2장. 알고리즘 분석

알고리즘은 컴퓨터의 자원을 사용하여 실행해야하는 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 일차적으로 집중할 것이며, 프로그램 실행 동안 메모리 사용과 시간 복장성과의 관계도 부분적으로 다룰 것이다. 2장에서는 다음 주제를 상세하게 다룬다.

An algorithm can be defined as a set of step-by-step instructions which govern the outline of a program that needs to be executed using computational resources. The execution can be in any programming language such as R, Python, and Java. Data is an intricate component of any program, and depending on how data is organized (data structure), your execution time can vary drastically. That's why data structure is such a critical component of any good algorithm implementation. This book will concentrate primarily on running time or time complexity and partly on memory utilization and their relationship during program execution. The current chapter will cover following topics in detail:

* 최선, 최악, 그리고 평균적인 경우
* 컴퓨터 대 알고리즘
* 알고리즘 점근 분석(asymptotic analysis)
  + 실행시간 상한 평가 = 빅 오 표기법(Big O notation)
  + 실행시간 하한 평가 = 빅 오메가 표기법(Big Ω notation)
  + 빅 세타 표기법 (Big Θ notation)
  + 규칙 단순화
  + 분류 함수
* 프로그램의 성능 평가
* 문제 분석
* 시스템 공간 한계
* 경험적 분석
* Best, worst, and average cases
* Computer versus algorithm
* Algorithm asymptotic analysis
  + Upper bounds evaluation
  + Lower bounds evaluation
  + Big Θ notation
  + Simplifying rules
  + Classifying functions
* Computation evaluation of a program
* Analyzing problems
* Space bounds
* Empirical analysis

<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 오른차순으로 출력하는 알고리즘을 프로그래밍해야 한다. 사용자가 정의한 입력과 사용자가 원하는 출력 사이의 연결 고리 역할을 하는 정렬 알고리즘은 다양한 방법으로 접근할 수 있다.

Data structure is a critical component in any algorithm. Before we go into details; let' illustrate this with an example; a sorting algorithm for positive integer for a finite length needs to be programmed using user input, and the output is to be displayed in ascending order. The sorting algorithm, which acts as a connector between the user-defined input and user-desired output can be approached in multiple ways:

* 버블 정렬(bubble sort)과 쉘 정렬(shell sort). 정렬의 단순한 형태이지만 매우 비효율적이다.
* Bubble sort and shell sort, which are simple variants of sorting, but are highly inefficient
* 삽입 정렬과 선택 정렬. 일차적으로 작은 데이터셋 정렬에 사용된다.
* Insertion sort and selection sort, primarily used for sorting small datasets
* 병합 정렬, 힙 정렬, 그리고 퀵 정렬. 평균적인 시스템 런타임을 보이는 복잡성에 기반한 효율적인 정렬 방법
* Merge sort, heap sort, and quick sort, which are efficient ways of sorting based on the complexities involved in an average system runtime
* 계수 정렬, 버킷 정렬, 그리고 기수 정렬 등의 분산 정렬. 런타임과 메모리 사용량 모두 처리할 수 있다.
* Distributed sorts such as counting sort, bucket sort, and radix sort, which can handle both runtime and memory usage

이 각각의 방법은 차례로 특정 인스턴스의 집합을 더 효과적으로 처리할 수 있다. 이렇게 말하면 좋은 알고리즘에 대한 개념을 본질적으로 흐리게 된다. 다른 많은 특성 중에 다음과 같은 특성을 가지고 있어야 알고리즘이 좋다고 말할 수 있다.

Each of these options can, in turn, handle a particular set of instances more effectively. This essentially reduces the concept of good algorithm . An algorithm can be termed as good if it possesses attributes such as the following among many others:

* 실행 시간 감소
* 메모리 사용량 감소
* 읽기 쉬운 코드
* 입력값에 대한 일반성
* Shorter running time
* Lesser memory utilization
* Simplicity in reading the code
* Generality in accepting inputs

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

In general, a problem can be approached using multiple algorithms, and each algorithm can be assessed based on certain parameters such as:

* 시스템 런타임
* 메모리 요구사항
* System runtime
* Memory requirement

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

However, these parameters are generally affected by external environmental factors such as:

