = O(n2), g(n) is n2ㅇ이고 c=(25+12+32+4)인.

하지만 이 접근법에는 한계가 있다. 선형 함수의 계수가 매우 높으면 실제 시나리오에서는 고차의 다항식 또는 작은 계수의 지수가 선호된다.

However, there exists a limitation with this approach. If the coefficient of the linear function is very high, then a polynomial of higher order or an exponential with a smaller coefficient is preferred in practical scenarios.

다음은 알고리즘의 성능을 평가하기 위해 일반적으로 사용되는 증가율 함수의 차수 유형이다. 다음 그림에서 볼 수 있듯이 2O(n) 과 O(2n)은 서로 다른 결과와 다른 해석을 내 놓는다.

The following are some of the growth orders widely used to assess an algorithm's performance. Both 2O(n) and O(2n) yield different results and different interpretations as shown in the following figure:



<그림시작>

증가 함수 차수의 유형

빅 오 표기법을 사용한 표현

상수

선형

2차

i차

로그

nlog2(n)

i차 다항식

제곱

<그림끝>

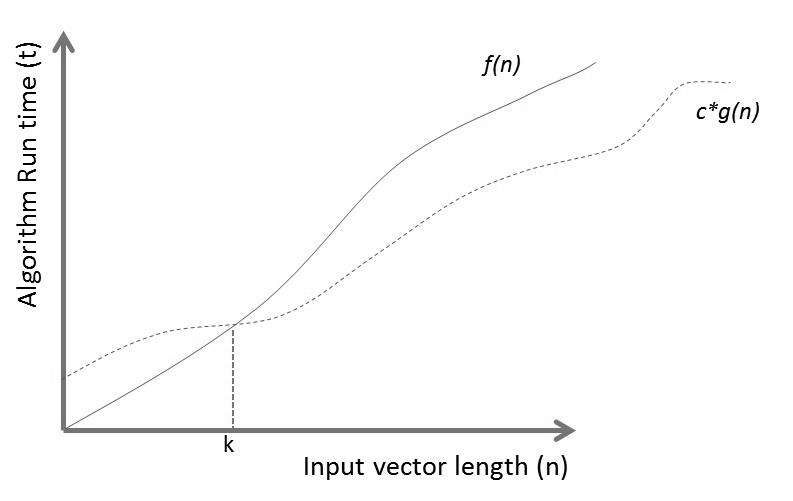
<그림 2.7:>

Figure 2.7: Big O representation of various growth order functions

<소> 하한 경계 또는 빅 오메가 표기법(Big Omega notation)

알고리즘 실행 시간의 하한 경계(lower bound)는 빅 오메가(Ω)로 표시한다. 이것은 알고리즘의 최소 수행 시간 또는 주어진 입력 벡터 길이에 대한 최선의 경우 시나리오를 평가하는데 사용된다. 즉, 알고리즘의 최소 증가율을 말한다.

The lower bound of an algorithm's running time is denoted as Ω. It is used in evaluating the least running time of an algorithm, or the best-case scenario for any given length of input vector. In other words, it is the minimum growth rate of an algorithm.



<그림시작>

알고리즘 런타임(t)

입력 벡터 길이(n)

<그림끝>

<그림 2.8: n>k 일 때 f(n)은 g(n)의 빅 오메가이다.>

Figure 2.8: f(n) is Big- Ω of g(n) for all n>k

다양한 입력 벡터 길이 n에 기초하여 알고리즘의 런타임 t를 결정하는, 음수가 아니고 감소하지 않는 두 함수 f(n)과 g(n)을 생각해보자. 이 함수식은 주어진 알고리즘의 최선, 최악, 평균의 경우 시나리오에 대한 실행 시간과 동일하다.

Let us consider two non-negative and non-decreasing functions f(n) and g(n) , which determine an algorithm's runtime t based on a varying input vector length n . These functional forms are an equivalent to the running time of best, average, and worst-case scenarios of any given algorithm.

