```
knitr::opts_chunk$set(echo = TRUE)
```

Blockchains for Securing Transactions

The concept of *blockchain* originated with the pseudonomous Satoshi Nakamoto (whose identity is hotly debated to this day) in a whitepaper "Bitcoin: A Peer-to-Peer Electronic Cash System" in 2008. This paper proposed a decentralised electronic currency (Bitcoin) that runs on the blockchain. The idea spread from this point and can be applied to many other areas -- decentralised, incorruptible database of monetary transactions, contracts, un-hackable voting machines, and so forth. This section presents the technology underlying a blockchain through a simple sequence of R code chunks.

Assume your goal is to store some data in a secure way. You first store the data in a container -- called a block. In the case of BitCoin, each block contains several financial transactions. When there are new transactions (or there is new data) a new block will be created and will be combined with previous blocks to form a chain - the blockchain.

The "Block"

```
block_example <- list(index = 1,
                      timestamp = "2020-01-01 00.00 CST",
                      data = "a transaction",
                      previous_hash = 0,
                      proof = 9,
                      new_hash = NULL)
```

Before you can start building the blockchain -- chaining different containers with data together -- you need two core concepts: *hashing* and *proof-of-work-algorithms*.

Hashing

A *hash function* is a term used to loosely refer to checksums, check digits, fingerprints, lossy compression, randomization functions, error-correcting codes, and ciphers. True hash functions are used to index a hash table for scatter storage addressing, and the term has come to apply in security as well. Although storage and security concepts overlap to some extent, each one has its own uses and requirements and is designed and optimized differently. It might be more correctly termed a fingerprinting algorithm -- a procedure that maps an arbitrarily large data item (such as a computer file) to a much shorter bit string (its fingerprint) that uniquely identifies the original data.

A *hash value* (or just *hash*) generated by the hash function helps to ensure the integrity of a block by connecting it to the other blocks in the chain. A hash function takes input data and returns a unique, encrypted output. The hash guarantees the validity of a blockchain block by using a hashing algorithm (e.g., SHA256). The R `digest` package implements a function 'digest()' for the creation of hash values (digests)

of arbitrary R objects, using the 'md5', 'sha-1', 'sha-256', 'crc32', 'xxhash', 'murmurhash' and 'spookyhash' algorithms. This allows easy comparison of R language objects, and is applied here to secure the blockchain blocks.

```
    library("digest")

# generate a unique hash value 'fingerprinting' the author's name
    digest("j_christopher_westland" ,"sha256")
```

In the case of blockchains, the input data is the block (index,timestamp, data) to the hash function, but also the hash of the previous block. This means you can only calculate the valid hash if you know the hash of the previous block, which is created using the hash of the block before and so on and so forth -- thus block*chain*. This provides you with an immutable, sequential chain of blocks. If you would alter one block afterwards you would have to calculate all the hashes for the sequential blocks again. Here is an implementation.

```
    #Function that creates a hashed "block"
    hash_block <- function(block){
      block$new_hash <- digest(c(block$index,
                                 block$timestamp,
                                 block$data,
                                 block$previous_hash), "sha256")
      return(block)
    }
```

Proof-of-Work (PoW)

Where large amounts of information are stored in the blockchains, you need many blocks. But cryptocurrencies in particular need to control the number of new blocks created -- otherwise crypto-coins would lose their value were there an infinite number of coins created. Bitcoin, for example, is designed so that the maximum total amount of bitcoins that can ever exist is 21 million.

Blockchain works as a cryptocurrency because there is an valuable asset underlying each bitcoin that is mined -- the work required to generate the hash. For cryptocurrencies like BitCoin this could be a problem as the time to create a new block should me more or less constant (around 10 minutes in the case of BitCoin). Therefore the difficulty of PoW has to be adjusted continuously to account for increasing computational speed and varying numbers of miners in the network at a given time.