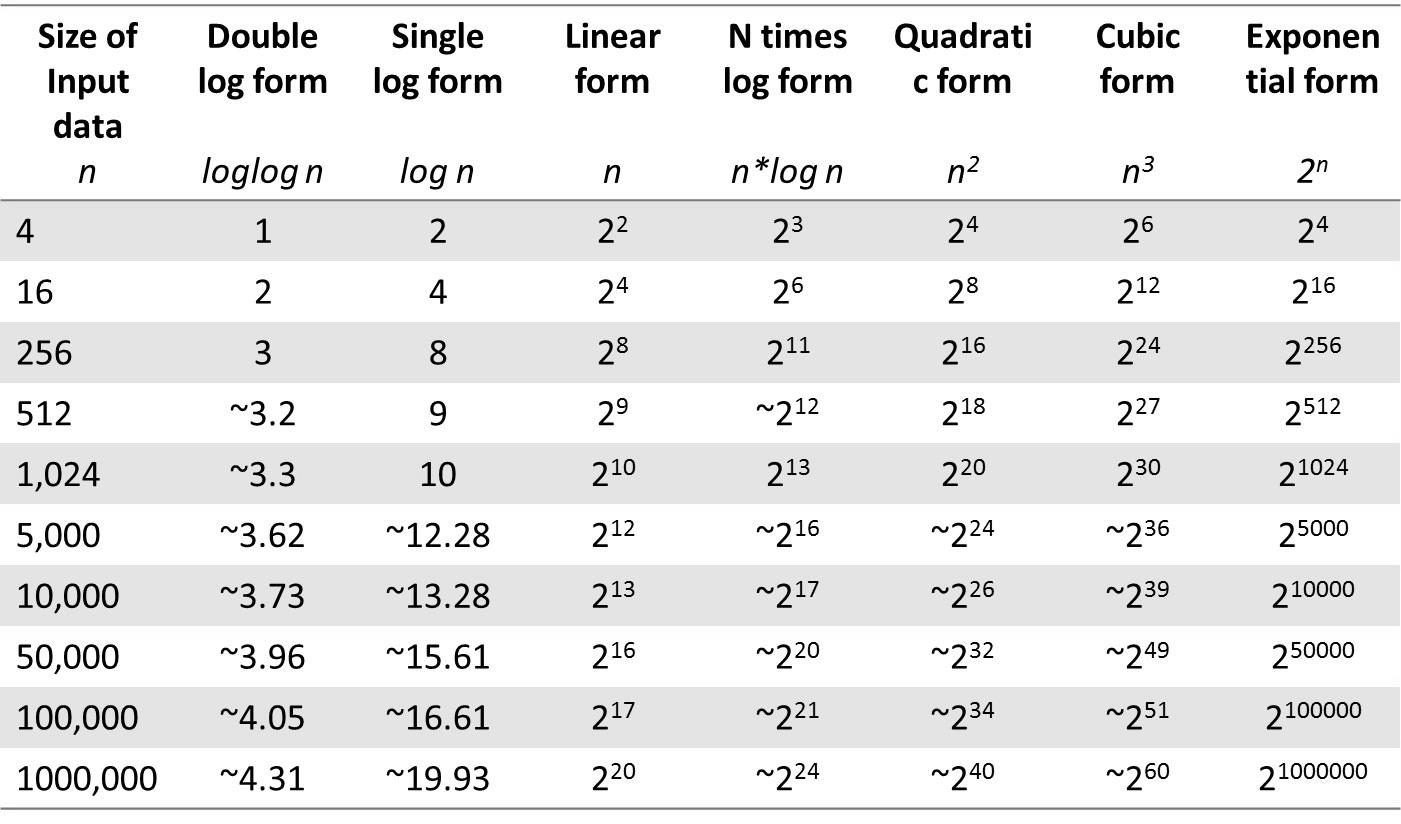
* 데이터 구조에 대한 처리
* 시스템의 소프트웨어 및 하드웨어 구성
* 코드를 작성하고 컴파일하는 스타일
* 프로그래밍 언어
* Handling of data structures
* System software and hardware configurations
* Style of writing and compiling codes
* Programming language

모든 외부적인 요인을 제어하기는 거의 불가능하기 때문에, 성능 비교를 위해 다양한 알고리즘의 시스템 런타임을 추정하는 것(이상적 시나리오 분석)은 어렵다. 이상적 시나리오 분석은 알고리즘 성능 평가를 위해 알고리즘을 구현하고 실행시키는 과정이 요구된다. 하지만 알고리즘을 설계하려고 하는 시나리오에서는 점근 분석이 알고리즘 성능을 평가하기 위해 사용된다.

As it is almost impossible to control all external parameters, it becomes difficult to estimate the system runtime of multiple algorithms for performance comparison (ideal scenario analysis). Ideal scenario analysis requires algorithms to be implemented and executed for evaluating algorithm performance. However, in a scenario where the user is trying to design an algorithm, asymptotic analysis is utilized to evaluate algorithm performance.

점근 분석은 전체 프로그램을 실제적으로 작성하고 컴파일하는 과정 없이 알고리즘의 효율성을 평가하는 방법이다. 점근 분석은 입력 데이터의 크기와 작업의 갯수를 기초로 가상의 시스템 런타임을 나타내는 함수식이다. 이 함수식은 입력 데이터의 증가율은 시스템 런타임과 직접적으로 비례한다는 원리를 바탕으로 한다. 예를 들어, 삽입 정렬의 경우 크기는 입력 벡터의 길이를 나타내고, 작업의 수는 정렬 처리의 복잡성을 나타낸다. 점근 분석은 알고리즘의 장점과 단점을 비교 평가하기 보다는 알고리즘을 구현하는데 들어가는 수고를 재보기 위해 사용된다. 다음 표는 넓리 사용되는 증가율 함수식을 보여준다. 더 자세한 내용은 뒷부분에서 논의한다.

Asymptotic analysis assesses algorithm's efficiency without actually coding and compiling the entire program. It is a functional form representing pseudo system runtime based on the size of input data and number of operations. It is based on the principle that the growth rate of input data is directly proportional to the system runtime. For example, in the case of insertion sorting, the size represents the length of the input vector, and the number of operations represents the complexity of sort operations. This analysis can only be used to gauge the consideration of implementing the algorithm rather than evaluating the comparative merits and demerits of algorithms. The following table represents the most widely used growth rate functional forms. The details are mentioned in later parts of the chapter.



<그림시작>

입력 데이터의 크기

이중로그 함수

로그 함수

선형 함수

n배 로그 함수

2차 함수

3차 함수

지수 함수

<그림끝>

<그림 2.1: 복잡성 평가를 위해 사용되는 다양한 증가율 함수식>

Figure 2.1: Diﬀerent growth rate functional form used for complexity evaluation

일반적으로 사용되는 증가율 함수식은 입력 데이터의 크기를 기준으로 하며, 알고리즘의 성능 분석을 위해 사용된다. 또한 알고리즘의 시스템 런타임 평가를 위한 가상의 함수식 역시 사용된다.

The most widely used functional forms of growth rates are based on the size of input data, which are used to analyze the performance of algorithms. These are also considered as pseudo-functional forms to evaluate algorithm's system runtime.

<대> R에서의 메모리 관리

메모리 관리는 일차적으로 가용한 메모리에 대한 관리와 함수의 실행을 더 유연하고 빠르게 실행하기 위해 요구되는 추가적인 메모리에 대한 예측을 포함한다. 여기서는 R 환경에서 객체의 저장을 다루는 메모리 할당의 개념을 다룰 것이다.

Memory management primarily deals with the administration of available memory and the prediction of additional memory required for smoother and faster execution of functions. The current section will cover the concept of memory allocation , which deals with storage of an object in the R environment.

메모리를 할당에 있어서 R은 서로 다른 객체에 각각 서로 다른 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 동작한다. pryr 패키지의 object\_size 함수는 R 기본 패키지의 object.size 함수와 비슷하다. 하지만 다음과 같은 점에서 더 정확하다.

