R 데이터 구조와 알고리즘

1장. 시작하기

빠르고 효율적인 정보 검색은 대다수 컴퓨터 프로그램의 일차적인 목적이다. 데이터 구조와 알고리즘은 데이터를 더 빠르게 처리하고 검색하는 목적을 이루는데 도움이 된다. 데이터로부터 추론될 수 있는 다음과 같은 질문에 대답할 때 정보 검색은 알고리즘과 쉽게 통합될 수 있다.

* 매출은 시간이 지남에 따라 어떻게 증가하는가?
* 고객의 방문 시간 분포는 어떠한가?
* 오후 3시에서 6시 사이에 방문한 모든 고객 중에서 아시아인 대비 중국인은 얼마나 많이 구매하는가?
* 모든 방문 고객 중에서 같은 도시에서 온 사람은 얼마나 되는가?

이 질문들에 대한 데이터를 처리하는 데 있어서, 특히 빅데이터 상황이라면, 데이터 구조와 알고리즘은 데이터 검색을 수행하는데 아주 중요한 역할을 한다. 이 책에서는 정보 처리에 일반적으로 사용되는 리스트, 큐, 스택과 같은 기본적인 데이터 구조와 그에 대립되는 다른 데이터 구조들에 대해 설명할 것이다. 또한 정의된 데이터 구조의 검색 및 처리 성능에 대한 데이터 구조 및 알고리즘 평가 방법에 대해서도 알아볼 것이다.

알고리즘은 복잡성과 효율성에 의해 평가된다. 복잡성은 알고리즘 설계가 프로그래밍과 디버깅이 쉽게 되어 있는지를 나타내고, 효율성은 알고리즘이 컴퓨터의 자원을 최적으로 사용하는지를 말해준다. 이 책은 데이터 구조를 사용하는 알고리즘의 효율성 부분에 초점을 맞출 것이며, 이 장에서는 데이터를 추출해내기 위해 사용되는 데이터 구조와 알고리즘의 중요성에 대해 이야기할 것이다.

데이터 구조 소개

집적회로(IC)는 발명된 이후로 평방 인치당 트랜지스터의 수가 매년 두 배씩 증가한다는, 1965년에 발표된 무어의 법칙에 따라 컴퓨터의 계산 능력은 향상되고 있다. 1975년 그는 포화 상태로 인해 매년이 아닌 2년마다 두 배가 된다고 그의 예측을 수정했다.

<그림 1.1 : 무어의 법칙>

컴퓨팅 능력이 계속 향상되고 있음에도 불구하고, 문제의 복잡성과 데이터 소스도 지난 10여 년 간 기하급수적으로 증가하면서 효율적인 알고리즘의 필요성도 더 강조되고 있다.

<그림 1.2 : 비정형 데이터 크기의 증가>

2008년에서 2015년 사이에 일어난 데이터의 폭발은 정형, 반정형, 그리고 비정형 등 모든 종류의 데이터셋을 사용하여 통찰력을 끌어내기 위해 많은 노력을 기울이는 데이터 과학이라는 새로운 분야로 이끌었다. 그러므로 대규모의 데이터를 효율적으로 다루는 것은 데이터셋을 효율적으로 저장하고 검색하기 위해 매우 중요하다. 예를 들어, 사전에서 한 단어를 찾을 때 만약 데이터가 랜덤하게 구성되어 있다면 많은 시간이 걸릴 것이다. 그렇기 때문에 정렬된 리스트 데이터 구조는 단어의 빠른 검색을 보증한다. 또한 입력 위치를 기반으로 한 도시에서 최적의 이동 경로를 찾는 것은 도로 연결망, 위치 정보, 그리고 지오메트리 형태로 저장된 데이터를 필요로 한다. 이상적으로 문자, 정수, 부동소수 등과 같은 기존의 내장 데이터 타입으로 저장된 변수도 스칼라 유형의 데이터 구조라고 할 수 있다. 그러나 공식적으로 데이터 구조는 컴퓨터에서 연관된 정보를 조직화하여 효율적으로 사용할 수 있는 하나의 스킴(scheme)으로 정의된다.