그림을 통해 볼 수 있듯이, 초기에 n<k 인 경우 c\*g(n)은 f(n)보다 높지만, 그 뒤에 n>k인 경우에는 c\*g(n)이 f(n)보다 낮아진다. 그러므로 알고리즘의 하한 경계는 다음과 같이 나타낼 수 있다.

As we can see, initially c\*g(n) is higher than f(n) for values of n<k, and subsequently, c\*g(n) becomes lower than f(n) for n>k . Thus, the lower bound of the algorithm can be represented as follows:

f(n) = Ω(g(n)) - 즉, n>k>0 이고 c>0 이면 f(n) > c\*g(n) 이다.

f(n) = Ω(g(n)) that is – f(n) > c\*g(n) for n>k>0 and c>0 .

그러므로 모든 가능한 입력 n (n>k>0)과 임의의 상수 c (c>0)에 대해서 g(n)보다 f(n)이 더 느리게 실행될 때만 증가율이 f(n)인 알고리즘을 g(n)의 ‘빅 오메가’라고 한다.

Hence an algorithm with growth rate f(n) is known as Big O of g(n) only when f(n) executes faster than g(n) for all possible input n ( n>k>0 ) and any constant c ( c>0 ).

(## 문장 오류. 앞부분에서 복사한 것 같습니다.)

이제 실행 시간이 2차의 다항식 f(n)으로 표현되는 알고리즘을 생각해보고, f(n)의 하한 경계를 나타내는 g(n)을 결정해보자.

Now, let's consider an algorithm whose running time can be expressed as f(n) of polynomial order 2, and we need to determine g(n), which represents the lower bound for f(n) :

f(n) = 25 + 12n + 32n2 + 4\*log(n)

n>0 인 경우, 하한 경계의 최대값은 다음과 같다.

Now for every n>0 , the largest of the lower bound is as follows:

f(n) > 25n2

f(n) > Ω(n2), g(n)이 n2이고 c=25 인.

하한 경계의 최소값은 다음과 같다.

The smallest of lower bound is as follows:

f(n) > 25

f(n) > Ω(25)

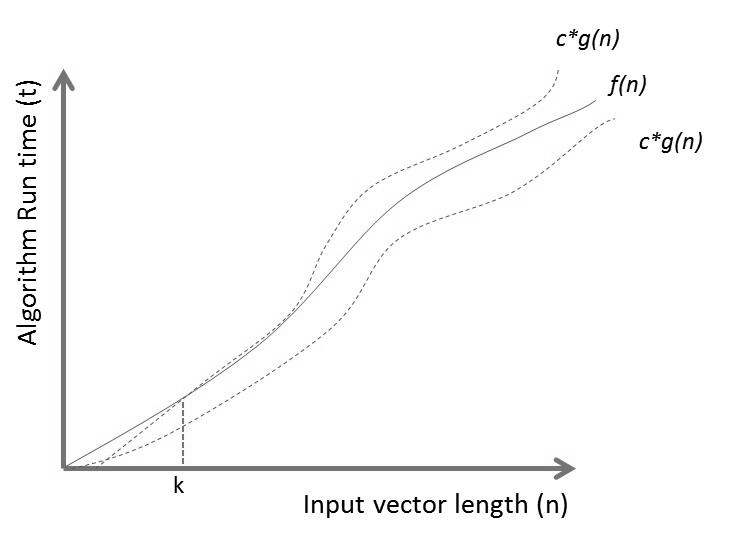
여기서 g(n)은 상수이며 c=25 이다.

Here, g(n) is a constant and c=25 .