During memory allocation R allocates memory differently to different objects in its environment. Memory allocation can be determined using the object\_size function from the pryr package. The pryr package can be installed from the CRAN repository using install.packages("pryr") . The package is available for ‎ R ( ≥ 3.1.0) . The object\_size function in pryr is similar to the object.size function in the base package. However, it is more accurate as it takes into account the:

* 현재 객체와 연관된 환경의 크기도 포함
* 주어진 객체 내의 공유된 요소까지 고려됨
* Environment size associated with the current object
* Shared elements within a given object under consideration.

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

The following are examples of using the object\_size function in R to evaluate memory allocation:

> object\_size(1) ## 한 개의 수치형 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 있다.

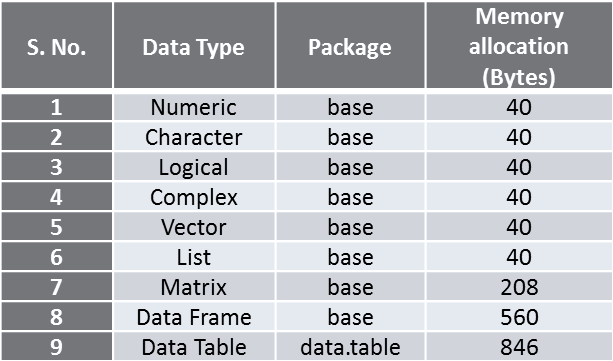
The storage required by an object can be attributed to the following parameters:

* 메타데이터(metadata) : 객체의 메타데이터는 문자형, 정수형, 논리형 등과 같이 객체의 타입에 의해 정의된다. 객체 타입은 디버깅할 때도 매우 유용하다.
* Metadata : Metadata of an object is defined by the type of object used such as character, integers, logical, and so on. The type can also usually be helpful during debugging.
* 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 메모리를 더 많이 필요로 한다.
* Node pointer : The node pointer maintains the link between the different nodes, and depending on the number of node pointers used, memory requirement changes. For example, a doubly linked list requires more memory than a singly linked list, as it uses two node pointers to connect to the previous and next nodes.
* 속성 포인터(attribute pointer) : 속성에 대한 참조를 유지하기 위한 포인터이다. 특히 변수에 의해 저장된 데이터에 대한 메모리 할당을 줄이는데 도움이 된다.
* Attribute pointer : Pointer to keep reference for attributes; this helps to reduce memory allocation, especially the data stored by a variable.
* 메모리 할당 : 현재 사용중인 시스템 공간을 나타내는 벡터의 길이
* Memory allocation : Length of the vector representing the currently used space.
* 크기 : 벡터의 길이에 실제로 할당된 시스템 공간의 크기
* Size : Size represent the true allocated space length of the vector.
* 메모리 패딩(memory padding) : 패딩은 하나의 구조체에 적용된다. 예를 들어, 각 요소는 8 바이트 경계 이후에 시작된다.
* Memory padding : Padding applied to a component, for example, each element begins after an 8-byte boundary.

\* 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 나중에 남는 빈공간을 패딩비트(padding bits)로 채운다.

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)

The Object\_size() command is also used to see the inherent memory allocation as shown in the following table:



<그림시작>

데이터 타입

패키지

메모리 할당량 (bytes)

<그림끝>

<그림 2.2: R의 다양한 데이터 타입의 초기화시에 할당되는 메모리>

Figure 2.2: Memory allocated during initialization of diﬀerent data types in R

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

The preceding table shows inherent memory allocated by each data structure/type. Let's simulate scenarios with varying lengths of a vector with different data types such as integer, character, Boolean, and complex. The simulation is performed by varying vector length from 0 to 60 as follows:

> 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 0 개부터 60 개까지 가지고 있는 각 수치형 벡터가 필요로 하는 계산된 메모리 크기를 저장한 변수이다. 이 수치형 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.

Num\_vec\_size computes the memory requirement for each numeric vector from zero to 60 number of elements. These elements are integers increasing sequentially, as stated in the function. Similarly, incremental memory requirements are calculated for character ( char\_vec\_size ), logical ( log\_vec\_size ), and complex ( comp\_vec\_size ) vectors. The result obtained from the simulation can be plotted using code:

> 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")

> plot(char\_vec\_size ~ vec\_length, xlab = "Character seq vector",

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

> abline(h = (c(0,56,64,80,96,112,176)+40), col = "grey")

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

> plot(log\_vec\_size ~ vec\_length, xlab = "Logical seq vector",

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

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

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

> plot(comp\_vec\_size ~ vec\_length, xlab = "Complex seq vector",

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

> abline(h = (c(0,56,64,80,96,112,176)+40), col = "grey")

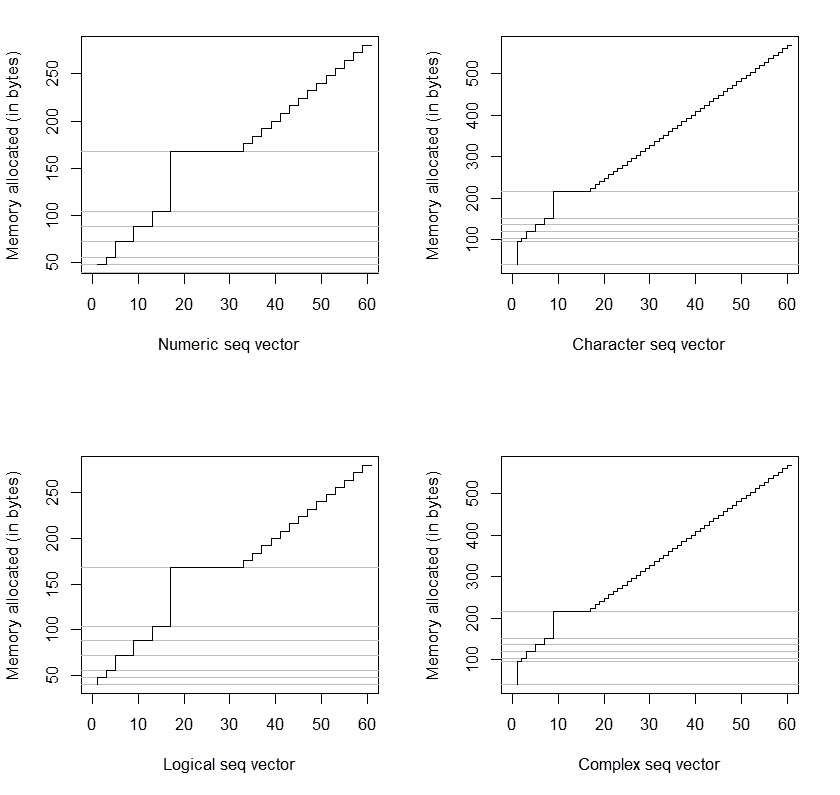
> lines(comp\_vec\_size, type = "S")