알고리즘의 경우 충분한 공간과 시간이 주어진다면 관심있는 질문들에 대답하기 위해 어떤 데이터셋이라도 저장하고 처리할 수 있다. 하지만 정확한 데이터 구조를 선택하는 것은 컴퓨터의 메모리와 자원을 절약하는데 상당한 도움이 된다. 예를 들어, 매일 방문하는 고객의 수를 정수형 데이터 타입이 아닌 부동소수형 데이터 타입으로 설정한다면 두 배의 메모리를 필요로 하게 된다. 하지만 현실세계에서 컴퓨터의 자원과 공간은 항상 제한되어 있다. 그러므로 주어진 자원과 시간 안에서 원하는 목표를 이룰 수 있다면 효과적인 솔루션이라고 할 수 있다. 이것은 알고리즘을 설계하는 동안 서로 다른 데이터 구조들 간의 성능을 비교하는 비용함수로 사용할 수 있다. 데이터 구조를 선택할 때 고려해야 할 두 가지 주요한 제약은 다음과 같다.

* 선택한 데이터 구조에서 지원해야 하는 항목 추가, 항목 삭제, 검색과 같은 기본적인 작업을 결정하기 위해 문제를 분석
* 각 작업에 대한 자원 제약사항을 평가

데이터 구조는 문제 상황에 따라 선택된다. 예를 들어, 전체 데이터가 초기에 로드되고, 데이터에 대한 변경이나 추가가 없으면 비슷한 데이터 구조가 요구된다. 하지만 위 상황에서 데이터 구조에 대한 삭제 작업이 포함된다면 데이터 구조 구현은 좀더 복잡해질 것이다.

[팁]

코드를 다운받는 자세한 방법은 이 책의 서문에 있으니 꼭 확인하기 바란다. 이 책을 위한 코드는 깃허브의 다음 주소에 있다.

<https://github.com/PacktPublishing/R-Data-Structures-and-Algorithms>.

또한 출판사의 다양한 책들과 비디오의 소스 코드는 다음 주소에서 확인 가능하다.

<https://github.com/PacktPublishing/>

추상 데이터 타입과 데이터 구조

추상 데이터 타입(abstract data type, ADT)은 데이터 구조에 대한 기능과 처리를 고수준에서 정의할 때 사용되며, 데이터 구조를 상세하게 구현하기 전에 확인해야 한다. 예를 들어, 링크드 리스트를 구현하기 전에 정의된 링크드 리스트에서 다음 중 수행하려는 작업이 무엇인지 아는 것이 좋을 것이다.

* 링크드 리스트에 항목을 추가할 수 있어야 한다.
* 링크드 리스트로부터 항목을 삭제할 수 있어야 한다.
* 링크드 리스트에서 항목을 검색할 수 있어야 한다.
* 링크드 리스트가 비어 있는지 아닌지 확인할 수 있어야 한다.

정의된 추상 데이터 타입은 전략을 수립하기 위해 필요하다. 이 책에서는 여러가지 데이터 구조에 대한 ADT를 더 자세하게 알아볼 것이다. ADT의 정의를 말하기 전에, 데이터 구조를 위한 생태계를 구성하는 데이터 타입과 그 특성에 대해 먼저 이해하는 것은 매우 중요하다.

데이터 타입은 불리언(boolean), 정수(integer), 부동소수(float), 문자열(string) 등과 같이 데이터의 다양한 유형을 분류하는 방법이다. 데이터셋을 효율적으로 분류하기 위해서 모든 데이터 타입은 다음과 같은 특성을 가져야 한다.

* 원자성 : 하나의 단위 개념으로 정의될 수 있어야 한다.
* 추적가능성 : 동일한 데이터 타입으로 묶을 수 있어야 한다.
* 정확성 : 모호하지 않아야 한다.
* 명확성과 간결성 : 이해 가능해야 한다.

데이터 타입은 다음과 같이 두 가지로 나눌 수도 있다.

* 내장 데이터 타입
* 파생 데이터 타입

한 언어에 내장되어 지원되는 데이터 타입을 내장 데이터 타입이라고 한다. R은 다음과 같은 데이터 타입을 지원한다.