<소> 빅 세타 표기법(Big θ notation)

‘빅 오’와 ‘빅 오메가’는 각각 알고리즘 실행 시간의 상한(최대)와 하한(최소) 경계를 설명한다. 세타(θ)는 같은 함수를 이용하여 알고리즘 런타임의 상한과 하한 경계 모두를 결정할 때 사용된다. 다시 말해서 빅 세타 표기법은 실행 시간에 점근적으로 강하게 묶여 있다. ‘점근적으로’라는 말은 오직 대량의 관측치에 대해서만 중요하기 때문이며, ‘강하게 묶여 있다’는 것은 실행 시간이 일정한 상수 요인 범위 내에 있기 때문이다.

As you just learned, about O and Ω, which describe the upper (maximum) and lower (minimum) bound of an algorithm's running time respectively, θ is used to determine both the upper and lower bound of the algorithm's runtime, using the same function. In other words, it is asymptotically tight bound on the running time. Asymptotically because it is significant only for large number of observations, and tight bound because the running time is within constant factor bounds:



<그림시작>

알고리즘 런타임(t)

입력 벡터 길이(n)

<그림끝>

<그림 2.9: n>k 일 때 f(n)은 g(n)의 빅 세타이다.>

Figure 2.9: f(n) is Big- θ of g(n) for all n>k

다양한 입력 벡터 길이 n에 기초하여 알고리즘의 런타임 t를 결정하는, 음수가 아니고 감소하지 않는 두 함수 f(n)과 g(n)을 생각해보자.

Let us consider two non-negative and non-decreasing functions f(n) and g(n) which determine an algorithm's run time t based on varying input vector length n .

n>k>0 이고 c>0 이라면,

Then, for every n>k>0 and c>0 ,

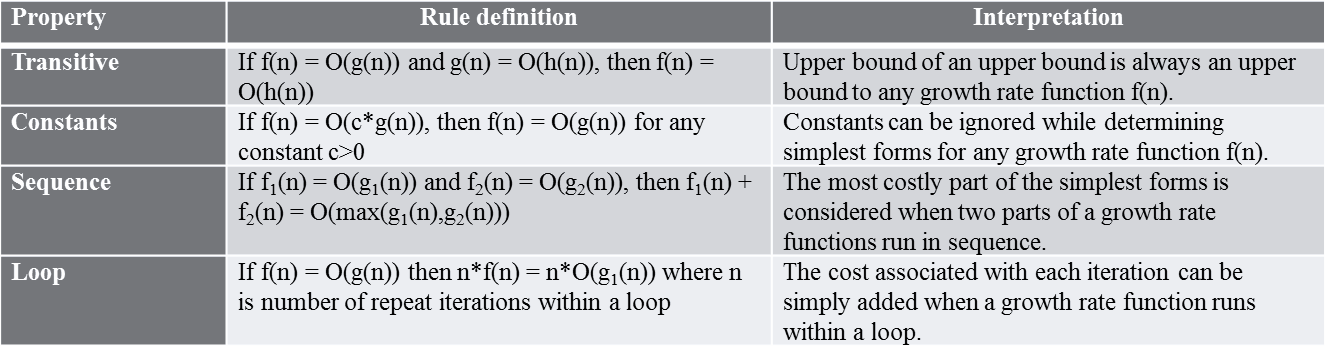
f(n) = θ(g(n)), 오직 O(g(n)) = Ω(g(n)) 인 경우에만.

f(n) = θ(g(n)) if and only if O(g(n)) = Ω(g(n)) .

<소> 단순화 규칙

빅 오(상한 경계), 빅 오메가(하한 경계), 그리고 빅 세타(평균)는 알고리즘의 증가율 또는 시스템 런타임을 나타내는 함수 방정식의 가장 단순한 형식이다. 단순화 규칙은 공식적인 점근 분석에 대한 걱정을 덜기 위해 이 단순한 형식을 사용하는 것이다. 이 규칙은 빅 오, 빅 오메가, 빅 세타 모두에 적용 가능하다. 하지만 아래 표는 빅 오 점근선에 대해서만 표시했다.