> par(mfrow=c(1,1))

(## 책에서 빠진 코드 부분 추가)

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

The result obtained on running the preceding code is shown in Figure 2.3 . It can be observed that memory allocated to a vector is a function of its length and the object type used. However, the relationship does not seem to be linear rather, it seems to increase in step. This is due to the fact that for better and consistent performance, R initially assigns big blocks of memory from RAM and handles them internally. These memory blocks are individually assigned to vectors based on the type and the number of elements within. Initially, memory blocks seem to be irregular towards a particular level (128 bytes for numeric/logical vector, and 176 bytes for character/complex vectors), and later become stable with small increments of 8 bytes as can be seen in the plots:



<그림시작>

할당된 메모리(bytes)

수치형 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>

Figure 2.3: Memory allocation based on length of vector

처음에 할당된 메모리의 크기가 다르기 때문에 수치형과 논리형 벡터는 비슷한 메모리 할당 패턴을 보이며, 복소수형 벡터는 문자형 벡터와 비슷하다. 메모리 관리는 알고리즘이 효율적으로 동작하는 데 도움이 되지만, 어떤 프로그램을 실행하기 전에 런타임을 기준으로 평가해야 한다. 이제 함수의 런타임을 얻고 그것을 다른 비슷한 함수들과 비교하는 것과 관련된 기본 개념을 이야기할 것이다.

Due to initial memory allocation differences, numeric and logical vectors show similar memory allocation patterns, and complex vectors behave similarly to the character vectors. Memory management helps to efficiently run an algorithm however before the execution of any program, we should evaluate it based on its runtime. In the next sub-section, we will discuss the basic concepts involved in obtaining the runtime of any function, and its comparison with similar functions.

<중> R에서의 시스템 런타임

시스템 런타임은 서로 다른 알고리즘들을 비교하는 데 핵심이다. 프로세스는 여러가지 옵션들을 비교하고 최선의 알고리즘을 선택하는데 도움이 된다. 다양한 알고리즘을 비교하는 것은 뒤에서 자세히 다룰 것이다.

System runtime is essential for benchmarking different algorithms. The process helps us to compare different options, and to pick the best algorithm. Benchmarking of different algorithms will be dealt with in detail in subsequent chapters.

CRAN에 있는 microbenchmark 패키지는 어떤 구문, 함수, 코드의 런타임을 밀리세컨드(1000분의 1초) 단위까지 측정할 때 사용된다. 그 정확성은 system.time() 함수를 대치할 만하다. 또한 모든 측정이 C 코드로 수행되기 때문에 과부하를 최소화한다. 다음 방법들은 수행 시간을 측정하는 데 사용된다.

The microbenchmark package on CRAN is used to evaluate the runtime of any expression/function/code at an accuracy of a sub-millisecond. It is an accurate replacement to the system.time() function. Also, all the evaluations are performed in C code to minimize any overhead. The following methods are used to measure the time elapsed:

* 윈도우즈 OS의 QueryPerformanceCounter 인터페이스
* 리눅스 OS의 clock\_gettime API
* 맥 OS의 mach\_absolute\_time 함수
* 솔라리스 OS의 gethrtime 함수
* The QueryPerformanceCounter interface on Windows OS
* The clock\_gettime API on Linux OS
* The mach\_absolute\_time function on MAC OS
* The gethrtime function on Solaris OS

이 책의 예제에서는 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)

In our current example, we will be using the mtcars data, which is in the package datasets. This data is obtained from 1974 Motor Trend US magazine, which comprises of fuel consumption comparison along with 10 automobile designs and the performance of 32 automobiles (1973-74 models).

Now, we would like to perform an operation in which a specific numeric attribute ( miles per gallon ( mpg ) needs to be averaged to the corresponding unique values in an integer attribute (carb means no of carburetors). This can be performed using multiple ways such as aggregate , group\_by , by , split , ddply(plyr) , tapply , data.table , dplyr , sqldf , dplyr and so on. For illustration, we have used the following four ways:

* aggregate function:

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

* ddply from plyr package:

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

* data.table format:

library(data.table)

mtcars\_tb = data.table(mtcars)

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

* group\_by function:

library(dplyr)

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

Then, microbenchmark is used to determine the performance of each of the four ways mentioned in the preceding list. Here, we will be evaluating each expression 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)),

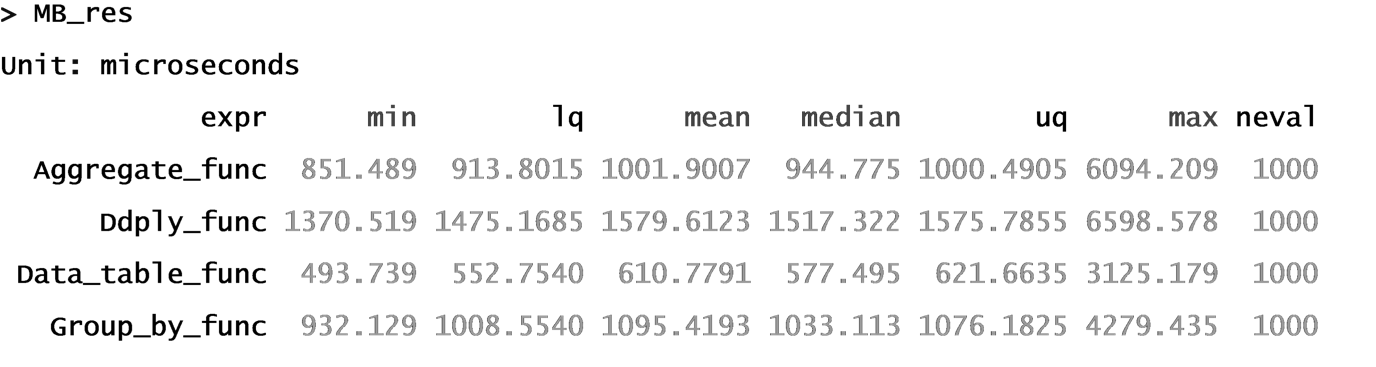
+ Data\_table\_func = mtcars\_tb[,mean(mpg),by=carb],