As a consequence, we add a so-called "Proof-of-Work" (PoW) algorithm which controls the difficulty of creating a new block. "Proof" means that the computer has performed a certain amount of work. In practice the goal is to create an item that is hard to create but easy to verify. The following code chunk uses

the following "task"as a PoW: find the next number that is divisible by 99 and divisable by the proof-number of the last block.

```
    ### Simple Proof of Work Alogrithm
    proof_of_work <- function(last_proof){

      proof <- last_proof + 1

      # Increment the proof number until a number is found that is
divisable by 99 and by the proof of the previous block
      while (!(proof %% 99 == 0 & proof %% last_proof == 0 )){
        proof <- proof + 1
      }
      return(proof)
    }
```

For blockchains like BitCoin or Ethereum the job of creating new blocks is done by so-called miners. When a new block has to be created, a computational problem is sent out to the network. The miner which solves the PoW problem first creates the new block and is rewarded in bitcoins (this is the way new BitCoins are actually created). This "lottery" of finding the new correct proof ensures that the power of creating new blocks is decentralised. When a new block is mined it is sent out to everybody so that every node in the network has a copy of the latest blockchain. The idea that the longest blockchain in the network (the one which "the most work was put into") is the valid version of the blockchain is called "decentralised consensus".

In the case of BitCoin the PoW problem involves the problem of finding numbers that generate hashes with a certain amount of leading zeros. To account for increasing computational speed and varying numbers of miners in the network the difficulty of the PoW can be adjusted to hold the time to create a new block constant at around ten minutes.

Adding transactions (new blocks) to the blockchain

A blockchain is a growing list of transaction records (blocks) that are linked using cryptography (the hash). Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data (generally represented as a Merkle tree). By design, a blockchain is resistant to modification of the data, and provides an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way. In cryptocurrency and other applications, it is important that creating new blocks (such as creation of a new Bitcoins) be expensive, so that the currency has some value. The following code chunk shows how blocks are chained together using hashes and how the cost of creating new blocks is being regulated by PoWs.

```
    #A function that takes the previous block and normally some data (in
our case the data is a string indicating which block in the chain it is)
```

```
    gen_new_block <- function(previous_block){

      #Proof-of-Work
      new_proof <- proof_of_work(previous_block$proof)

      #Create new Block
      new_block <- list(index = previous_block$index + 1,
                        timestamp = Sys.time(),
                        data = paste0("this is block ",
previous_block$index +1),
                        previous_hash = previous_block$new_hash,
                        proof = new_proof)

      #Hash the new Block
      new_block_hashed <- hash_block(new_block)

      return(new_block_hashed)
    }
```

The first block in a blockchain is the so-called *genesis* block, containing no data and arbitrary values for proof and previous hash (as there is no previous block).

```
    # Define Genesis Block (index 1 and arbitrary previous hash)
    block_genesis <-  list(index = 1,
                           timestamp = Sys.time(),
                           data = "Genesis Block",
                           previous_hash = "0",
                           proof = 1)
```

As each new transaction is processed the blockchain grows by adding new blocks

```
## Build the blockchain, starting with the Genesis block and then add a few
blocks using a loop.

    blockchain <- list(block_genesis)
    previous_block <- blockchain[[1]]

    proof <- NULL
      # How many blocks should we add to the chain after the genesis block
      num_of_blocks_to_add <- 10

      # Add blocks to the chain
      for (i in 1: num_of_blocks_to_add){
        block_to_add <- gen_new_block(previous_block)
        blockchain[i+1] <- list(block_to_add)
```

```
        previous_block <- block_to_add

        print(cat(paste0("Block ", block_to_add$index, " has been added",
"\n",
                  "\t", "Proof of Work: ", block_to_add$proof, "\n",
                  "\t", "Hash Value: ", block_to_add$new_hash)))

        proof[i] <- block_to_add$proof

      }
```

This is how one block in the chain looks.

```
    blockchain[[10]]
```

Note that the cost of generating new blocks (the"Proof of Work") grows exponentially.

```
library(tidyverse)

proof_work <- data.frame(1:length(proof),proof)
colnames(proof_work)[1] <- "cycle"

ggplot(proof_work, aes(cycle, proof))+
  geom_line() +
  xlab("Block Number") +
  ylab("Cost of New Block Generation")
```

That's it! Reading vendor and industry hype surrounding blockchain would lead one to believe that the technology is much more complex. Indeed, implementing markets and commercial applications can become involved, but the underlying concepts are easy to implement and simple to understand.

