Scientific Programming Assignment 2

Changmin Yu

November 2017

1 Sujiko problems

Sujiko is a logic-based, combinatorial number-placement puzzle created by Hai Gomer of Kobayaashi Studios.
The puzzle takes place on a 3×3 grid with four circled nnumber clues at the centre of each quandrant which indicates the sum of the four numbers in that quadrant. The numbers 1-9 must be placed in accordance with the circled clues, to complete the puzzle. A typical example of the Sujiko problem is shown as below:

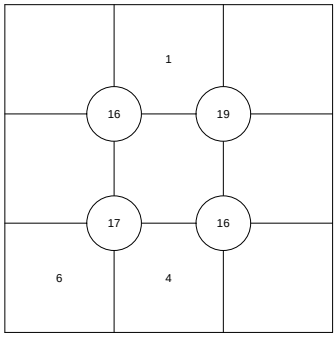

Figure 1: Typical Sujiko example

1.1

In this part, we are asked to write the function "drawgrid" which takes three inputs: the four sums in the circles , any numbers in the squares and the title of the plot. To do this, firstly one need to check that the input arguments are of the correct data type, where the type of the first two arguments are *double* and the third argument is of type *character*, if these conditions are not met, return *FALSE*.
After meeting all the conditions for the inputs, we can now starting plotting. We are required to generate the 3×3 grids with circles at the centres of the quadrants and the numbers in the grids as instructed. We firstly generate the grids, we do this by generating a new plot without showing anything, including the outter frame, the x-axis, the y-axis and the data points. We can do this by using $plot()$ function with a few parameters specified as following R command:

```
plot(c(-2,1),c(-2,1),type='n',bty='n',xaxt='n',yaxt='n',ann=FALSE,asp=1,cex=10)
```

As we can see, we specified *type*, *bty*, *xaxt* and *yaxt* parameters as 'n's,, *ann* pararmeter as *FALSE*, *asp* to 1 and *cex* to 10. Then we can have a blank plot with nothing shown in it, but actually is a rectangular space,

x ranges from -2 to 1 and y ranges from -2 to 1. Then, we can use $segments()$ function, which takes input of x_0, y_0, x_1, y_1 as the starting and ending x and y coordinates of the line segments, so then I simply draw eight of such line segments to produce the grids.
Then, we can use $symbols()$ to generate the circles at the centres of the quandrants. In my case, the centres of the circles are (-1, 0), (0, 0), (-1, -1), (0, -1), hence I simply put these into the function, and specifices that $circles = 0.25$, $inches = FALSE$, $add = TRUE$ and $bg =' white'$ to add the four white circles with radii all equal to scaled 0.25 to this existing grid.
Then I simply use the $text()$ function to fill the circles and grids with the information of the input vectors in the locations at which the specified x and y coordinates. Note that in the second input, the vector contains the information of the numbers fill the grids, whenever there is a zero, we just simply leave that grid as blank. Also note that, when we are filling the girds, we need to make sure that the order of the numbers given in the input vector fits well into the grids, to do this, I simply transform the vector to a 3×3 matrix where row 1 consists of the first three numbers, row 2 consists of the middle three numbers and row 3 consists of the last three numbers.
The process of the above workflow can be visualized in the following plots:

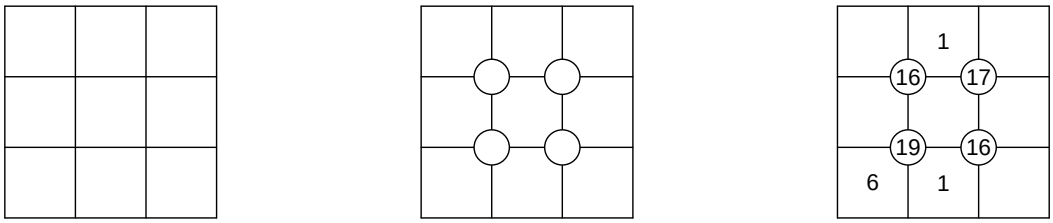

Figure 2: Workflow for generating the grids

So after putting all these into a function $drawgrid()$, we can actually now reproduce the three grids shown in the instructions:

```
par(mfrow=c(1, 3), mar=c(2, 2, 1.5, 1))
drawgrid(c(16, 19, 17, 16), c(0, 1, 0, 0, 0, 0, 6, 4, 0), main='A')
drawgrid(c(16, 11, 21, 21), c(8, 0, 2, 0, 0, 0, 0, 0, 0), main='B')
drawgrid(c(18, 13, 18, 18), c(0, 0, 0, 0, 0, 0, 0, 0, 0), main='C')
```