Big O (upper bound), Big Omega (lower bound), and Big Theta (average) are the simplest forms of functional equations, which represent an algorithm's growth rate or its system runtime. Simplifying rules can be used to determine these simplest forms without worrying much about formal asymptotic analysis. These rules are applicable to all the three simplest forms. However, the examples shown in the following table are based on the Big O asymptote.



<그림시작>

특성

규칙 정의

해석

이행

f(n) = O(g(n))이고 g(n) = O(h(n))이면 f(n) = O(h(n))

모든 증가 함수 f(n)에 대해 상한 경계의 상한 경계는 항상 상한 경계이다.

상수

f(n) = O(c\*g(n))이면 상수 c>0에 대해 f(n) = O(g(n))

모든 증가 함수 f(n)에 대해 단순 공식을 정할 때 상수는 무시될 수 있다.

순차

f1(n) = O(g1(n))이고 f2(n) = O(g2(n)) 이면 f1(n) + f2(n) = O(max(g1(n), g2(n)))

두 개의 증가 함수가 순차적으로 실행될 때는 단순 공식 중 가장 비용이 높은 부분이 고려되어야 한다.

루프

f(n) = O(g(n))이면 n\*f(n) = n\*O(g(n)). n은 루프 안에서 반복되는 횟수.

증가 함수가 루프 안에서 실행될 때 반복과 관련된 비용은 단순하게 추가될 수 있다.

<그림끝>

<그림 2.10: 단순화 규칙 정의와 해석>

Figure 2.10: Deﬁnition of simplifying rules along with their interpretations

이 단순화 규칙은 이어지는 장들에서 알고리즘의 증가율 또는 시스템 런타임 함수식에 대한 비용을 평가할 때 계속 사용된다.

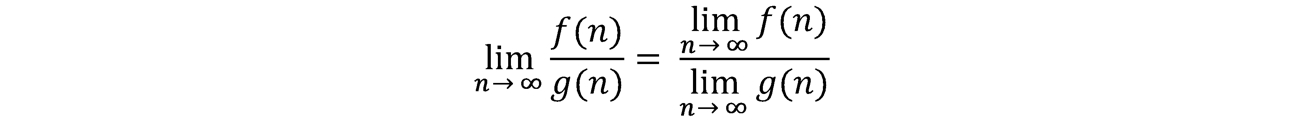
These simplifying rules are widely used in the following chapters while evaluating costs for an algorithm's growth rate or system runtime functional form.

\* 역자주 : 데이터의 크기가 작을 때는 증가율 함수의 모든 요소가 알고리즘의 성능 차이를 나타내는 중요한 요인이 된다. 그러나 데이터 크기가 커질수록 계수와 최고차 항을 제외한 나머지는 의미가 없어진다. 단순화 규칙이란 이런 이유로 점근 표기법에서 최고차 항으로만 간략하게 표시하여 알고리즘의 성능을 비교하기 쉽도록 한 것이다.

<소> 분류 규칙

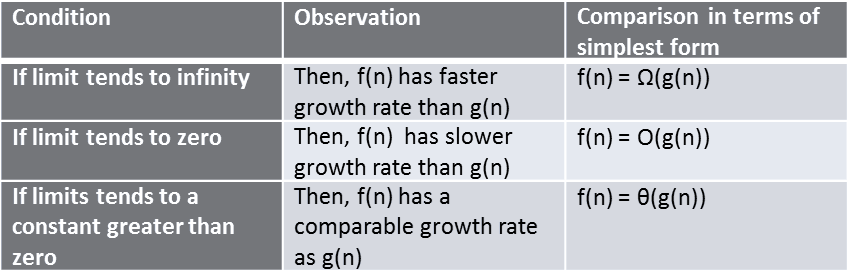
두 선형 증가 함수 f(n)과 g(n)을 생각해보자. 분류 규칙은 어떤 함수가 더 좋은 성능을 가지고 있는지 결정할 때 사용된다. 분류 규칙은 다음과 같이 극한 정리를 이용하여 평가할 수 있다.

