2.1

```
mysign <- function(x)
{
   # tworzymy nowy wektor numeryczny o długości length(x)
   # czyli takiej samej, jak wektor wejściowy
   # na wszystkich miejscach automatycznie pojawią się zera
   wynik <- numeric(length(x))
   # teraz na niektórych miejscach trzeba ustawić jedynki (gdy w wektorze wejściowym są
   # tam liczby dodatnie)
   wynik[x>0] <- 1
   # a w innych minus jedynki
   wynik[x<0] <- -1
   wynik
}
mysign(-4:4)
mysign(c(-5,6,0,0,3,-5,-6,0,2))
```

2.2

```
myabs <- function(x)
{
   # bardzo podobnie jak w poprzednim zadaniu, też potrzebujemy wektora o długości wektora
   # wejściowego, zainicjowanego zerami
   wynik <- numeric(length(x))
   # tym razem zamiast jedynek wstawiamy odpowiednie elementy z wektora wejściowego x
   wynik[x>0] <- x[x>0]
   # pamiętamy, że w przypadku ujemnych należy zmienić ich znak na dodatni
   wynik[x<0] <- -x[x<0]
   wynik
}
myabs(-4:4)
```

2.3

```
myall<-function(p)
{
# posługujemy się sztuczką: zamieniamy (niejawnie) wektor logiczny p na liczbowy (TRUE
# przechodzi na 1, a FALSE na 0), a następnie szukamy minimum. Następnie minimum konwertujemy
# w sposób jawny na wartość logiczną (1 na TRUE, 0 na FALSE)
# jeśli jest choć jedno FALSE w p, to minimum z całego wektora to 0, a więc zostanie zwrócone
# FALSE
   as.logical(min(p))
}
myall(c(TRUE, FALSE, TRUE))
myall(c(TRUE, TRUE, TRUE))
```

2.4

```
myany<-function(p)
{
# Bardzo podobna sztuczka jak w poprzednim zadaniu. Tym razem jednak szukamy maksimum. Jeśli
# choć jedna wartość w p to TRUE, to maksimum z całego wektora to 1, a więc TRUE
```

```
   as.logical(max(p))
}
myany(c(TRUE, FALSE, TRUE))
myany(c(FALSE, FALSE, FALSE))
```

2.5

```
mycumsum <- function(x)
{
# sprawdzamy, czy x jest aby na pewno wektorem numerycznym. Każda funkcja powinna sprawdzać
# wszystkie swoje argumenty
   stopifnot(is.numeric(x))
# w wyniku chcemy otrzymać wektor o tej samej długości, co wektor wejściowy
   n <- length(x)
   ret <- numeric(n)
# zadanie polega na zsumowaniu wszystkich elementów wektora, przy czym poszczególne sumy
# częściowe zapisujemy w wektorze wynikowym. Aby zsumować elementy wektora, trzeba mieć
# zmienną, w której będziemy przechowywać sumę dotychczas odwiedzonych elementów wektora x
   sum <- 0
# seq_along pozwoli nam nie wpaść w pułapkę dla wektora o długości 0
   for(i in seq_along(x))
   {
# i to indeks elementu, który właśnie odwiedzamy. A więc do elementu wektora x odwołujemy się
# przez x[i]. Zdecydowaliśmy się na indeks, aby wiedzieć, pod jaką pozycją zapisywać sumę
# częściową w wektorze wynikowym ret
      sum <- sum + x[i]
      ret[i] <- sum
   }
   ret
}
x <- c(5,3,4,7,8)
library(testthat)
expect_equal(cumsum(x), mycumsum(x))
```