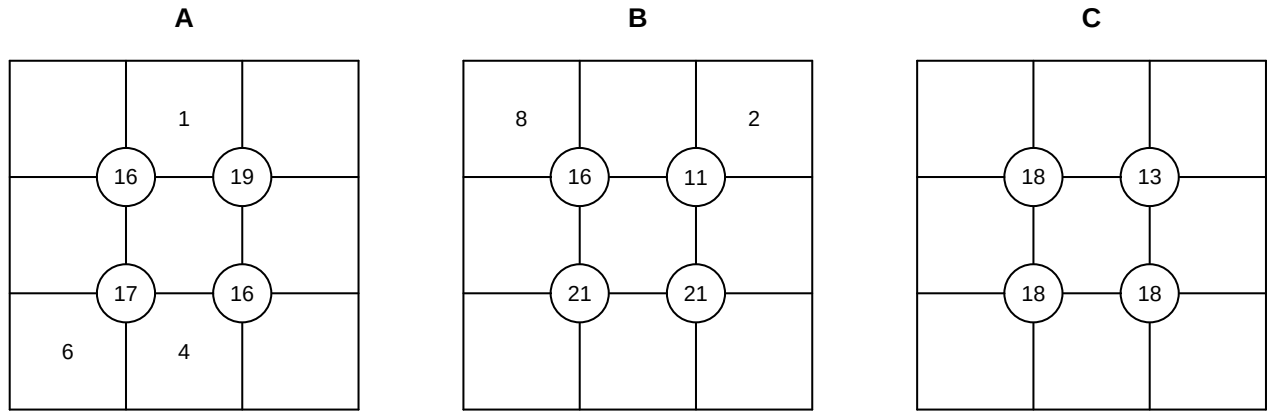

Figure 3: Reproducing the graphs of the three sujiko problems to be solved

You may find the algorithm for the function $drawgrid()$ written in R in the appendix.

1.2

In this part, we are asked to write the function $sujiko()$ which takes two inputs: the vector of the sums in the four circles, and the available numbers in the squares in the grid. The good news is that since within the *sujiko* problem, to search for the final solution, the search space is acutally quite small, hence we could actually use exhaustive search on this problem.
Firstly, as before, I have to check the data type of the input arguments and return *FALSE* is the inputs are not of the correct data type. Then, thanks to the very helpful function $permutations()$ provided in the Hints, which takes an input of a natural number and returns the list of all the permutations of the natural numbers from 1 up to n in a matrix A. For instance, $permutations(2)$ is going to return [1, 2] and [2, 1] vertically stacked into a matrix.
There are two cases. The first case is when the input vector *squares* is just a vector of zeros, we can do the complete exhaustive search of $permutations(9)$ and loop through all the 9! permutations, then by checking the correponding sums of the four quandrants in each permutation, we can eventually find one permutation of which the sums of the four quandrants all equal to the four numbers in the circles respoectively. Since the search space is fairly small, despite this is the most exhaustive search we are having in this question, it does not take more than 1 second to finish the search, hence this method is completely valid.
The second case is when the input vector *squares* is not a vector of zeros. In this situation, we can actually get rid of the non-zero entries in the squares vector and hence reduce the search space by only use $permutations()$ on the index of the remaining entries. E.g., if our squares vector is $c(0,1,0,0,0,0,6,4,0)$, we can first get rid of the non-zero entries, and we get a list [2, 3, 5, 7, 8, 9], then do $permutations(6)$, we can then loop through all the 6! permutations and then get the corresponding four sums using the original *squares* vector and match them to the four sums in the *sums* vector. The final output of the *sujiko* function is simply a list of non-repetitve numbers 1 to 9 which given our corresponding relation between the entry index and the location of the number in the grids, would satisfy the conditions that the sum of the quandrants are exactly the correponding numbers in the sums in the circles.
You may find the complete algorithm of function $sujiko()$ written in R in the appendix.

1.3

Given our effort in the first two parts, part 3 is very easy to solve. We simply feed the input vectors specified in the instructions to obtain the completed sujiko grids. The solved sujiko plots are as following:

```
par(mfrow=c(1, 3), mar=c(2, 2, 1.5, 1))
sq.1 <- sujiko(c(16, 19, 17, 16), c(0, 1, 0, 0, 0, 0, 6, 4, 0))
sq.2 <- sujiko(c(16, 11, 21, 21), c(8, 0, 2, 0, 0, 0, 0, 0, 0))
sq.3 <- sujiko(c(18, 13, 18, 18), c(0, 0, 0, 0, 0, 0, 0, 0, 0))
drawgrid(c(16, 19, 17, 16), sq.1, main='A')
drawgrid(c(16, 11, 21, 21), sq.2, main='B')
drawgrid(c(18, 13, 18, 18), sq.3, main='C')
```