* 정수형(Integers)
* 부동소수형(Float)
* 불리언(Boolean)
* 문자열(Character)

내장 데이터 타입과 통합되어 추가, 삭제, 정렬, 병합 등과 같은 연관된 작업을 처리하기 위해 다음과 같은 파생 데이터 타입이 있다.

* 리스트(List)
* 배열(Array)
* 스택(Stack)
* 큐(Queue)

파생 데이터 타입 또는 데이터 구조는 두 가지 부분에서 연구된다.

* 추상 데이터 타입 또는 수학/논리 모델
* 프로그램 구현

ADT는 소프트웨어에서 데이터 타입의 실현이다. 우리는 일반적으로 데이터 구조에 대해 사용자가 사용하는 고수준의 기능이나 작업에 관심을 갖지만, 내부적으로 이 기능들이 어떻게 동작하는지에 대해서는 모른다. 예를 들어, 한 사용자가 금융 소프트웨어에서 검색기능을 사용하여 스미스 씨의 거래 이력을 검색한다고 하자. 사용자는 이 작업이 동작하는 방법 또는 데이터 구조의 자세한 구현에 대해서는 전혀 알지 못한다. 그러므로 ADT의 동작은 오직 입력과 출력에 의해서만 관리된다.

<그림 1.3 : ADT 프레임워크>

ADT는 데이터 타입이 어떻게 구현되는지 알 수 없다. 사용자에게 숨겨지고 외부의 접근으로부터 보호되기 때문인데, 이것이 캡슐화의 개념이다. 데이터 구조는 ADT가 프로그래밍 언어에 의해 구현되는 부분이다. ADT는 그림 1.4에 보이는 것처럼 여러가지 구현 전략에 의해 달성될 수 있다.

<그림 1.4 : 정수형 데이터 타입으로 된 배열을 사용한 스택과 큐 구현>

ADT에서 제공되는 추상화는 프로그래밍의 복잡성을 관리하는데 도움이 된다. ADT는 구현에 필요한 형식과 작업을 결정하기 때문에 논리 형식이라고 한다. ADT는 특정 형식의 데이터 구조를 사용하여 구현된다. ADT를 구현하기 위해 사용한 데이터 구조는 데이터 타입의 물리적인 형식이다.

문제와 알고리즘과의 관계

Relationship between problem and algorithm

문제는 수행되어야 할 작업으로 정의할 수 있다. 수행할 작업의 실행은 일차적으로 두 요소로 나눌 수 있다.

Problem can be defined as the task to be performed. The execution of the task to be performed can be divided primarily into two components:

* Data
* Algorithm
* 데이터
* 알고리즘

그러나 문제의 제약사항, 자원 제약사항, 허용된 시간 등의 개발에 영향을 줄 수 있는 다른 관리 요인이 있을 수도 있다.

However, there could be other controlling factors which could affect your approach development, such as problem constraints, resource constraints, computation time allowed, and so on.

문제의 데이터 요소는 숫자, 텍스트, 파일과 같이 우리가 다루어야 할 정보를 대표한다. 예를 들어, 회사의 직원 정보를 관리하길 원한다고 가정하면 거기에는 직원 이름 및 그와 관계된 자세한 항목들이 포함된다. 이 데이터는 정기적으로 관리되고 업데이트 되어야 한다.

The data component of the problem represents the information we are dealing with, such as numbers, text, files, and so on. For example, let's assume we want to maintain the company employee records, which include the employee name and their associated work details. This data needs to be maintained and updated regularly.

문제의 알고리즘 부분은 상세한 구현을 대표한다. 여기에는 현재의 데이터를 어떻게 관리할 것인가 하는 문제가 수반된다. 데이터와 문제의 요구사항에 따라 우리는 데이터 구조를 선택한다. 그리고 데이터 구조에 따라 데이터셋을 관리하기 위한 알고리즘을 정의해야 한다. 예를 들어, 회사의 직원 데이터셋을 링크드 리스트 또는 딕셔너리에 저장할 수 있다. 데이터를 저장하기 위해 정의된 데이터 구조를 기반으로 검색, 추가, 삭제가 이루어지며, 데이터 구조 상에 수행되는 작업들은 알고리즘에 의해 제어되고 프로그램으로서 구현된다. 그러므로 프로그램은 어떤 작업을 하기 위해 컴퓨터에 주어지는 단계적인 명령이라고 할 수 있다.