2.6

```
dziesietnaNaBinarna <- function(liczby)
{
# sprawdzamy kolejno: czy liczby to wektor numeryczny, czy wszystkie elementy są całkowite i
# czy są większe od zera
   stopifnot(is.numeric(liczby), all(liczby%%1 == 0), all(liczby >= 0))
# unlist(lapply(...)) to bardzo typowa zagrywka i warto się jej nauczyć. Oznacza ona
# przeprowadzenie jakiegoś ciągu operacji na każdym elemencie wektora z osobna, niezależnie od
# wyników operacji na pozostałych elementach wektora wejściowego. unlist() służy nam, aby na
# końcu dostać wektor zamiast listy.
   unlist(lapply(liczby, function(liczba){
#osobno obsługujemy przypadek szczególny, bo logarytm nie zadziała dla zera
   if(liczba == 0) {
      return("0")
   }
```

```
# ile będzie nam potrzeba cyfr w systemie binarnym na daną liczbę? Warto wiedzieć, że wyznacza
# się to właśnie logarytmem o odpowiedniej podstawie. Zauważmy, że np. w systemie dziesiętnym
# liczba log10(1234) da nam 3.091315 .
   k<-floor( log2(liczba)+1)
# tworzymy wektor napisowy o długości k
   wynik <- character(k)
# zmienna przechowująca indeks, pod którym zapisujemy kolejne cyfry
   index <- 1
# pętla while, ponieważ nie wiemy z góry, po ilu iteracjach będziemy mieli cały zapis (to nie
# do końca prawda, można szacować z góry, np. Poprzez liczbę k)
   while(liczba > 0)
   {
# jedziemy zgodnie z algorytmem: reszta z dzielenia przez 2 idzie jako cyfra w zapisie
# binarnym
      wynik[index] <- liczba%%2
# index powiększamy, aby następna cyfra była na kolejnej pozycji w wektorze
      index <- index + 1
# liczba zmniejsza się dwukrotnie dzieleniem całkowitoliczbowym
      liczba = liczba %/% 2
   }
# tak naprawdę w wyniku dostaliśmy zapis odwrócony. Dlatego używamy rev(), aby dostać cyfry
# w dobrej kolejności, tak jak przyzwyczailiśmy się zapisywać i czytać liczby. Jako
# eksperyment myślowy warto zauważyć, że nic nie stoi na przeszkodzie, aby liczbę tysiąc
# dwieście trzydzieści cztery zapisywać jako 4321 (zapis dziesiętny). Przy okazji paste0
# łączy nam wszystkie elementy wektora w jeden napis, wstawiając pomiędzy poszczególne
# elementy wektora znak podany jako argument collapse. Przetestuj dla collapse=”##”.
   paste0(rev(wynik), collapse="")
   }))
}
dziesietnaNaBinarna(c(20,20,10,1,2,0,3,32,64,128))
```

2.7

```
# to zadanie jest bardzo podobne do poprzedniego, więc zostanie skomentowane tylko to, co je #
różni

# w tym zadaniu nie mamy pewności, że otrzymamy pełen zapis liczby w skończonej liczbie
kroków. Istnieją ułamki, które mają nieskończony zapis w notacji binarnej, tak jak i istnieją
ułamki, które mają taki zapis w notacji dziesiętnej (jak choćby ⅓ jako 0,(3) w systemie
dziesiętnym). Stąd argument rozwiniecieCyfry, który mówi, ile cyfr nas maksymalnie interesuje
dziesietnaNaBinarnaUlamek <- function(liczby, rozwiniecieCyfry)
{
   stopifnot(is.numeric(liczby), all(liczby >= 0))

   unlist(lapply(liczby, function(liczba){
      if(liczba == 0)
      {
         return("0")
      }
      wynik <- character(rozwiniecieCyfry)
      index <- 1
```

```
# możemy zakończyć pętle w jednym z dwóch przypadków: gdy po prostu otrzymaliśmy pełen zapis
# liczby (przypadek ułamka skończonego, liczba == 0) lub skończyły nam się cyfry, które
# planowaliśmy poświęcić na zapis ułamka (prawdopodobnie ułamek nieskończony, index >
# rozwiniecieCyfry)
      while(liczba != 0 && index <= rozwiniecieCyfry)
      {
# ciekawa funkcja ifelse: jeśli pierwszy argument jest prawdą, to zwraca drugi argument, a w
# przeciwnym wypadku zwraca trzeci argument. Przydatna, gdy chcemy na bazie warunku logicznego
# przypisać do zmiennej konkretną wartość. Zawsze da się ją zastąpić przez zwykłą instrukcję
# warunkową if (ale wtedy mamy dłuższy zapis)
         wynik[index] <- ifelse(liczba >= 1, 1, 0)
         if(liczba >= 1) liczba = liczba - 1
         index <- index + 1
         liczba = liczba * 2
      }
      # troszeczkę modyfikujemy paste0, aby po pierwszej cyfrze wstawić przecinek
      paste0(wynik[1],",", paste0(wynik[2:length(wynik)], collapse=""))
   }))
}
dziesietnaNaBinarnaUlamek(0.25,4)
dziesietnaNaBinarnaUlamek(0.5,4)
dziesietnaNaBinarnaUlamek(0.1,200)
dziesietnaNaBinarnaUlamek(c(0.1, 1/6),200)
```