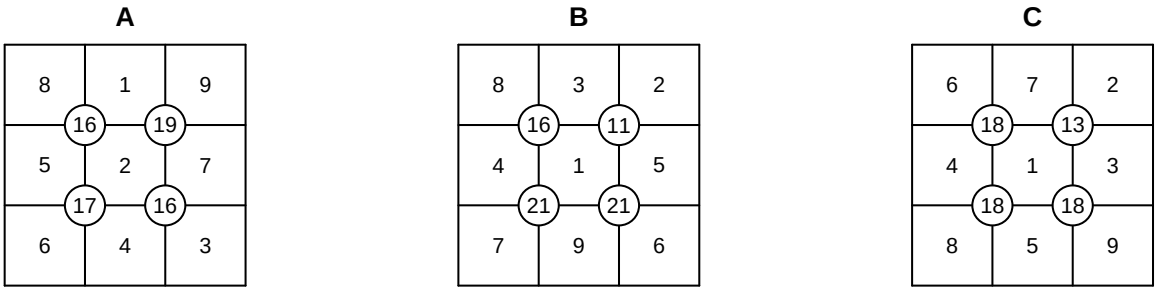

Figure 4: Solved sujiko problems

2 Search

In this question, we are asked to solve the Joe's pyramid. Given the instructions, we are actually looking for a sequence of distinct one or two digit positive numbers that could be fill in stones and followingthe rule, go all the way up to the top stone.

2.1

If we are not considering the constraints and global search, there are actually ${}^{99}P_6 = \frac{99!}{(93)!} = 806781064320$ ways of numering the six blocks on the bottom row, but note that our pyramid is identical up to symmetry, i.e., A-B-C-D-E-F is the same as F-E-D-C-B-A, hence we need to devide the above number by 2 we have that there are $\frac{{}^{99}P_6}{2} = 403390532160$ ways for numbering the bottom six blocks.

2.2

In this part, we are asked to explain the restrictions for each of the six numbers on the bottom row. Firstly, we define the six numbers on the bottom row to be A, B, C, D, E, F respectively. Then following the rules of the Joe's pyramid, we have that numbers on the second bottom row are A+B, B+C, C+D, D+E, E+F respectively. Similarly, we have that the numbers on the thrid bottom row are A+2B+C, B+2C+D, C+2D+E, D+2E+F. ... And finally, we have that the number in the top block is A+5B+10C+10D+5E+F. We know that all numbers in the pyramid must be one or two digit positive numbers, hence is less than 100. Also, since every stone is marked with a different one or two digit positive number, we must have that all numbers in the blocks are different. We can define a naive restrictions on each of the 6 numbers on the bottom row by taking the sum of all the other terms in the top block to be minimum, i.e., if we are looking for the restriction for C, we take D = 1, B = 2, E = 3, A = 4, F = 5, hence we have that $A+5B+10C+10D+5E+F < 100 \Rightarrow 10C < 58 \Rightarrow C \leq 5$ and we have the same argument and restriction for D. Similarly, we could deduce the aalagous arguments for A, B, E and F, which are $A, F \leq 30$ and $B, E \leq 11$. Another restriction is that every block must contain different numbers, but this is very hard to include in the naive definition of the restrictions for the six numbers, I will include these restrictions in part 3.

2.3

In this part, we are going to write and run a function that uses these restrictions to sesarch for a solution and the number X in the top block. We first create a function $sum.step()$ which takes a vector as input and treat as a row in Joe's pyramid and computes the list of numbers in the row above the input vector and returns this vector as the output. The function is a recursive function where when the input is a vector of length less than two simply returns an empty list.
And we also need a second function $check()$ for checking all the restrictions listed in the rules which takes a vector as input and returns Boolean values TRUE if this vector can be a row in the Joe's pyramid and FALSE if otherwise. Within this function, we apply a while-loop conditions on the length of the vector, say s, larger than 1. Within the while-loop we compute $s2 = sum.step(s)$ which is the expected row above s following the rules. Then we need to check, if the maximum element in s2 is larger than 99, we need to return FALSE. Then we update s by concatenating s and s2 by using $s < -c(s, s2)$, then if the length of the unique list of the updated vector s is less than the sum of the lengths of original s and s2, we know that there must be repetition(s) in the update, hence return FALSE since we must have different numbers in all blocks. Then assign the value of s back to s2 for next iteration. Also concatenate the vector in each step to a vector s4, and after we break from the while-loop, we need to check that if s4 == unique(s4) since without checking this, we might have same numbers in two non-adjacent rows.
Now we have the $check()$ function, we can now apply $check()$, $sum.step()$ and our naive restrictions defined in part 2 to search for a solution. Given the naive restrictions, I decide to apply six for-loops (which is very expensive but I have not found a way to reduce the computational cost) to search for the solution. I created

