STAT40780 Data Programming with C (online)

Lab Sheet 1

This week's lab will involve profiling slow R code to identify bottlenecks, writing an improved/more efficient version of the code, and benchmarking the improved version against the original.

1 Profiling R code

The R code on the next page counts the odd numbers in each row of an integer matrix. The function `countOdds()` takes an integer matrix as input, and returns a vector of counts of the odd numbers for each row of the matrix. The function `countOdds()` makes calls to the function `vecOdds()` to count the odd numbers in a given row (vector). The `vecOdds()` function in turn passes each element of the row (vector) to the `isOdd()` function - if the element is odd it returns a value of 1, and returns 0 otherwise. This code contains nested `for` loops - best avoided in R.

You may not have come across the modulus operator in R, %%, that returns the remainder from the division of two integer values. For example, 10 %% 2 returns 0, while 9 %% 2 returns 1.

Step 1: Have a look through this code and ensure you understand how it works, before proceeding.

Step 2: Create a new R script in RStudio. Then copy and paste the below R code into your R script. Save your script file as lab1.R.

Counts of odd numbers in each row of a matrix

```
countOdds <- function( X ){

  #type checking
  if( !is.matrix(X) ) stop("input is not a matrix")
  if( !is.integer(X) ) stop("input should be of type integer")

  N <- nrow(X) #number of rows of X
  numOdds <- rep(0, N ) #vector to store counts; initialize to zero

  for( i in 1:N ){ #for each row i in X
      numOdds[i] <- vecOdds( X[ i, ] ) #count odds in row i
  } #end of for

  return(numOdds) #return the vector of counts

} #end of countOdds()


#count the number of odd numbers in a vector y
vecOdds <- function(y){

   K <- length(y) #length of the vectors (row)
   #keep count of number of odd numbers in y
   sumOdds <- 0 #initialize counter to 0 (no odd numbers)

   for(j in 1:K){ #for each element of the vector y
     sumOdds <- sumOdds + isOdd( y[j] ) #update sumOdds
   }
   return(sumOdds)
} #end of vecOdds()


#function that checks whether its input is an odd number
#returns 1 if input is odd and 0 otherwise
isOdd <- function(num){
  if( num %% 2 ){
    return(1)
  }else{
    return(0)
  }
}
```

Step 3 Create an integer matrix to input to the R function.

Generate data from a binomial distribution, with number of events = 30, and probability of success = 0.5. This will generate integer data valued between 0 and 30. Make a barchart of your data to see what it looks like. Then arrange your random data into a matrix with 100000 rows.

Use the following R code to do this:

Generate a matrix of integer values

```
#generate 10000000 data points from a binomial distribution
simdata <- rbinom(n=10000000, size=30, prob = 0.5)

#check what the simulated data look like
barplot(table(simdata))

#arrange simulated data into a matrix with 100000 rows
X <- matrix(simdata, nrow= 100000)
dim(X) #print dimensionts of matrix
```

Call the countOdds() function

```
countOdds(X) #call to the countOdds() function
```

Step 4 Profiling.

This is the main task of this section of the lab. Profile the R function `countOdds()` to identify bottlenecks.

2 Benchmarking code

Step 1: Using the results of your profiling as a guideline, write a more efficient version of the `countOdds()` function - be sure to give the revised function a different name! Hint: for improving efficiency, make use of the `apply()` function.

Step 2: Benchmark your code. Compare the original `countOdds()` function to your (hopefully) improved version. How much faster is your new function, relative to the original? Hint: use the `Rprof()` function in R. Hint: use the `Rprof` function.

You can experiment with specifiying the option `line.profiling=TRUE` to instruct Rprof to profile code by line, rather than by function/operation.