The algorithm part of a problem represents more of implementation details. It entails how we are going to manage the current data. Depending on the data and problem requirements, we select the data structure. Depending on the data structure, we have to define the algorithm to manage the dataset. For example, we can store the company employees dataset into a linked list or dictionary. Based on the defined data structure to store data approach for searching, insertion, and deletion and so on operations performed on data structure changes which is control by algorithms and implemented as program. Thus, we could state that a program is the step-by-step instructions given to a computer to do some operations:

프로그램 = f(알고리즘, 데이터 구조)

Program = f(Algorithm, Data Structure)

정리하면, 프로그래은 모든 문제와 자원 제약사항을 고려하여 선택된 알고리즘을 사용하여 정의된 문제를 해결하기 위한 단계적인 명령들의 그룹이다. 이 책에서는 여러가지 데이터 구조와 알고리즘의 구현을 시연하기 위해 R 을 사용할 것이다.

To summarize, we could state that a program is the group of step-by-step instructions to solve a defined problem using the algorithm selected considering all problems and resource constraints. This book will use program representations using R to demonstrate the implementation of different data structures and algorithms.

R 기초

Basics of R

R은 Ross Ihaka와 Robert Gentleman에 의해 설계되고 만들어진 통계 프로그래밍 언어이다. 이것은 AT&T의 벨 연구소에서 만든 S 언어에서 파생되어 나왔다. 통계 분석뿐만 아니라, R은 강력한 시각화 기능을 지원한다. 이것은 오픈 소스이며, 일반 공중 라이선스(GPL) 하에서 자유롭게 배포 가능하다. Comprehensive R Archive Network(CRAN)이라는 저장소에는 다양한 분석에 사용되는 8,400 개 이상의 패키지가 있으며, 무료로 설치 및 사용할 수 있다.

R is a statistical programming language designed and created by Ross Ihaka and Robert Gentleman. It is a derivative of the S language created at the AT&T Bell laboratories. Apart from statistical analysis, R also supports powerful visualizations. It is open source, freely distributed under a General Public License ( GPL ). Comprehensive R Archive Network ( CRAN ) is an ever-growing repository with more than 8,400 packages, which can be freely installed and used for various kinds of analysis.

R은 인터프린터 기반의 직관적인 문법을 사용하는 고수준의 언어이다. R은 시스템 또는 서버의 메모리에서 실행되며, 실행 환경 내의 모든 파일, 함수, 그리고 파생된 결과들은 모두 객체로 저장된다. R이 실행되는 아키텍처는 그림 1.5에서 볼 수 있다.

R is a high-level language based on interpretations and intuitive syntax. It runs on system's or server's active memory (RAM) and stores all files, functions, and derived results in its environment as objects. The architecture utilized in R computation is shown in Figure 1.5:

<그림 1.5 : 로컬/서버 모드에서 R의 아키텍처>

Figure 1.5: Illustrate architecture of R in local/server mode

Installation of R

R 설치

R은 윈도우, 맥 OS X, 그리고 리눅스 등 모든 OS에 설치된다. 최신 버전의 설치 파일은 CRAN (<https://cran.r-project.org>) 및 미러 사이트들에서 다운로드 받을 수 있다. 또한 32비트와 64비트 아키텍처를 모두 지원한다.

R can be installed on any operating system, such as Windows, Mac OS X, and Linux. The installation files of the most recent version can be downloaded from any one of the mirror sites of CRAN: https://cran.r-project.org/ . It is available under both 32-bit and 64-bit architectures.

The installation of r-base-dev is also highly recommended as it has many built-in functions. It also enables us to install and compile new R packages directly from the R console using the install.packages() command.