a function *pyramid*() which takes a postive integer which indicates how many rows in Joe's pyramid as input, then apply our restrictions. I start with loop through the values C and D can take, within the loop of C, we then choose an integer between 1 and 5 for C, then we choose an integer not equal to C but between 1 and 5 for D, then we choose an integer B not equal to C or D but between 1 and 11, now we need to apply *check*() function to the list (B, C, D), if it returns false, we use *next* to halt the process of the current iteration and advances the loop index. Then similarly for the loop of E, A and F. Finally, after we reach to a valid vector of integers, l, at the end of the six loops, it is then a valid list of integers that could fill the bottom row of the pyramid. Then, we use a while-loop condition on the length of the list l larger than zero for reporting the solution in each row of the pyramid. And also, for convenience, I store the six lists in a 6×6 matrix, which eventually will be an upper triangular matrix which shows a upside down version of the Joe's pyramid.
Note that, if we use a function describes as above, we would have achieved to two answers, which are exactly the symmetry to each other, hence these two answers can be considered as one solution to the pyramid, since by intuition, Joe's pyramid is symmetric, hence if we have one solution, its symmetry naturally becomes a valid solution. This problem can be solve by taking $D > C$ in its loop since we know that this problem globally only gives us one unique solution. This will be shown in the complete algorithm of the function *pyramid*() in the appendix. The output of this function is as following:

```
A <- pyramid(6)

## [1]  8  6  1  3  2 10
## [1] 14  7  4  5 12
## [1] 21 11  9 17
## [1] 32 20 26
## [1] 52 46
## [1] 98

A

##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    8    6    1    3    2   10
## [2,]   14    7    4    5   12    0
## [3,]   21   11    9   17    0    0
## [4,]   32   20   26    0    0    0
## [5,]   52   46    0    0    0    0
## [6,]   98    0    0    0    0    0
```

As we can see from the output, X = 98 is the unique solution for the top block.
You may find the complete algorithms for functions *sum.step*(), *check*() and *pyramid*() written in R in the appendix.

2.4

In this part, we are asked to draw the correct number within each stone. I write a function *solve.pyramid*() which takes a positive integer n as input which can be feed to *pyramid*() to generate the list of unique solution. Then by using *plot*(), *segments*() and *symbols*() we could draw the blocks for Joe's pyramid by similar constructions used in plotting Sujiko problems in Question 1. Then we use a for loop to loop through the matrix which is the output of *pyramid*(n) and find a one-to-one mapping between the matrix indices of the numbers in the matrix to the position of the number in the pyramid, which is also quite similar to the way we fill up the solved Sujiko problem in Question 1.
The plot for the final solution (up to symmetry) of Joe's pyramid is as following: (You may find the complete algorithm of function *solve.pyramid*() written in R in the appendix.)

```
solve.pyramid(6)
```

