Assignment 1

Po-Hsiang Peng

Section 4.1.1

Exercise 2

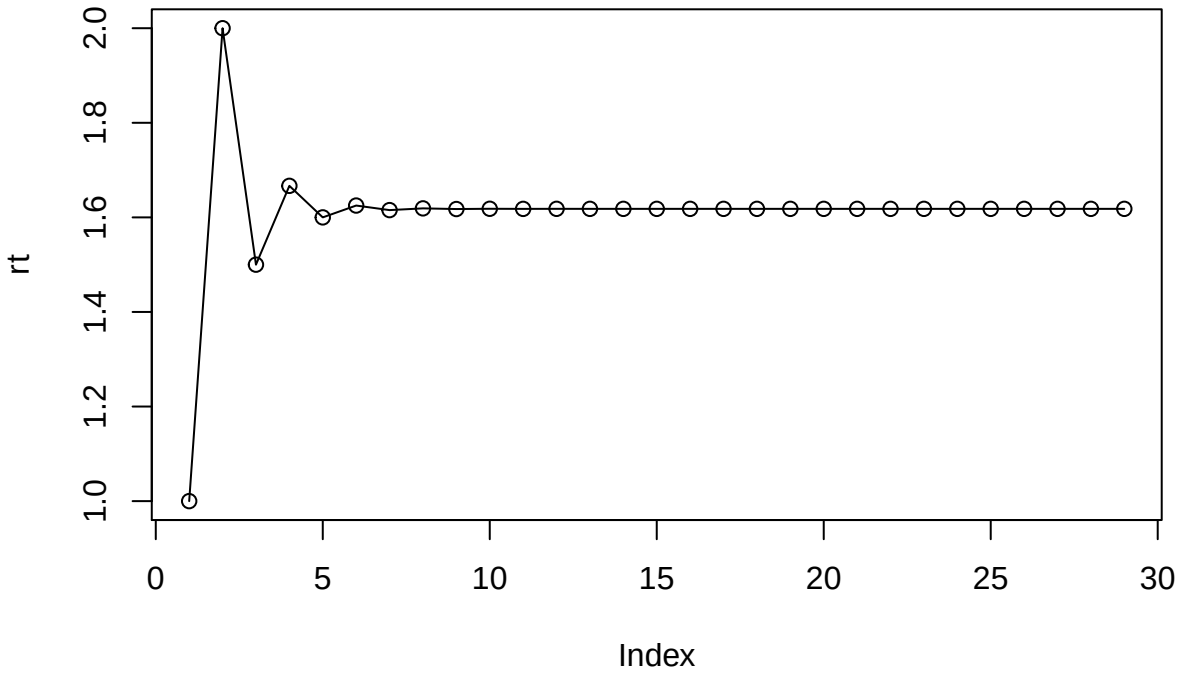
Problem (a)

```
# compute the ratio of f_{n}/f_{n-1} for n = 2,...,30
n = 30
fib = numeric(n)
fib[c(1,2)] = 1
for(i in 3:n) {
  fib[i] = fib[i-1] + fib[i-2]
}
rt = fib[2:n]/fib[1:(n-1)]
rt
```

```
##  [1] 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385
##  [8] 1.619048 1.617647 1.618182 1.617978 1.618056 1.618026 1.618037
## [15] 1.618033 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
## [22] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
## [29] 1.618034
```

According to the following plot, we can see this ratio is convergent. (See problem (b) for proof.)

```
plot(rt, type="o")
```



Problem (b)

The Golden ratio is `1.618034`.

```
(1+sqrt(5))/2
```

```
## [1] 1.618034
```

We then prove f_n/f_{n-1} equals to golden ratio. Let $\lim f_n/f_{n-1} = L$, then

$$\begin{aligned} L &= \lim_{n\to\infty} \frac{f_{n+1}}{f_n} \\ &= \lim_{n\to\infty} \frac{f_n + f_{n-1}}{f_n} \\ &= 1 + \frac{1}{L} \end{aligned}$$

which yields $L = (1 \pm \sqrt{5})/2$. Moreover, $(1 - \sqrt{5})/2$ is impossible because Fibonacci numbers are positive.