Upon installation, R can be called from program files, desktop shortcut, or from the command line. With default settings, the R console looks as follows:

Figure 1.6: The R console in which one can start coding and evaluate results instantly

R can also be used as a kernel within Jupyter Notebook. Jupyter Notebook is a web-based application that allows you to combine documentation, output, and code. Jupyter can be installed using pip :

pip3 install --upgrade

pip pip3 install jupyter

To start Jupyter Notebook, open a shell/terminal and run this command to start the Jupyter Notebook interface in your browser:

jupyter notebook

To start a new R Notebook, click on right hand side New tab and select R kernel as shown in Figure 1.7. R kernel does not come as default in Jupyter Notebook with Python as prerequisite. Anaconda distribution is recommended to install python and Jupyter and can be downloaded from https://www.continuum.io/downloads . R essentials can be installed using below command.

conda install -c r r-essentials

Figure 1.7: Jupyter Notebook for creating R notebooks

Once notebook is opened, you could start organizing your code into cells. As R does not require formal compilation, and executes codes on runtime, one can start coding and evaluate instant results. The console view can be modified using some options under the Windows tab. However, it is highly recommended to use an Integrated Development Environment ( IDE ), which would provide a powerful and productive user interface for R. One such widely used IDE is RStudio, which is free and open source. It comes with its own server (RStudio Server Pro) too. The interface of RStudio is as seen in the following screenshot:

Figure 1.8: RStudio, a widely used IDE for R

Basic data types in R

R supports multiple types of data, which can be organized by dimensions and content type (homogeneous or heterogeneous), as shown in Table 1.1:

Table 1.1 Basic data structure in R

Homogeneous data structure is one which consists of the same content type, whereas heterogeneous data structure supports multiple content types. All other data structures, such as data tables, can be derived using these foundational data structures. Data types and their properties will be covered in detail in Chapter 3 , Linked Lists .

Operations in R

The syntax of operators in R is very similar to other programming languages. The following is a list of operators along with their explanations.

The following table defines the various arithmetic operators:

Table 1.2 Basic arithmetic operators in R

The following table defines the various logical operators:

Table 1.3 Basic logical operators in R

An example of assigned in R is shown below. Initially a column vector V is assigned followed by operations on column vector such as addition, subtraction, square root and log. Any operation applied on column vector is element-wise operation.

> V <- c(1,2,3,4,5,6) ## Assigning a vector V

> V

[1] 1 2 3 4 5 6

> V+10 ## Adding each element in vector V with 10

[1] 11 12 13 14 15 16

> V-1 ## Subtracting each element in vector V with 1

[1] 0 1 2 3 4 5

> sqrt(V) ## Performing square root operation on each element in vector V

[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490

> V1 <- log(V) ## Storing log transformed of vector V as V1

> V1

[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595

Control structures in R

Control structures such as loops and conditions form an integral part of any programming language, and R supports them with much intuitive syntax.

If condition

The syntax for the if condition is as follows:

if (test expression) {

Statement upon condition is true

}

If the test expression is true , the statement is executed. If the statement is false , nothing happens.

In the following example, an atomic vector x is defined, and it is subjected to two conditions separately, that is, whether x is greater than 5 or less than 5 . If either of the conditions gets satisfied, the value x gets printed in the console following the corresponding if statement:

> x <- 10

> if (x < 5) print(x) > if (x > 5) print(x) [1] 10

[ 19 ]

-----------------------------------------------------Page 31-----------------------------------------------------

Getting Started

If…else condition

The syntax for the if...else condition is as follows:

if(test expression) {

Statement upon condition is true

} else {

Statement upon condition is false

}

In this scenario, if the condition in the test expression is true , the statement under the if condition is executed, otherwise the statement under the else condition is executed. The following is an example in which we define an atomic vector x with a value 10 . Then, we verify whether x, when divided by 2, returns 1 (R reads 1 as a Boolean True). If it is True, the value x gets printed as an odd number, else as an even number:

> x=10

> if (x %% 2) + {

+ print(paste0(x, " :odd number")) + } else {

+ print(paste0(x, " :even number")) + }

[1] "10 :even number"

Ifelse function

An ifelse() function is a condition used primarily for vectors, and it is an equivalent form of the if...else statement. Here, the conditional function is applied to each element in the vector individually. Hence, the input to this function is a vector, and the output obtained is also a vector. The following is the syntax of the ifelse condition:

ifelse (test expression, statement upon condition is true, statement upon condition is false)

In this function, the primary input is a test expression which must provide a logical output. If the condition is true , the subsequent statement is executed, else the last statement is executed.