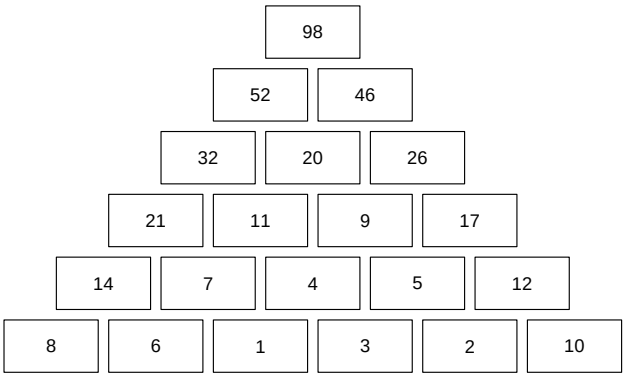

Figure 5: Solved Joe's pyramid

3 Logistic map

In this question, our first job is to reproduce the logistic maps as shown in the original paper by (May, 1976). After reading the paper, we could now first define the logistic map in our own terms. Given the logistic equation $x_{t+1} = \alpha x_t(1 - x_t)$, and given different α values, the sequence of $\{x_t\}_{t=0,1,2,...}$ will eventually either get trapped in a steady state or will reach a bifurcation point. When this doubling event occurs, it can be indicated by the derivative at the fixed point of the logistic equation, if the derivative is between -1 and 1, the fixed point is a (locally) stable state and unstable otherwise.
Hence, now we can start reproducing the logistic map. We first generate a list of equally distanced α values between 2.8 and 3.6, for each α value we start from a random number x between 0 and 1, note that we have to restrict x to [0, 1] since if x exceeds unity, the sequence will almost surely diverge to $-\infty$. After we have assigned a random number between 0 adn 1 to x, we can then start iterating the sequence and hence if we set the number of iterations enough we will be able to obtain a relatively reliable asymptotic dynamical property of the sequence and the logistic map. Then for the plot, if we have 1000 iterations for each α value, we then only need the last 200 values since 1000 iterations is a rather appropriate number of iterations since this can ensure the sequence is already trapped in the steady state if it has one and the sequence has not reach more bifurcations points after the first bifurcation occured if it has no steady state.
After we have obtained the stable/unstable values after applying the function *logistic.map* to the sequence of α values using *sapply* by assigning any random number x between zero and one, we can then use them to reproduce the plot as required, to do this, we simply need to use $rep()$ to generate a vector of 201 copies of the α sequence and then use $sort()$ to group them in ascending order. Then we could plot our output from $logistic()$ against the generated repeated α vector using simply $plot()$. Note that if we want to generate the exact same logistic maps as in the paper, we need to specify the *pch* parameter to be the ASCII character '.'.

You may find the complete algorithm of the function $logistic.map()$ written in R in the appendix.

```
par(mar=c(4,4,.1,.1),cex.lab=.95,cex.axis=.9,mgp=c(2,.7,0),tcl=-.3, mfrow=c(1, 1))
alpha <- seq(2.8, 3.6, length.out = 300)
logistic <- sapply(alpha, logistic.map, x = runif(1))
alpha.plot <- sort(rep(alpha, 201))
plot(alpha.plot, logistic, pch = '.', bty = 'n', xlab = expression(paste(alpha)),
     ylab = paste('steady-state','x'))
```

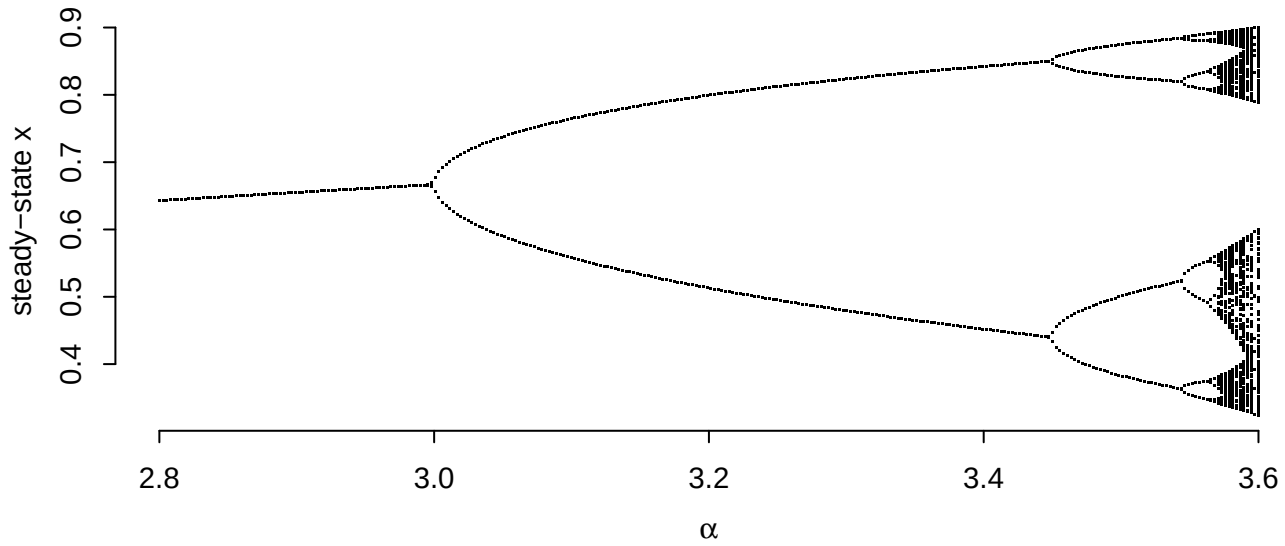

Figure 6: Reproducing Logistic map: $x_{t+1} = \alpha x_t(1-x_t)$, as per figur 4 of (May, 1976)