Exercise 3

Problem (a)

The problem (a) is equivalent to $\sum_{j=1}^{5} j$. Thus, we have `answer=15`.

```
answer = 0
for(j in 1:5) answer = answer + j
answer
```

```
## [1] 15
```

Problem (b)

In problem (b), we collect from 1 to 5. Thus, we have `answer=c(1,2,3,4,5)`.

```
answer = NULL
for(j in 1:5) answer = c(answer, j)
answer
```

```
## [1] 1 2 3 4 5
```

Problem (c)

In problem (b), we collect from 0 to 5. Thus, we have `answer=c(0,1,2,3,4,5)`.

```
answer = 0
for(j in 1:5) answer = c(answer, j)
answer
```

```
## [1] 0 1 2 3 4 5
```

Problem (d)

This is equivalent to $\prod_{j=1}^{5} j$. Thus, we have `answer=120`.

```
answer = 1
for(j in 1:5) answer = answer * j
answer
```

```
## [1] 120
```

Problem (e)

Problem (e) is equivalent to the following

$$
\begin{aligned}
x_1 &= 3 \\
x_2 &= (7 \cdot x_1) \bmod 31 \\
x_3 &= (7 \cdot x_2) \bmod 31 \\
&\vdots
\end{aligned}
$$

Thus, we have `answer=c(3,21,23,6,...)`.

```
answer = 3
for(j in 1:15) answer = c(answer, (7 * answer[j]) %% 31)
answer
```

```
##  [1]  3 21 23  6 11 15 12 22 30 24 13 29 17 26 27  3
```

Note that the last number of `answer` is 3. Thus, even though we do not know the rule used to determine this sequence, we still know the next number is 21.

```
answer = 3
for(j in 1:16) answer = c(answer, (7 * answer[j]) %% 31)
answer
```

```
##  [1]  3 21 23  6 11 15 12 22 30 24 13 29 17 26 27  3 21
```

Section 4.1.2

Exercise 4

```
# @P: initial investment amount
# @n: length of the term
# output: amount of interest earned over the term
GIC = function(P, n) {

  if(n <= 3)
    i = 0.04
  else
    i = 0.05

  I = P * ((1+i)^n - 1)
  return(I)
}
```

Exercise 5

```
# @P: original principal
# @n: length of the term (in months)
# @open: whether the mortgage term is open or closed
#        if the mortgage term is open, then the input should be TRUE
# output: monthly mortgage payment
MIR = function(P, n, open) {
```

```
  if(open==TRUE)
    i = 0.05
  else
    i = 0.04

  R = P * i / (1 - (1+i)^(-n))
  return(R)
}
```

Section 4.1.3

Exercise 2

```
# @n: the upper bound of Fibonacci numbers f_k
# output: Fibonacci numbers f_k, where f_k < n for all k
Fib = function(n) {
  Fibonacci = c(1,1)

  while(sum(tail(Fibonacci,2)) < n) {
    Fibonacci = c(Fibonacci, sum(tail(Fibonacci,2)))
  }
  return(Fibonacci)
}

Fib(300)
```

```
##  [1]   1   1   2   3   5   8  13  21  34  55  89 144 233
```

Exercise 4

```
# @x0: initial point
# epsilon: tolerance error
# output: i such that i = (1-(1+i)^{-20})/19
fp = function(x0, epsilon=1e-6) {

  i = x0
  i_s = (1-(1+i)^(-20))/19

  while(abs(i-i_s) > epsilon){
    i = i_s
    i_s = (1-(1+i)^(-20))/19
  }
  return(i_s)
}

fp(0.06)
```

```
## [1] 0.00495397
```