Let's consider two algebraic growth rate functions f(n) and g(n) . The classifying rules are then used to determine which functional form has a better performance over the other. This can be evaluated using the limit theorem, which is as follows:



다음은 f(n)과 g(n)을 분류할 때 사용하는 세 가지 시나리오이다.

The following three scenarios are used to classify f(n) and g(n):



<그림시작>

조건

전망

비교를 위한 단순화 공식

극한이 무한대로 향할 경우

그러면, f(n)이 g(n)보다 더 빠른 증가율을 갖는다.

f(n) = Ω(g(n))

극한이 0을 향할 경우

그러면, f(n)이 g(n)보다 느린 증가율을 갖는다.

f(n) = O(g(n))

극한이 0보다 큰 상수를 향할 경우

그러면, f(n)과 g(n)은 비슷한 증가율을 갖는다.

f(n) = θ(g(n))

<그림끝>

<그림 2.11: 분류 규칙>

Figure 2.11: Classifying rule forms

<중> 프로그램의 계산능력 평가

이제 점근 분석을 사용하여 프로그램 또는 알고리즘 내의 여러 요소들의 계산 능력을 평가해 보자.

Let's' evaluate the computations of different components within a program or algorithm using asymptotic analysis.

<소> 요소 1 – 할당 연산자

각 개체(숫자, 문자, 복소수 또는 논리형)를 객체에 할당하는 것은 일정한 시간을 필요로 한다. 개체는 벡터, 데이터 프레임, 매트릭스나 다른 것일 수도 있다.

Assigning an element (numeric, character, complex, or logical) to an object requires a constant amount of time. The element can be a vector, dataframe, matrix and others.

int\_Vector <- 0:60

그러므로 할당 작업의 점근선(빅 세타 표기법)은 θ(1)이다.

Hence, the asymptote (Big Theta notation) of the assignment operation is θ(1) .

<소> 요소 2 – 단순 루프

루프 내에 할당 작업을 하는 단순 for 루프를 보자.

Consider a simple for loop with assignment operations.

a <- 0

for(i in 1:n) a <- a + i

다음은 코드의 각 라인에 대한 점근선이다.

The following are asymptotes for each line of execution in the code:



<그림시작>

코드 라인

시스템 런타임

단순화 규칙

점근선(빅 세타 표기법)

상수

상수

상수 (n번 반복)

루프

<그림끝>

<그림 2.12: 단순 for 루프의 점근 분석>

Figure 2.12: Asymptotic analysis of a simple for loop

그러므로 단순화 규칙을 사용한 이 for 루프의 전체 비용은 θ(n)이다.

Hence, the total cost of this for loop using simplifying rules is θ(n) .

<소> 요소 3 – 복잡한 루프

while 루프와 할당 작업을 가진 중첩 for 루프가 있는 복잡한 루프를 살펴보자.

Consider a complex loop using a while loop, and a nested for loop using assignment operations.

a <- 1

i <- 1

b <- list()

while(i<=n) {

a <- a+i

i <- i+1

}

for(j in 1:i)

for(k in 1:i) {

b[[j]] <- a+j\*k

}

다음은 코드의 각 라인에 대한 점근선이다.

The following are asymptotes for each line of execution in the code:



<그림시작>

코드 라인

시스템 런타임

단순화 규칙

점근선 (빅 세타 표기법)

상수

상수

상수 (n번 반복)

루프

상수 (j와 k 수만큼 반복)

중첩 루프

<그림끝>

<그림 2.13: 복잡한 루프에 대한 점근 분석>

Figure 2.13: Asymptotic analysis of a complex loop

그러므로 단순화 규칙을 사용한 이 루프의 전체 비용은 θ(n2)이다.

<소> 요소 4 – 조건문을 가진 루프

아래 예제 코드처럼 내부에 if…else 조건문을 가진 for 루프를 살펴보자.