Now we are going to estimate when the doubling occurs based on the numerical calculations. We could use similar argument as before, we create a function $doub()$ which takes a sequence of α values and a random number x for initial number as inputs, then we borrow the bulk in the $logistic.map()$ function to generate the last 201 elements in the sequence. At the start of teh function, we define a variable $ind = 1$ which is the initial number of steady states, then through iteration, if there is an α value that gives more than the ind value number of unique steady state x values rounding up to 1 decimal places, we put the α values into a vector and this is our output, and then we can add vertical dotted lines to the existing logistic map to visually illustrate when doubling occurs. The plot is shown as following:

```r
par(mar=c(4,4,.1,.1),cex.lab=.95,cex.axis=.9,mgp=c(2,.7,0),tcl=-.3, mfrow=c(1, 1))

alpha <- seq(2.8, 3.6, by = 0.001)
doubling.list <- doub(alpha, 0.1)
print(doubling.list)

## [1] 3.003 3.449 3.451 3.558 3.559

alpha <- seq(2.8, 3.6, length.out = 300)
logistic <- sapply(alpha, logistic.map, x = runif(1))
alpha.plot <- sort(rep(alpha, 201))
plot(alpha.plot, logistic, pch = '.', bty = 'n', xlab = expression(paste(alpha)),
     ylab = paste('steady-state','x'))

for (i in 1:length(doubling.list)){
  abline(v = doubling.list[i], lty = 2)
}
```

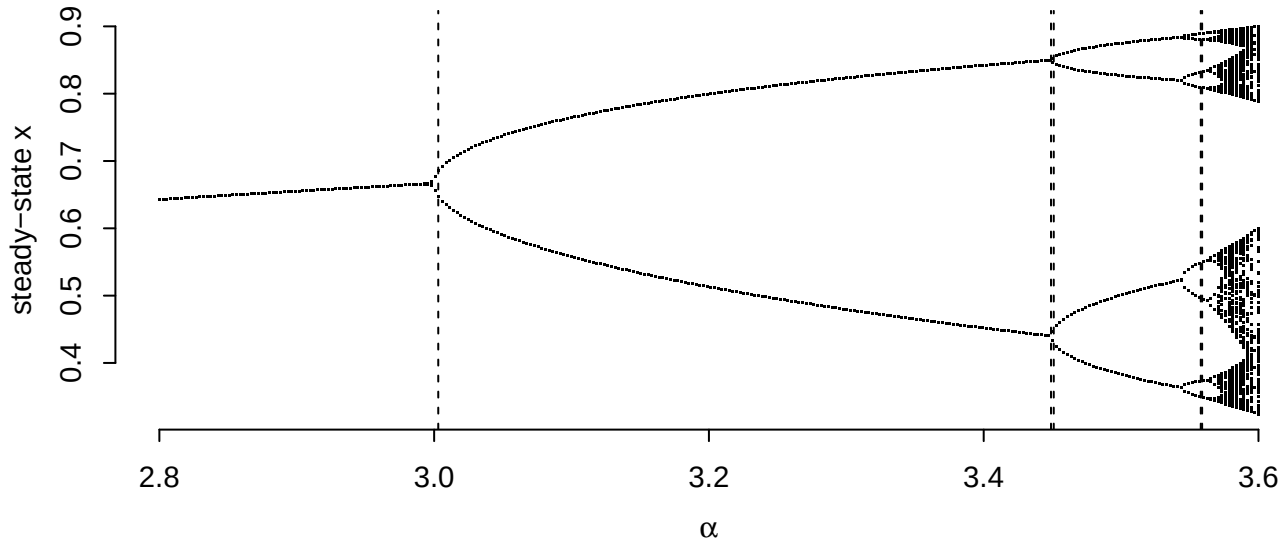

Figure 7: Logistic map with the indication of when doubling occurs

As we can see from the plot and the reported *doubling.list* values, we can roughly estimate that there are doublings occur at α around 3, 3.45 and 3.58 and after α reaches around 3.58, the the process becomes really chaotic and begins to become highly unstable, hence it is very hard to compute when the doubling begins at this stage since the sequences become highly messy and hard to detect when the doubling occurs.

You may find the complete codes for the various functions and scripts for Question 3 written in R in the appendix.