When the starting value is 0, the result becomes 0. Furthermore, when the absolute value of the initial point

is too small, say `1e-5`, then the result will be different. Other initial points (even negative number) give the same result.

```
fp(0)
```

```
## [1] 0
```

```
fp(1e-5)
```

```
## [1] 1.052521e-05
```

```
fp(-1e-5)
```

```
## [1] -1.052742e-05
```

```
fp(100)
```

```
## [1] 0.004953779
```

```
fp(-100)
```

```
## [1] 0.004953779
```

Exercise 5

```
# @x0: initial point
# epsilon: tolerance error
# output: c(i,count) where
#   i is the number such that i = (1-(1+i)^{-20})/19
#   count is the iteration number
fp_m = function(x0, epsilon=1e-6) {

  i = x0
  i_s = (1-(1+i)^(-20))/19

  count = 1

  while(abs(i-i_s) > epsilon){
    i = i_s
    i_s = (1-(1+i)^(-20))/19
    count = count + 1
  }
  return(c(i_s,count))
}

fp_m(0.06)
```

```
## [1] 4.95397e-03 1.05000e+02
```

Section 4.1.5

Exercise 2

Problem (a)

This algorithm finds all prime number up to n. It first adds 2 in the prime list and deletes all numbers that are divisible by 2. Then, it adds the next number (3 in this case) in the prime list and delete all numbers that are divisible by 3. It repeats until n.

Problem (b)

If n is not a prime, then n can be written as

$$n = a \cdot b$$

where $1 < a \leq b < n$. It follows that

$$n = a \cdot b \geq a^2 \implies a \leq \sqrt{n}$$

Thus, we only need to test the number that are less than $\sqrt{n}$.

Problem (c)

```
# @n: find all prime number less than n
# output: all prime number less than n
Eratosthenes = function(n) {
  if(n>=2) {
    sieve = seq(2, n)
    primes = c()
    while (length(sieve) > 0) {
      p = sieve[1]
      primes = c(primes, p)
      sieve <- sieve[(sieve %% p) != 0]
      if(p>=sqrt(n)) break
    }
    primes = c(primes, sieve)
    return(primes)
  } else {
    stop("Input value of n should be at least 2.")
  }
}
```

Section 4.2.1

Exercise 2

Problem (a)

```
# @P: principle
# @n: length of the term
# @i: interest rate
# output: accumulated value
compound.interest = function(P, n, i) {
  AV = P * (1+i)^n
  return(AV)
}
```

Problem (b)

In this case, we have $P = 1000$, $n = 30$ and $i.r = 0.01$.

```
compound.interest(P=1000, n=30, i=0.01)
```

```
## [1] 1347.849
```

Exercise 3

```
# @f: the user defined function, for example f=function(x) {x+1}
# @a,b: a,b such that f(a)*f(b)<0
# @epsilon: tolerance error
# output: x such that f(x)=0
bisec = function(f, a, b, epsilon=1e-6) {

  if(f(a)*f(b) > 0) {
    print("Enter a, b such that f(a)*f(b) < 0")
    return(NULL)
  }

  m = (a+b)/2

  while(abs(f(m)) > epsilon) {

    if(f(m)*f(a) > 0) {
      a = m
      m = (a+b)/2
    } else {
      b = m
      m = (a+b)/2
    }
  }
  return(m)
}
```

Section 4.4.1

Exercise 1

We can simply add a few lines to allow decreasing order.

```
mergesort = function (x, decreasing=FALSE) {

  ...

  if(decreasing==TRUE)
    return(rev(result))
  else
    return(result)
}
```

Note that this code is not correct when `length(x)` is odd. We have to use the following line when split `x` in half:

```
m = floor(n/2)
y = x[1:m]
z = x[(m+1):n]
```