Consider a for loop with a nested if...else condition, as shown in the following example code:

a <- 1

for(i in 1:n)

{

if(i <= n/2)

{

for(j in 1:i)

a <- a+i

} else {

a <- a\*i

}

}

다음은 코드의 각 라인에 대한 점근선이다.

The following are the asymptotes for each line of execution in the code:



<그림시작>

코드 라인

시스템 런타임

단순화 규칙

점근선 (빅 세타 표기법)

상수

상수

상수 (n(n+3)/8 번 반복)

if 조건이 true일 때 중첩 루프

상수 (n/2 번 반복)

if 조건이 false일 때 단순 루프

<그림끝>

<그림 2.13: 조건절이 있는 루프에 대한 점근 분석>

Figure 2.14: Asymptotic analysis of a conditional loop

if…else 조건문을 가진 이 루프의 단순화 규칙을 사용한 전체 비용은 θ(n2)이다. if…else 조건문의 비용 평가는 최악의 경우 시나리오를 사용한다. 여기서 최악의 경우 시나리오는 if 조건문이 true일 때이며, else 조건에서의 단순 for 루프 대신에 중첩 for 루프가 실행된다. 그러므로 최대 증가율 (또는 시스템 런타임)은 조건문의 점근선에 대한 평가가 반영되었다.

The total cost of this loop with if...else conditions using simplifying rules is θ(n2) . The cost assessment of an if...else condition is evaluated using the worst-case scenario. Here, the worst-case scenario is when the if condition is True , and the nested for loop is executed instead of a simple for loop in the else condition. Hence, maximum growth rate (or system runtime) is considered for evaluating the asymptote of the conditional statements.

<소> 요소 5 – 재귀 명령문

루프 안에서 조건이 만족될 때까지 동일한 함수를 반복해서 호출하는 것을 재귀 명령문이라고 한다. 가장 일반적으로 사용되는 재귀 명령문은 팩토리얼 함수이다. 다음은 정수 n의 팩토리얼을 계산하는 코드이다.

A statement which iterates in a loop using the same function till a condition is satisfied is called a recursive statement. The most commonly used recursive statement is the factorial function. The following code calculates the factorial of an integer n .

fact\_n <- 1

for(i in 2:n) {

fact\_n <- fact\_n \* i

}

다음은 코드의 각 라인에 대한 점근선이다.

The following are the asymptotes for each line of execution in the code:



<그림시작>

코드 라인

시스템 런타임

단순화 규칙

점근선 (빅 세타 표기법)

상수

상수

상수 (n번 반복)

루프

<그림끝>

<그림 2.15: 재귀 명령문의 점근 분석>

Figure 2.15: Asymptotic analysis of a recursive statement

단순화 규칙을 사용한 이 재귀 명령문의 전체 비용은 θ(n)이다.

The total cost of a recursive statement using simplifying rules is θ(n) .

<중> 문제 분석 Analyzing problems

알고리즘은 문제 분석을 위한 본질적인 기반을 형성하며, 각 문제는 다양한 알고리즘을 통해 분석될 수 있다. 더 나아가, 앞에서 살펴 보았듯이, 알고리즘은 자신의 함수적 성능을 기초로 평가된다. 하지만 알고리즘 수만큼 해결책을 가지고 있는 문제를 어떻게 평가할 것인가 하는 기초적인 질문이 생긴다.

Algorithms form an intrinsic base for analyzing a problem, and each problem can be analyzed using multiple algorithms. These algorithms are further evaluated based on their functional performances, as covered under previous sections. However, there arises a basic question – how to evaluate a problem which has many solutions vis-à-vis many algorithms.

m개의 알고리즘을 가진 (m은 무한대로 수렴한다) 문제를 생각해보자. 상한 경계 또는 최악의 경우 시나리오는 최선의 경우 알고리즘의 상한보다 낮을 수 없고, 하한 경계 또는 최선의 경우 시나리오는 최악의 경우 알고리즘의 하한보다 높을 수 없다. 즉, 한 알고리즘의 하한과 상한 경계를 정의하는 것이 더 쉽지만, 전혀 고려해보지 않은 알고리즘이 있을 수 있기 때문에 문제에 대해 정의할 때 곤란해진다.