A *drawgrid*

```
drawgrid <- function(sums, n, main=''){
  if (typeof(sums)!='double'||typeof(n)!='double'||typeof(main)!='character'){
    return(FALSE)
  }
  else{
    plot(c(-2,1),c(-2,1),type="n",bty='n',xaxt='n',yaxt='n',ann=FALSE,asp=1,cex=10)
    segments(-2, 1, 1, 1)
    segments(-2, -2, 1, -2)
    segments(-2, -1, 1, -1)
    segments(-2, 0, 1, 0)
    segments(-2, 1, -2, -2)
    segments(-1, 1, -1, -2)
    segments(0, 1, 0, -2)
    segments(1, 1, 1, -2)

    symbols(-1, 0, circles = 0.25, inches = FALSE, add = TRUE, bg = 'white')
    symbols(-1, -1, circles = 0.25, inches = FALSE, add = TRUE, bg = 'white')
    symbols(0, 0, circles = 0.25, inches = FALSE, add = TRUE, bg = 'white')
    symbols(0, -1, circles = 0.25, inches = FALSE, add = TRUE, bg = 'white')

    text(-1, 0, labels = sums[1])
    text(0, 0, labels = sums[2])
    text(-1, -1, labels = sums[3])
    text(0, -1, labels = sums[4])

    if (length(n) != 9){
      return(FALSE)
    }
    else{
      N <- matrix(n, 3, 3, byrow=TRUE)
      for (i in 1:3){
        for (j in 1:3){
          if (N[i, j] != 0){
            text(j-2.5, 1.5-i, labels = N[i, j])
          }
        }
      }
    }
    title(main=main, line = -3)
    return(plot)
  }
}
par(mfrow=c(1, 3), mar=c(2, 2, 1.5, 1))
drawgrid(c(16, 19, 17, 16), c(0, 1, 0, 0, 0, 0, 6, 4, 0), main='A')
drawgrid(c(16, 11, 21, 21), c(8, 0, 2, 0, 0, 0, 0, 0, 0), main='B')
drawgrid(c(18, 13, 18, 18), c(0, 0, 0, 0, 0, 0, 0, 0, 0), main='C')
```

B *sujikio & permutations*

```
sujiko <- function(sums, squares){
  if (typeof(sums)!='double'||typeof(squares)!='double'||
      length(sums)!=4||length(squares)!=9){
    return(FALSE)
  }
  else{
    num <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
    if (identical(squares, rep(0, 9))){
      per <- permutations(9)
      for (i in seq.int(factorial(9))){
        squares.1 <- per[i,]
        bool.1 <- (squares.1[1]+squares.1[2]+squares.1[4]+squares.1[5])==sums[1]
        bool.2 <- (squares.1[2]+squares.1[3]+squares.1[5]+squares.1[6])==sums[2]
        bool.3 <- (squares.1[4]+squares.1[5]+squares.1[7]+squares.1[8])==sums[3]
        bool.4 <- (squares.1[5]+squares.1[6]+squares.1[8]+squares.1[9])==sums[4]
        bool.final <- bool.1&&bool.2&&bool.3&&bool.4
        if (bool.final){
          break
        }
      }
      return(squares.1)
    }
    else{
      nonzerosq <- squares[which(squares!=0)]
      num <- num[-nonzerosq]
      per <- permutations(length(num))
      for (i in seq.int(length(per)/length(num))){
        num.1 <- num[per[i,]]
        squares.1 <- squares
        for (j in 1:9){
          if (squares.1[j]==0){
            squares.1[j] <- num.1[1]
            num.1 <- num.1[-1]
          }
        }
        bool.1 <- (squares.1[1]+squares.1[2]+squares.1[4]+squares.1[5])==sums[1]
        bool.2 <- (squares.1[2]+squares.1[3]+squares.1[5]+squares.1[6])==sums[2]
        bool.3 <- (squares.1[4]+squares.1[5]+squares.1[7]+squares.1[8])==sums[3]
        bool.4 <- (squares.1[5]+squares.1[6]+squares.1[8]+squares.1[9])==sums[4]
        bool.final <- bool.1&&bool.2&&bool.3&&bool.4
        if (bool.final){
          break
        }
      }
      return(squares.1)
    }
  }
}
```

```
permutations <- function(n){
  if (n==1){
    return(matrix(1))
  }
  else{
    sp <- permutations(n-1)
    p <- nrow(sp)
    A <-matrix(nrow=n*p, ncol=n)
    for (i in 1:n){
      A[(i-1)*p+1:p,] <- cbind(i, sp+(sp>=i))
    }
    return(A)
  }
}
```

C 1.3: Reporting the output

```
par(mfrow=c(1, 3), mar=c(2, 2, 1.5, 1))
sq.1 <- sujiko(c(16, 19, 17, 16), c(0, 1, 0, 0, 0, 0, 6, 4, 0))
sq.2 <- sujiko(c(16, 11, 21, 21), c(8, 0, 2, 0, 0, 0, 0, 0, 0))
sq.3 <- sujiko(c(18, 13, 18, 18), c(0, 0, 0, 0, 0, 0, 0, 0, 0))
drawgrid(c(16, 19, 17, 16), sq.1, main='A')
drawgrid(c(16, 11, 21, 21), sq.2, main='B')
drawgrid(c(18, 13, 18, 18), sq.3, main='C')
```