Forensic Analytics: Benford's Law

Benford's Law concludes that the first digit in a large population of transactions (10,000 plus) will most often be a 1. Less frequently will the first digit be a 2; even less frequently a 3. In general the probability the the most significant digit of a number is equal to $d_1 \in [1,2,...,9]$ is $Pr(d_1) = log_{10}(1+ \frac{1}{d_1})$

```
library(tidyverse)
```

```
prd1 <- data.frame("digit"=1:9, "Pr_digit"=log10(1+(1/{1:9})))

ggplot(prd1, aes(digit, Pr_digit)) +
  geom_bar(stat="identity") +
  xlab("Leading digit") +
  ylab("Probability of leading digit")
```

An analysis of the frequency distribution of the first or second digits can detect abnormal patterns in the data and may identify possible fraud. An even more focused test can be used to examine the frequency distribution of the first two digits, first three digits and so forth. Some audit software programs can be used to determine the frequency distribution for first digits, first two digits, and second digits.

This ‘law’ has been found to apply to a wide variety of data sets, including electricity bills, street addresses, stock prices, population numbers, death rates, lengths of rivers, physical and mathematical constants, and processes described by power laws (which are very common in nature). It tends to be most accurate when values are distributed across multiple orders of magnitude. But not all data will have distributions as predicted by Benford's Law. Sometimes there is valid rationale for certain numbers occurring more frequently than expected. For example, if a company sends a large amount of correspondence via courier, and the cost is a standard rate ($6.12) for sending a package of under one pound, then the first digit (6) or the first two digits (61) may occur more often than predicted by Benford's Law.

Benford's law can only be applied to data that are distributed across multiple orders of magnitude. Generally, if there is any cut-off which excludes a portion of the underlying data above a maximum value or below a minimum value, then the law will not apply. Thus, real-world distributions that span several orders of magnitude rather smoothly (e.g. populations of settlements, provided that there is no lower limit) are likely to satisfy Benford's law to a very good approximation. On the other hand, a distribution that covers only one or two orders of magnitude (e.g. heights of human adults, or IQ scores) is unlikely to satisfy Benford's law.

The following code chunks apply Benford's Law analysis to the 'expenditures' file generated in the last chapter of this book.

```
library(tidyverse)
library(benford.analysis)

# Read the employee expenditures file

expenditures <-
  read.csv("~/audit_analytics_book/audit_simulated_files/expenditures.csv",
na.strings="0", stringsAsFactors=FALSE)

hist(expenditures$amount)

lead_digit <- extract.digits(expenditures$amount, number.of.digits=1)
lead_digit <- lead_digit[,2]
```

```
hist(lead_digit, breaks=9)

ben <- benford(expenditures$amount, number.of.digits=2)

plot(ben)
```

The results of this analysis reveals that the digits 49 were in the data more often than expected.

```
library(tidyverse)
library(benford.analysis)

lead_digit <- extract.digits(expenditures$amount, number.of.digits=2)
employee_check <- cbind(expenditures,lead_digit)

employee_check[employee_check$data.digits ==49,] %>%
  arrange(employee_no) %>%
  select(employee_no, date, amount)
```

Classifying on the employee for all contracts with 49 as the first two digits determined several employee who submitted an excessive number of expense reports for $499 (perhaps to avoid regulations required purchases over $500 to have a supervisor's signature). The auditor could then use this information to initiate further investigation of potential abuse of corporate control systems.