[ 20 ]

-----------------------------------------------------Page 32-----------------------------------------------------

Chapter 1

The following is example in which a vector x is defined with integer values from 1 to 7. Then, a condition is applied on each element in vector x to determine whether the element is an odd number or an even number:

> x <- 1:6

> ifelse(x %% 2, paste0(x, " :odd number"), paste0(x, " :even number")) [1] "1 :odd number" "2 :even number" "3 :odd number" [4] "4 :even number" "5 :odd number" "6 :even number"

For() loop

A for loop is primarily used to iterate a statement over a vector of values in a given sequence. The vector can be of any type, that is, numeric, character, Boolean, or complex. For every iteration, the statement is executed. The following is the syntax of the for loop:

for(x in sequence vector) {

Statement to be iterated

}

The loop continues till all the elements in the vector get exhausted as per the given sequence.

The following example details a for loop. Here, we assign a vector x, and each element in vector x is printed in the console using a for loop:

> x <- c("John", "Mary", "Paul", "Victoria") > for (i in seq(x)) { + print(x[i]) + }

[1] "John" [1] "Mary" [1] "Paul"

[1] "Victoria"

[ 21 ]

-----------------------------------------------------Page 33-----------------------------------------------------

Getting Started

Nested for( ) loop

The nested for loop can be defined as a set of multiple for loops within each for loop as shown here:

for( x in sequence vector) {

First statement to be iterated for(y in sequence vector)

{

Second statement to be iterated .........

} }

In a nested for loop, each subsequent for loop is iterated for all the possible times based on the sequence vector in the previous for loop. This can be explained using the following example, in which we define mat as a matrix (3×3), and our desired objective is to obtain a series of summations which will end up with the total sum of all the values within the matrix. Firstly, we initialize sum to 0 , and then subsequently, sum gets updated by adding itself to all the elements in the matrix in a sequence. The sequence is defined by the nested for loop, wherein for each row in a matrix, each of the values in all the columns gets added to sum:

> mat <- matrix(1:9, ncol = 3) > sum <- 0

> for (i in seq(nrow(mat))) + {

+ for (j in seq(ncol(mat))) + {

+ sum <- sum + mat[i, j] + print(sum) + } + }

[1] 1 [1] 5

[1] 12 [1] 14 [1] 19 [1] 27 [1] 30 [1] 36 [1] 45

[ 22 ]

-----------------------------------------------------Page 34-----------------------------------------------------

Chapter 1

While loop

In R, while loops are iterative loops with a specific condition which needs to be satisfied. The syntax of the while loop is as follows:

while (test expression)

{

Statement upon condition is true (iteratively)

}

Let's understand the while loop in detail with an example. Here, an object i is initialized to 1 . The test expression which needs to be satisfied for every iteration is i<10 . Since i = 1 , the condition is TRUE, and the statement within the while loop is evaluated. According to the statement, i is printed on the console, and then increased by 1 unit. Now i increments to 2, and once again, the test expression, whether the condition ( i < 10 ) is true or false, is checked. If TRUE, the statement is again evaluated. The loop continues till the condition becomes false, which, in our case, will happen when i increments to 10 . Here, incrementing i becomes very critical, without which the loop can turn into infinite iterations:

> i <- 1

> while (i < 10) + {

+ print(i)

+ i <- i + 1 + }

[1] 1 [1] 2 [1] 3 [1] 4 [1] 5 [1] 6 [1] 7 [1] 8 [1] 9

Special statements in loops

In R, the loops can be altered using break or next statements. This helps in inducing other conditions required within the statement inside the loop.