Consider a problem with m number of algorithms, where m tends to infinity. Then, the upper bound or the worst-case scenario cannot be lower than the upper bound of the best algorithm, and the lower bound or the best-case scenario cannot be higher than the lower bound of the worst algorithm. In other words, it is easier to define the lower and upper bounds for an algorithm, but it becomes tricky when it is to be defined for a problem, since there might be algorithms which might not have been explored at all.

이 문제는 이어지는 장들에서 예제와 함께 더 상세히 다룰 것이다.

More details along with examples will be covered in subsequent chapters.

<중> 공간 한계 Space bounds

지금까지 오직 알고리즘의 시스템 런타임 함수를 이용하여 알고리즘 성능을 평가했다. 시스템 공간 또는 가용 메모리는 알고리즘 개발자에게 또다른 중요 제약사항이 될 수 있다. 런타임 증가 함수가 주로 입력 데이터 구조의 크기에 의존적이라면, 공간 증가 함수는 데이터 구조의 유형과 크기 모두에 의존적이다. 예를 들면, k 바이트 크기의 요소 n 개를 가진 벡터는 k\*n (θ(n)) 바이트의 메모리가 필요하다. 각 데이터 구조에 요구되는 시스템 공간은 효율적인 데이터 액세스를 위한 데이터 저장 모드에 따라 다르다.

So far, the performance of an algorithm was evaluated using only its functional form of system runtime. Another functional form can be a key constraint for algorithm developers in system space or available memory. The space functional form depends on both the type and size of data structure unlike the runtime functional form, which depends primarily on the size of the input data structure. As an example, a vector of n elements requires k\*n (θ(n)) bytes of memory provided that each element requires k bytes. The space required by each data structure depends on the mode of data storage for efficient data access within.

예를 들면, 링크드 리스트는 리스트의 요소들 뿐만 아니라 내부적인 탐색을 쉽게 하기 위한 포인터도 저장하고 있다. 포인터는 유지비 같은 부가적인 요소이므로 추가적인 저장 공간 할당이 필요하다. 그러므로 유지비가 적은 데이터 구조는 공간 증가 함수 관점에서 알고리즘의 성능을 향상시킬 수 있다.

For example, a linked list not only stores a list of elements but also pointers for easy navigation within. These pointers are additional storage elements, which act as overheads and require additional space allocation. Thus, a data structure with lower overheads can enhance the performance of algorithms in terms of space functional form.

그러나 한 알고리즘을 효과적으로 평가하기 위해서는 시스템 런타임과 요구되는 저장 공간 사이에 상호 절충관계가 있어야 한다. 최고의 알고리즘은 적은 공간과 적은 런타임을 가진 것이다. 하지만 현실적으로 알고리즘 개발자가 두 가지 모두를 만족시키는 것은 매우 어렵다. 필요한 저장 공간을 줄이기 위해서 개발자는 데이터 정보를 인코딩하는 경향이 있다. 그러나 이것은 디코딩하기 위해 추가적인 시간이 필요하기 때문에 결과적으로 시스템 런타임을 증가시킨다. 그와 반대로, 개발자는 시스템 런타임을 줄이기 위해 저장 공간을 더 많이 소비하여 알고리즘이 실행되는 동안 데이터 저장 정보를 재구성하기도 한다. 이와 관련된 자세한 내용은 이후의 장에서 다룰 것이다.