The full code is as follows:

```
# x: a vector
# @decreasing: FALSE for ascending; TRUE for descending sort
# output: sorted vector
mergesort = function (x, decreasing=FALSE) {
  n = length(x)

  if(n < 2){
    result = x
  }
  else{

    m = floor(n/2)
    y = x[1:m]
    z = x[(m+1):n]

    y = mergesort(y)
    z = mergesort(z)

    result = c()

    while(min(length(y),length(z)) > 0) {
      if(y[1] < z[1]) {
        result = c(result, y[1])
        y = y[-1]
      } else {
        result = c(result, z[1])
        z = z[-1]
      }
    }

    if(length(y) > 0)
      result = c(result, y)
    else
      result = c(result, z)
  }

  if(decreasing==TRUE)
    return(rev(result))
  else
    return(result)
}
```

Exercise 2

Problem (a)

```
# @x0: initial value for x
# @y0: initial value for y
# @f: should be the form expression(f(x,y))
#     for exmaple, expression(x+2*y+5)
# @g: should be the form expression(g(x,y))
# @eps: tolerance
# output: c(x,y) such that f(x,y)=g(x,y)=0
Newton = function(x0, y0, f, g, eps) {

  df = deriv(f, c("x","y"))
  dg = deriv(g, c("x","y"))

  x = x0
  y = y0

  while(abs(eval(f)) > eps || abs(eval(g)) > eps) {

    evdf = eval(df)
    evdg = eval(dg)

    f = evdf[1]
    fx = attr(evdf,"gradient")[1]
    fy = attr(evdf,"gradient")[2]

    g = evdg[1]
    gx = attr(evdg,"gradient")[1]
    gy = attr(evdg,"gradient")[2]

    d = fx * gy - fy * gx

    x = x - (gy*f - fy*g)/d
    y = y - (fx*g - gx*f)/d
  }

  return(c(x,y))
}
```

Problem (b)

```
f = expression(x + y)
g = expression(x^2 + 2*y^2 - 2)
Newton(1,-1,f,g,1e-5)
```

```
## [1]  0.8164966 -0.8164966
```

```
Newton(-1,1,f,g,1e-5)
```

```
## [1] -0.8164966  0.8164966
```

Analytically, the solution of this system is $(\sqrt{2/3}, -\sqrt{2/3})$ or $(-\sqrt{2/3}, \sqrt{2/3})$, which is in agreement with my numerical solution.

Chapter 4

Exercise 1

```
# This function compute cn * x^(n-1) + ... + c2 * x + c1
# @x: can be the form x, or c(x1,x2,...,xn)
# @c: should be the form c(cn,...,c2,c1)
directpoly = function(x,c) {
  n = length(c)

  if(length(x)==1)
    return(sum(x^((n-1):0) * c))
  else
    return(rowSums(sapply((n-1):0, function(i) x^i * c[n-i])))
}
```

Exercise 2

```
# This function compute cn * x^(n-1) + ... + c2 * x + c1
# @x: can be the form x, or c(x1,x2,...,xn)
# @c: should be the form c(cn,...,c2,c1)
hornerpoly = function(x,c) {
  n = length(c)

  a = matrix(0, nrow=length(x), ncol=n)
  a[,1] = c[1]

  for(i in 2:n) {
    a[,i] = a[,i-1] * x + c[i]
  }

  return(a[,n])
}
```

Exercise 3

Problem (a)

```
system.time(directpoly(x=seq(-10,10,length=5000000), c(1,-2,2,3,4,6,7)))
```

```
##    user  system elapsed
##   1.299   0.292   1.597
```

```
system.time(hornerpoly(x=seq(-10,10,length=5000000), c(1,-2,2,3,4,6,7)))
```

```
##    user  system elapsed
##   0.368   0.198   0.567
```

Problem (b)

We can see that the time difference is smaller, but `hornerpoly` is still more efficient.

```
system.time(directpoly(x=seq(-10,10,length=5000000), c(2,17,-3)))
```

```
##    user  system elapsed
##   0.264   0.102   0.368
```

```
system.time(hornerpoly(x=seq(-10,10,length=5000000), c(2,17,-3)))
```

```
##    user  system elapsed
##   0.169   0.038   0.208
```