D Question 2 R codes

```
sum.step <- function(v){
  if (length(v)<2){
    return(c())
  }
  return(c(v[1]+v[2], sum.step(v[-1])))
}

check <- function(s){
  s4 <- s
  while (length(s)>1){
    s2 <- sum.step(s)
    if (max(s2)>99){
      return(FALSE)
    }
    s.n <- length(s)
    s <- unique(c(s, s2))
    if (length(s)<(s.n+length(s2))){
      return(FALSE)
    }
```

```
    s <- s2
    s4 <- c(s4, s2)
  }
  if (!identical(s4, unique(s4))){
    return(FALSE)
  }
  return(TRUE)
}

pyramid <- function(n){
  A <- matrix(0, n, n)
  for (i.C in 1:5){
    for (i.D in (i.C+1):5){
      for (i.B in seq(1:11)[-c(i.C, i.D)]){
        if (!check(c(i.B, i.C, i.D))){
          next
        }
        for (i.E in seq(1:11)[-c(i.C, i.D, i.B)]){
          if (!check(c(i.B, i.C, i.D, i.E))){
            next
          }
          for (i.A in seq(1:30)[-c(i.C, i.D, i.B, i.E)]){
            if (!check(c(i.A, i.B, i.C, i.D, i.E))){
              next
            }
            for (i.F in seq(1:30)[-c(i.C, i.D, i.B, i.E, i.A)]){
              l <- c(i.A, i.B, i.C, i.D, i.E, i.F)
              if (!check(l)){
                next
              }
              i <- 1
              while (length(l)>0){
                print(l)
                A[i, 1:length(l)] <- l
                i <- i+1
                l <- sum.step(l)
              }
            }
          }
        }
      }
    }
  }
  return(A)
}

solve.pyramid <- function(n){
  A <- pyramid(n)
  par(mfrow=c(1, 1), mar=c(2, 2, 1.5, 1))
  plot(c(0, 6),c(0, 3.5),type="n",bty='n',xaxt='n',yaxt='n',ann=FALSE,asp=1,cex=10)
  symbols(0.5, 0.2, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
```

```
  symbols(1.5, 0.2, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(2.5, 0.2, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(3.5, 0.2, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(4.5, 0.2, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(5.5, 0.2, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(1.0, 0.8, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(2.0, 0.8, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(3.0, 0.8, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(4.0, 0.8, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(5.0, 0.8, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(1.5, 1.4, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(2.5, 1.4, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(3.5, 1.4, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(4.5, 1.4, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(2.0, 2.0, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(3.0, 2.0, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(4.0, 2.0, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(2.5, 2.6, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(3.5, 2.6, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  symbols(3.0, 3.2, rectangles = matrix(c(0.9, 0.5), 1, 2), add = TRUE, inches = FALSE,
          bg = 'white')
  for (i in 1:n){
    for (j in 1:(7-i)){
      text(0.5*i+j-1, 0.2+0.6*(i-1), A[i, j])
    }
  }
  return(plot)
}
```

E Question 3 R codes

```r
logistic.map <- function(alpha, x){
  x.t1 <- rep(NA, 1000)
  x.t1[1] <- x
  for (i in 1:999){
    x.t1[i+1] <- alpha * x.t1[i] * (1-x.t1[i])
  }
  return(x.t1[800:1000])
}

alpha <- seq(2.8, 3.6, length.out = 300)
logistic <- sapply(alpha, logistic.map, x = runif(1))
alpha.plot <- sort(rep(alpha, 201))
plot(alpha.plot, logistic, pch = '.', bty = 'n')


doub <- function(alpha, x){
  ind <- 1
  doubl <- c()
  for (i in 1:length(alpha)){
    x.t1 <- rep(NA, 1000)
    x.t1[1] <- x
    for (j in 1:999){
      x.t1[j+1] <- alpha[i] * x.t1[j] * (1-x.t1[j])
    }
    x.unique <- unique(round(x.t1[800:1000], 1))
    if (length(x.unique) > ind){
      ind <- length(x.unique)
      doubl <- c(doubl, alpha[i])
    }
  }
  return(doubl)
}

alpha <- seq(2.8, 3.6, by = 0.001)
doubling.list <- doub(alpha, 0.1)

alpha <- seq(2.8, 3.6, length.out = 300)
logistic <- sapply(alpha, logistic.map, x = runif(1))
alpha.plot <- sort(rep(alpha, 201))
plot(alpha.plot, logistic, pch = '.', bty = 'n')

for (i in 1:length(doubling.list)){
  abline(v = doubling.list[i], lty = 2)
}
```