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

+ times=1000

+ )

The output table is as follows:



The output plot demonstrating distribution of execution time from each approach is shown in Figure 2.4 :

> library(ggplot2)

> autoplot(MB\_res)

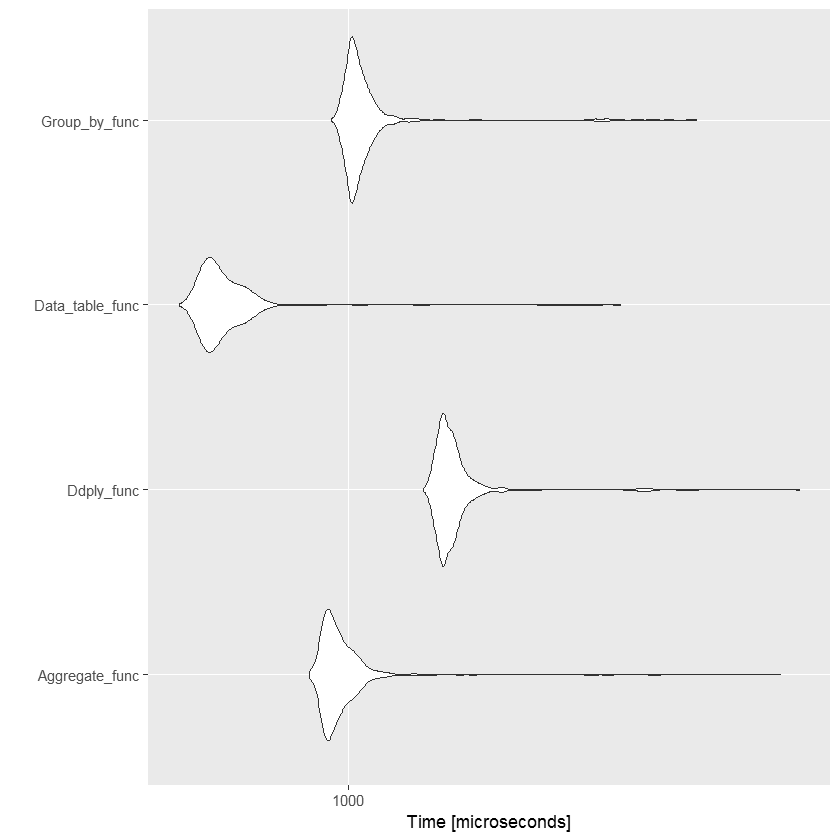


Figure 2.4: Distribution of time (microseconds) for 1000 iterations in each type of aggregate operation

Among these four expressions and for the given dataset, data.table has performed effectively in less possible time as compared to the others. However, expressions need to be tested under scenarios with a high number of observations, high number of attributes, and prior to finalizing the best operator.

<중> Best, worst, and average cases

Based on the performance in terms of system runtime, a code can be classified under best, worst or average category for a particular algorithm. Let's consider a sorting algorithm to understand this in detail. A sorting algorithm is used to arrange a numeric vector in an ascending order, wherein the output vector should have the smallest number as its first element and largest number as its last element with intermediate elements in subsequent increasing order. Currently, we will be implementing insertion sorting, however Chapter 5 , Sorting Algorithms , will cover various types of sorting algorithms in detail. In insertion sorting algorithm, the elements within a vector are arranged based on moving positions. The best, worst and average cases are data dependent. Now, let's define best, worst and average case scenarios for insertion sorting algorithm.

* Best case : A best case is one which requires the least running time. For example – a vector with all elements arranged in increasing order requires the least amount of time for sorting.
* Worst case : A worst case is one which requires the maximum possible runtime to complete sorting a vector. For example – a vector with all the elements sorted in decreasing order requires the most amount of time for sorting.
* Average case : An average case is one which requires an intermediate time to complete sorting a vector. For example – a vector with half elements sorted in increasing order and the remaining in decreasing order. An average case is assessed using multiple vectors of differently arranged elements.

Generally, the best-case scenarios are not considered to benchmark an algorithm, since it evaluates an algorithm most optimistically. However, if the probability of occurrence of best case is high, then algorithms can be compared using the best-case scenarios. Similar to best- case, worst-case scenarios evaluate the algorithm most pessimistically. It is only used to benchmark algorithms which are used in real-time applications, such as railway network controls, air traffic controls, and the like. Sometimes, when we are not aware of input data distributions, it is safe to assess the performance of the algorithm based on the worst-case scenario.

Most of the time, average-case scenario is used as a representative measure of algorithm's performance; however, this is valid only when we are aware of the input data distribution. Average-case scenarios may not evaluate the algorithm properly if the distribution of input data is skewed. In the case of sorting, if most of the input vectors are arranged in descending order, the average-case scenario may not be the best form of evaluating the algorithm.

In a nutshell, realtime application scenarios, along with input data distribution, are major criterions to analyze the algorithms based on best, worst, and average cases.

<중> Computer versus algorithm

This section primarily deals with details on the trade-off between a computer's configuration and an algorithm's runtime. Let's consider two computers A and B, with B being 10 times faster than A, along with an algorithm whose system runtime is around 60 minutes in computer A for a dataframe of 100,000 observations. The functional form of algorithm's system runtime is n 3\* . However, this functional form can be considered as an equivalent to the growth in the number of operations required to complete the running of the algorithm. In other words, the functional form of system runtime and the growth rate is same. The following situations will help us understand the trade-off in detail:

Situation 1 : Will computer B, which is ten times faster than computer A, be able to reduce the system runtime of the algorithm to six minutes from the current 60 minutes?

This is perhaps yes, provided the size of the dataset remains consistent in both computers A and B. However, if we increase the size of the dataframe by 10 times, the following situation arises.