However, there needs to be a trade-off between the system's runtime and space requirement for effective evaluation of an algorithm. The best algorithm is one which requires less space and less runtime. But in reality, satisfying both criteria is difficult for algorithm developers. In order to reduce space requirement, developers tend to encode data information, which, in turn, requires additional time to decode, thereby increasing the system runtime. On the other hand, developers tend to restructure data storage information while executing algorithms to decrease the system runtime at the expense of greater space. More details along with examples will be covered in subsequent chapters.

<대> 연습문제

1. 다음은 몇 가지 증가 함수이다. 성능이 가장 느린 것에서 빠른 것 순으로 정렬할 수 있는가?

1. The following are some growth-rate functional forms. Can you arrange them in the order of slower to faster performance?

10n3

3(logen)2

10n

100n

Log2n2

Log2n3

Log3n2

Log3n3

n1.5

2. 다음 질문에 답하라.

\*

\* How can we evaluate the total memory currently being used by a given R environment? What is the purpose of garbage collection (GC) in the context of R?

\* Which occupies more size – a matrix with 10 numbers of categorical attributes, or a dataframe with 10 numbers of corresponding factors?

\* Can you evaluate and plot the memory allocation for dataframes and matrices with an increment of five observations for a fixed number of attributes (15 columns)?

\* Why does data.table occupy more memory than data.frame ?

3. Is data.table scalable in terms of performance (faster execution of operations) related to data pre-processing and transformations?

(Hint: microbenchmark using large number of variables and observations with a higher number of iterations for each scenario).

4. What are the best, worst, and average-case scenarios for the factorial n(n!)?

5. Consider two computing systems A and B, where B is 100 times faster than A. Suppose an algorithm requires 100,000 iterations in system A in a given time t . The following are the functional forms which represent system runtime:

10nlog2n

5n3

8log3n2

Calculate the following:

\* Time required by system B to complete 100,000 iterations

\* Number of iterations processed by system B in the given time t

6. Determine the relationship between the following functional forms f(n) and g(n) based on the asymptotic analysis using suitable limits for the input size n.

f(n) = nlog n ; g(n) = n2log n

f(n) = n2 ; g(n) = 2n

f(n) = 25 ; g(n) = 210

f(n) = 2n ; g(n) = 3n

f(n) = nlog n ; g(n) = (log n)2

7. 다음 코드의 빅 세타 값을 구하라:

(1)

for(i in 1:100)

{

a = i\*10

b = a+50

}

(2)

i=1; a=0

while(i<100) {

a = c(a,i)

i=i+1

}

(3)

a = data.frame(i=0, j=0)

for(i in 1:100)

{

for(j in 1:100)

{

a[i,1] = i

a[j,2] = j

}

}

(4)

a=50

for(i in 1:100)

{

if(i <= a)

print("i is less than or equal to a")

else

print("i is greater than a")

}

(## 원서 코드 오류 수정함.)

<대> 요약

2장에서는 R에서 알고리즘 평가의 기본 개념과 차이점을 간략하게 살펴봤다. R에서 메모리 관리 및 시스템 런타임에 대한 개념도 다루었다. 알고리즘의 성능을 추정하는 최선, 최악, 그리고 평균의 경우 시나리오에 대해서도 논의했다. 추가적으로 컴퓨터 성능과 알고리즘의 시스템 런타임 간의 상호 절충관계, 알고리즘 점근 분석, 단순화 규칙과 분류 규칙, 그리고 프로그램의 계산 능력 추정에 대한 내용을 이야기 했다. 3장에서는 R의 기본적인 데이터 구조와 리스트의 개념에 대해서 다룬다.

This chapter summarizes the basic concepts and nuances of evaluating algorithms in R. We covered the conceptual theory of memory management and system runtime in R. We discussed the best, worst, and average-case scenarios to evaluate the performance of algorithms. In addition, we also looked into the trade-off between a computer's configuration and algorithm's system runtime, algorithm asymptotic analysis, simplifying and classifying rules, and computational evaluation of programs. The next chapter will cover fundamental data structure and the concepts of lists in R.