[ 23 ]

-----------------------------------------------------Page 35-----------------------------------------------------

Getting Started

Break statement

The syntax for the Break statement is break . It is used to terminate a loop and stop the remaining iterations. If a break statement is provided within a nested loop, then the innermost loop within which the break statement is mentioned gets terminated, and iterations of the outer loops are not affected.

The following is an example in which a for loop is terminated when i reaches the value 8 :

> for (i in 1:30) + {

+ if (i < 8) + {

+ print(paste0("Current value is ",i)) + } else {

+ print(paste0("Current value is ",i," and the loop breaks")) + break + } + }

[1] "Current value is 1" [1] "Current value is 2" [1] "Current value is 3" [1] "Current value is 4" [1] "Current value is 5" [1] "Current value is 6" [1] "Current value is 7"

[1] "Current value is 8 and the loop breaks"

Next statement

The syntax for a Next statement is next . The next statements are used to skip intermediate iterations within a loop based on a condition. Once the condition for the next statement is met, all the subsequent operations within the loop get terminated, and the next iteration begins.

This can be further explained using an example in which we print only odd numbers based on a condition that the printed number, when divided by 2 , leaves the remainder 1 .

> for (i in 1:10) + {

+ if (i %% 2) + {

+ print(paste0(i, " is an odd number.")) + } else { + next + }

[ 24 ]

-----------------------------------------------------Page 36-----------------------------------------------------

Chapter 1

+ }

[1] "1 is an odd number." [1] "3 is an odd number." [1] "5 is an odd number." [1] "7 is an odd number." [1] "9 is an odd number."

Repeat loop

The repeat loop is an infinite loop which iterates multiple times without any inherent condition. Hence, it becomes mandatory for the user to explicitly mention the terminating condition, and use a break statement to terminate the loop. The following is the syntax for a repeat statement:

repeat {

Statement to iterate along with explicit terminate condition including break statement

}

In the current example, i is initialized to 1 . Then, a for loop iterates, within which an object cube is evaluated and verified using a condition whether the cube is greater than 729 or not. Simultaneously, i is incremented by 1 unit. Once the condition is met, the for loop is terminated using a break statement:

> i <- 1 > repeat + {

+ cube <- i \*\* 3 + i <- i + 1

+ if (cube < 729) + {

+ print(paste0(cube, " is less than 729. Let's remain in the

loop."))

+ } else {

+ print(paste0(cube, " is greater than or equal to 729. Let's exit

the loop.")) + break + } + }

[1] "1 is less than 729. Let's remain in the loop." [1] "8 is less than 729. Let's remain in the loop." [1] "27 is less than 729. Let's remain in the loop." [1] "64 is less than 729. Let's remain in the loop." [1] "125 is less than 729. Let's remain in the loop."

[ 25 ]

-----------------------------------------------------Page 37-----------------------------------------------------

Getting Started

[1] "216 is less than 729. Let's remain in the loop." [1] "343 is less than 729. Let's remain in the loop." [1] "512 is less than 729. Let's remain in the loop." [1] "729 is greater than 729. Let's exit the loop."

First class functions in R

R is primarily a functional language at its core. In R, functions are treated just like any other data types, and are considered as first-class citizens. The following example shows that R considers everything as a function call. Here, the operator + is a function in itself:

> 10+20 [1] 30

> "+"(10,20) [1] 30

Here, the operator ^ is also a function in itself:

> 4^2

[1] 16

> "^"(4,2) [1] 16

Now, let's dive deep into functional concepts, which are crucial and widely used by R programmers.