Situation 2 : Will the algorithm in computer B be able to run the increased dataframe of 1,000,000 observations in 60 minutes, as computer B is 10 times faster than computer A?

This becomes tricky, as we are dealing not only with a change in computer configuration but also change in the size of input data, which makes the algorithm perform non-linearly (in our case-cubic form). The following table elucidates the capacity of computer B, which is 10 times faster than computer A, to handle the increase in size of the input dataframe, which can be run in a fixed time period for a given functional form of the algorithm's growth rate. Assume that computer A can perform 100,000 operations in 60 minutes, whereas computer B can perform 1,000,000 operations in 60 minutes. The \*k is a constant positive real number ∼ same time period of x minutes for computer A and computer B.

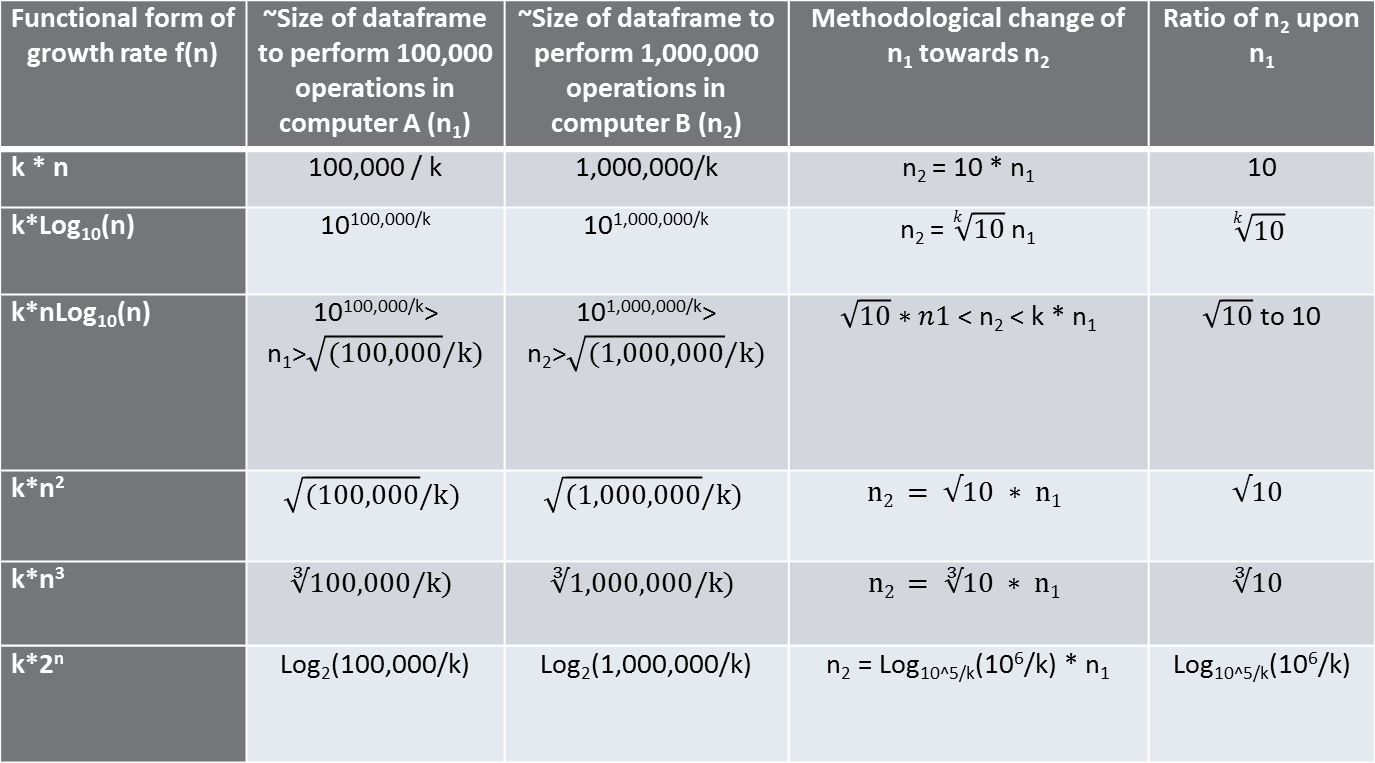


Figure 2.5: Performance comparison of widely used growth rate functions using two diﬀerent sizes of dataframes

Let's understand each functional form of algorithm's growth rate:

* Linear form : From Figure 2.5 , it can be seen that for any constant k , computer B can process 10 times bigger input dataframe within the same time period of 60 minutes. In other words, the processing speed of an algorithm with a linear runtime functional form is independent of the constant k , which affects the runtime behavior of the absolute size of input data. Also, for a given fixed runtime, if a system is i times faster than another system, then the data handling capacity of the faster system is also i times higher than the slower system. Hence, relative performance of the two computers is independent of the algorithm's growth rate constant k .
* Square and cubic form : We can see that for any constant k , computer B can process only the square root of 10 (3.16) and cube root of 10 (2.15) times the input dataframe within the same time period of 60 minutes. Here also, the performance of computer B is not affected by the constant k, which affects the absolute size of the input data size. In other words, computer B, which is 10 times faster than computer A, can run only 3.6 (square root of performance increase in case of square function form) times of the data in a given fixed time period, unlike 10 times as in the case of linear form. Hence, as computers perform much faster, the benefit attained towards the size of input data becomes highly disproportionate due to the inherent nature of ith root (where i is 2,3,4, and so on).
* Logarithmic form : For this functional form two variants are widely used:
  + Log(n) : The increment in size of the input dataframe is dependent on two factors – one being the increment in the system's computing performance, and other being the constant k . However, disparity between the system's increase in computing configuration and its performance continues as the increase in size of input data is directly proportional to k th root of increment in the system's performance.
* nLog(n) : The enhancement in handling higher input data size upon increase in the system's computing performance is greater than the improvement obtained using the quadratic functional form, but lower than algorithms with a linear functional form.
* Power form (exponential) : In power form, the system runtime of the algorithm increases exponentially upon increase in size of input data. For k= 1 , the size of the input data to perform 100,000 operations in computer A is ∼ 11. Similarly, the size of the input data to perform 1,000,000 operations in computer B is ∼ 14. Hence, n 2 = n 1 + 3. This clearly shows that a system with 10 times increase in performance can handle only a marginal increase in data size within a given, fixed runtime period. The increase in size of the input data for an algorithm with an exponential or power functional form is almost additive rather than multiplicative. In other words, if the algorithm in computer A has a system runtime of 60 minutes for a data size of 100,000 observations, then computer B, which is 10 times faster than computer A, can run only an input data of size 100,003 observations in 60 minutes. Thus, the performance of algorithms with an exponential functional form is much different than the remaining growth functional forms.