Vectorized functions are among the most popular functional concepts which enable the programmer to execute functions at an individual element level for a given vector. This vector can also be a part of dataframe, matrix, or a list. Let's understand this in detail using the following example, in which we would like to have an operation on each element in a given vector V\_in . The operation is to square each element within the vector and output it as vector V\_out . We will implement them using three approaches as follows:

Approach 1 : Here, the operations will be performed at the element level using a for loop. This is the most primitive of all the three approaches in which vector allocation is being performed using the style of S language:

> V\_in <- 1:100000 ## Input Vector > V\_out <- c() ## Output Vector

> for(i in V\_in) ## For loop on Input vector + {

[ 26 ]

-----------------------------------------------------Page 38-----------------------------------------------------

Chapter 1

+ V\_out <- c(V\_out,i^2) ## Storing on Output vector + }

Approach 2 : Here, the vectorized functional concept will be used to obtain the same objective. The loops in vectorized programming are implemented in C language, and hence, perform much faster than for loops implemented in R ( Approach 1 ). The time elapsed to run this operation is instantaneous:

> V\_in <- 1:100000 ## Input Vector > V\_out <- V\_in^2 ## Output Vector

Approach 3 : Here, higher order functions (or nested functions) are used to obtain the same objective. As functions are considered first class citizens in R, these can be called as an argument within another function. The widely used nested functions are in the apply family. The following table provides a summary of the various types of functions within the apply family:

Table 1.4 Various types of functions in the apply family

Now, lets' evaluate the first class function through examples. An apply function can be applied to a dataframe, matrix, or array. Let's illustrate it using a matrix:

> x <- cbind(x1 = 7, x2 = c(7:1, 2:5)) > col.sums <- apply(x, 2, sum) > row.sums <- apply(x, 1, sum)

[ 27 ]

-----------------------------------------------------Page 39-----------------------------------------------------

Getting Started

The lapply is a first class function to be applied to a vector, list, or variables in a dataframe or matrix. An example of lapply is shown below:

> x <- list(x1 = 7:1, x2 = c(7:1, 2:5)) > lapply(x, mean)

The use of the sapply function for a vector input using customized function is shown below:

> V\_in <- 1:100000 ## Input Vector

> V\_out <- sapply(V\_in,function(x) x^2) ## Output Vector

The function mapply is a multivariate sapply . The mapply function is the first input, followed by input parameters as shown below:

mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = T, USE.NAMES = T)

An example of mapply to replicated two vector can be obtained as:

> mapply(rep, 1:6, 6:1)

The function call rep function in R with input from 1 to 6 and is replicated as 6 to 1 using the second dimension of the mapply function. The tapply applies a function to each cell of the ragged array. For example, let's create a list with a multiple array:

The output is a relationship between two vectors with position as a value. The function rapply is a recursive function for lapply as shown below:

> X <- list(list(a = pi, b = list(c = 1:1)), d = "a test") > rapply(X, sqrt, classes = "numeric", how = "replace")

The function applies sqrt to all numeric classes in the list and replace it with new values.

[ 28 ]

-----------------------------------------------------Page 40-----------------------------------------------------

Chapter 1

Exercises

1. Can you think of ways in which we can extract a few attributes (columns) and

observations (rows) based on a certain condition?

Dataset – El Nino dataset from UCI KDD ( https://kdd.ics.uci.edu /databases/el\_nino/el\_nino.html )

Filter for latitude and longitude with humidity > 88% and air temperature < 25.5 degree Celsius

10,000 iterations for evaluating each expression

2. Can we add multiple arguments within a function in the apply family? If yes,

what is the syntax for assigning multiple arguments?

3. A general notion states that for loops are slower than the apply functions. Is it true or false? If false, what are the conditions in which the notion gets negated? 4. Define ADT for calculating the area of any geometrical object, such as a circle,

square, and so on.

Summary

Computational power has been continuously increasing in the last couple of decades, and so does the amount of data captured by different industries. To cope with data size, faster and efficient information retrieval is an eminent requirement.

In this chapter, you were introduced to ADT and data structure. ADT is used to define high-level features and operations representing different data structures, and algorithms are used to implement ADT. A data type should be atomic, traceable, accurate, and have clear and concise characteristic properties for efficiency along with unambiguity. You also learned the basics of R, including data type, conditional loops, control structure, and first class functions.

The computational time taken by an algorithm is most important objective considered while selecting data structures and algorithms. The next chapter will provide the fundamentals for the analysis of algorithms.

[ 29 ]

-----------------------------------------------------Page 41-----------------------------------------------------

-----------------------------------------------------Page 42-----------------------------------------------------