Now, let's dive deep into situation 3, which deals with comparing the trade-off between algorithms and computers.

Situation 3 : Which scenario is better for an algorithm with a growth rate functional form of n 3 – to increase the computer's performance capability,or to reconfigure the algorithm to change its growth rate functional form?

As we have already assessed the scenario of increasing the system's performance capability under Situation 2 , let's now try to analyze the situation of reconfiguring the algorithm's growth rate functional form.

Currently, our algorithm possesses a functional form of n3. For an input data of size 1,000, the total number of operations required is 1,0003. Suppose, if the current algorithm can be reconfigured to nLog10(n), then the total number of operations would reduce to 3,000, which is much lower than 1,0003. As the number of operations using n3 is more than 10 times the number of operations using nLog10(n) for every n>2, it is more advisable to reconfigure the growth functional form of the algorithm rather than increase the computational performance capability by 10 times.

To summarize the trade-off:

* Algorithms with slower growth rate show a better performance in handling larger data observations upon upgrading the computer's computational configuration
* The rate of handling larger data sets by algorithms with a faster growth rate may not be proportionately handled upon increasing the computer's computational capability

<중> Algorithm asymptotic analysis

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:

* 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.

﻿

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:

* Standard insertion sorting : f(n) = α+ c\*n 2
* Standard merge sorting : f(n) = α + c\*n\*log 2 (n)

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

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.

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

<소> Upper bounds or Big O notation

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.

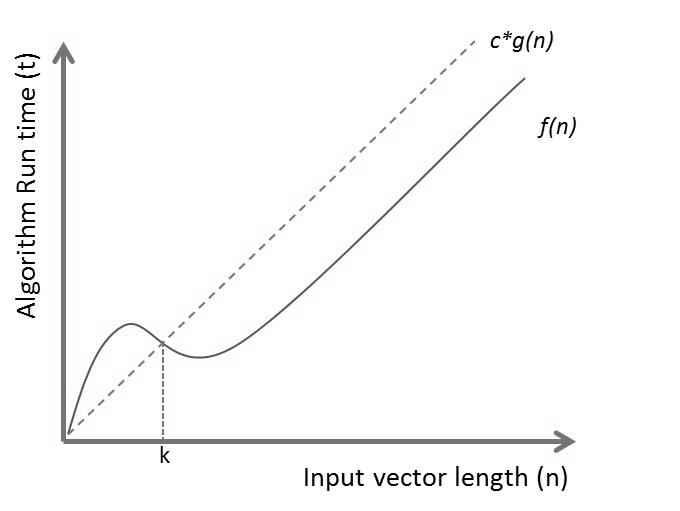


Figure 2.6: f(n) is Big O of g(n) for all n>k

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.

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

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 ).

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 + 32n 2 + 4\*log(n)

Now, for every n>0 :

f(n) < 25 n 2 + 12 n 2 + 32n 2 + 4 n 2

f(n) < (25+ 12+ 32+ 4)n 2

f(n) = O(n ) , wherein g(n) is n and c=(25+12+32+4)

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.

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

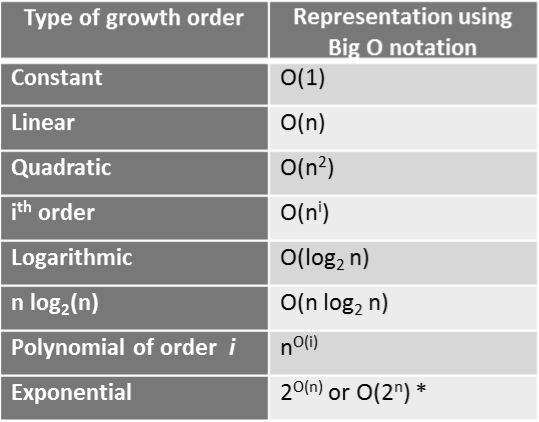


Figure 2.7: Big O representation of various growth order functions

<소> Lower bounds or Big Omega notation (Ω)

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.

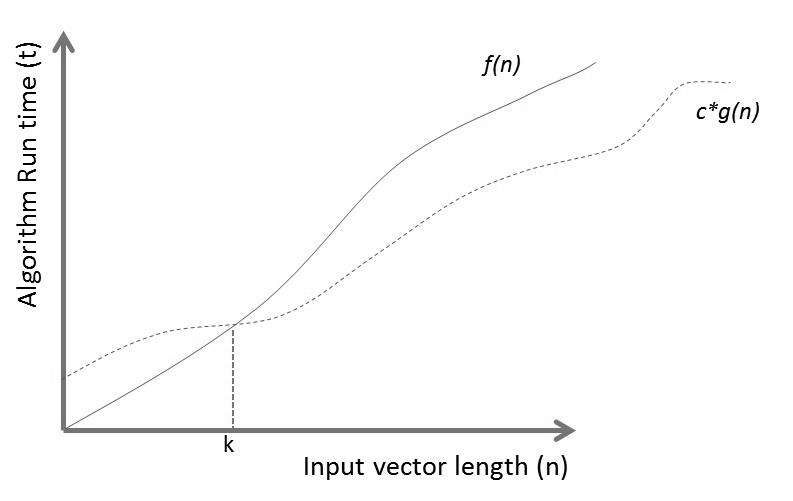


Figure 2.8: f(n) is Big- Ω of g(n) for all n>k

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.

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

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 ).

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 + 32n 2 + 4\*log(n)

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

f(n) > 25 n 2

f(n) > Ω (n 2 ) wherein g(n) is n 2 and c=25

The smallest of lower bound is as follows:

f(n) > 25

f(n) > Ω (25)

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

----------------------------------------------------------------- 12/17 Goal !!!

<소> Big θ notation

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:

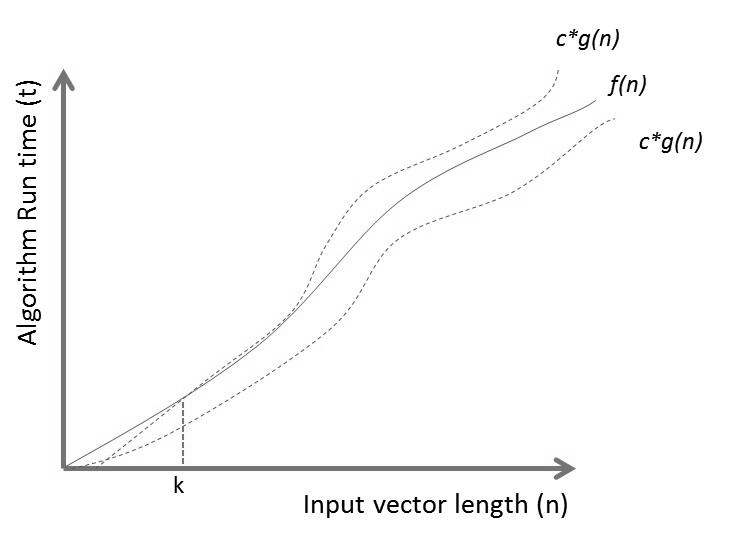


Figure 2.9: f(n) is Big- θ of g(n) for all n>k

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 .

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

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

<소> Simplifying rules

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.

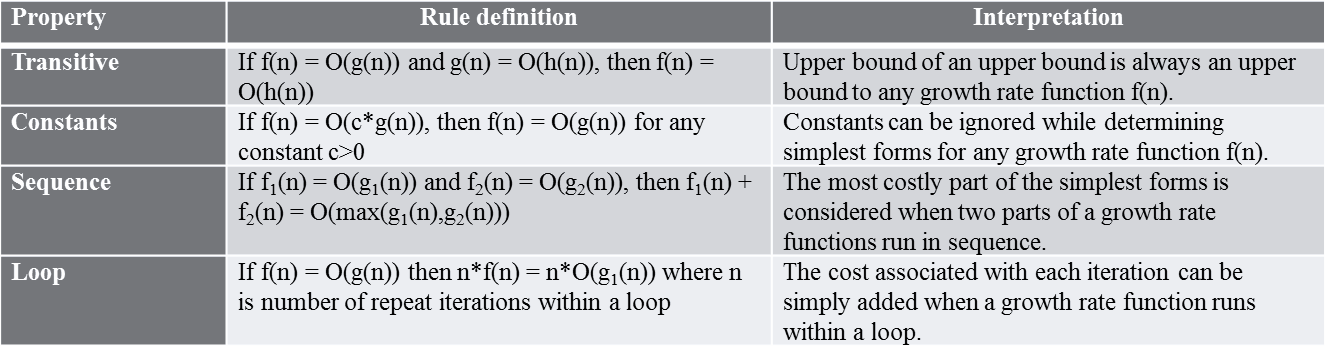
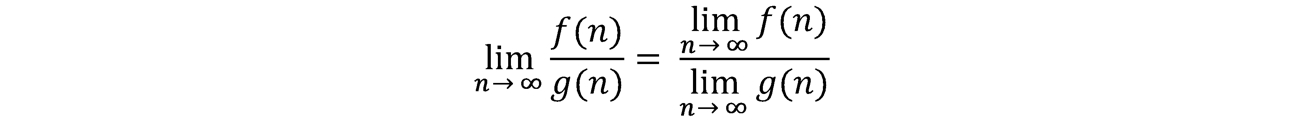


Figure 2.10: Deﬁnition of simplifying rules along with their interpretations

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

<소> Classifying rules

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:



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

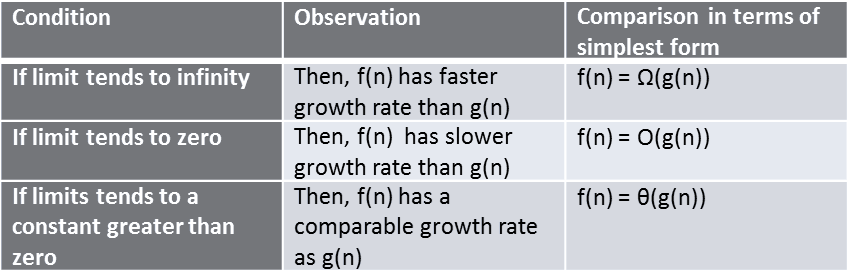


Figure 2.11: Classifying rule forms

<중> Computation evaluation of a program

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

<소> Component 1 – Assignment operator

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

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

<소> Component 2 – Simple loop

Consider a simple for loop with assignment operations.

a <- 0

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

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

Figure 2.12: Asymptotic analysis of a simple for loop

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

<소> Component 3 – Complex loop

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

}

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

Figure 2.13: Asymptotic analysis of a complex loop

2

<소> Component 4 – Loops with conditional statements

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

} }

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

Figure 2.14: Asymptotic analysis of a conditional loop

The total cost of this loop with if...else conditions using simplifying rules is θ(n 2 ) . 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> Component 5 – Recursive statements

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

}

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

Figure 2.15: Asymptotic analysis of a recursive statement

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

<중> Analyzing problems

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

<중> Space bounds

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.

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.

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> Exercises

1. The following are some growth-rate functional forms. Can you arrange them in

the order of slower to faster performance?

10n 3

3( log e n) 2 10n

100n

Log 2 n 2 Log 2 n 3 Log 3 n 2 Log 3 n 3 n 1.5

2. Answer the following questions:

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:

10nlog 2 n 5n 3

8log 3 n 2

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 .

2

f(n)= n 2 ; g(n) = 2 n f(n)= 25 ; g(n) = 2 10

n

n

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

7. Evaluate Big θ for the following code snippets:

First snippet:

for(i in 1:100) {

a = i\*10

b = a+50}

[ 56 ]

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

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

Second snippet:

i=1; a=0

while(i<100) {

a = c(a,i) i=i+1}

Third Snippet:

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}}

Fourth snippet:

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")}"

<대> Summary

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.

[ 57 ]