2.8

```
# w tym zadaniu przechodzimy po wektorze i sprawdzamy, czy dany element jest taki sam, jak
# poprzedni. Jeśli tak, to podbijamy licznik takich samych liczb pod rząd. Jeśli nie, to
zapisujemy
# liczbę i jej liczność w wektorach wynikowych.
myrle <- function(x)
{
   stopifnot(is.numeric(x))
   n <- length(x)
# wektory wynikowe. Zwróćmy uwagę, że nie wiemy z góry, jak długie one będą, ale na pewno nie
# będą dłuższe niż n. Innymi słowy, wektory będą najdłuższe, gdy każda liczba będzie
# występować jako jedyna pod rząd.
   lengths <- numeric(n)
   values <- numeric(n)
# w tej zmiennej trzymamy informację, pod jakim indeksem będziemy zapisywać następny wynik
   valuesLengthsIndex <- 1
# aktualna liczba, której liczność pod rząd badamy
   value <- NULL
# w tej zmiennej trzymamy informację, ile wystąpień pod rząd było do tej pory
   lengthNow <- 0
   for(i in seq_along(x))
   {
# jak dopiero zaczynamy, to przypisujemy sobie wartość aktualnej liczby do value
      if(is.null(value))
         value <- x[i]
```

```
# jeśli napotkaliśmy kolejną liczbę pod rząd, podbijamy licznik liczności
      if(x[i]==value)
      {
         lengthNow <- lengthNow + 1
      } else
      {
# w przeciwnym wypadku zapisujemy sobie poprzednią serię liczb i zaczynamy nową
         values[valuesLengthsIndex] <- value
         lengths[valuesLengthsIndex] <- lengthNow
         valuesLengthsIndex <- valuesLengthsIndex + 1
         value <- x[i]
         lengthNow <- 1
      }

   }
   # może się okazać, że nie zapisaliśmy w pętli ostatniej serii liczb. Dlatego robimy to
   # teraz.
   if(!is.null(value))
   {
      values[valuesLengthsIndex] <- value
      lengths[valuesLengthsIndex] <- lengthNow
   }
   # zwracamy listę
   list(values = values[seq_len(valuesLengthsIndex)],
        lengths = lengths[seq_len(valuesLengthsIndex)])
}

x <- c(5,3,4,7,8,5,6,5,3,4,4,4,4,3,3,3,5,5,4,4)
library(testthat)
expect_equal(rle(x)$values, myrle(x)[[1]])
expect_equal(rle(x)$lengths, myrle(x)[[2]])
```

2.13

```
dodawanie <- function(a,b, base=10)
{
   stopifnot(is.numeric(a))
   stopifnot(is.numeric(b))
   stopifnot(is.finite(a))
   stopifnot(is.finite(b))
   stopifnot(a >= 0, a <= base-1)
   stopifnot(b >= 0, b <= base-1)
   stopifnot(a %% 1 == 0, b %% 1 == 0)
   stopifnot(length(a) == length(b))
# dopisujemy po zerze z przodu, aby mieć gdzie umieścić nadmiar, np. W przypadku dodawania 999
# do 999
   a <- c(0,a)
   b <- c(0,b)

   while(TRUE)
   {
```

```
# dodajemy cyfry jak leci, nie przejmując się nadmiarami
      a = a+b
# robimy dzielenie całkowitoliczbowe, które wyłuskuje nam nadmiary
      reszta <-  floor(a/base)
# teraz w a możemy zostawić tylko cyfry jedności, używamy modulo
      a <- a%%base
# gdy reszta (nadmiar) jest wektorem zerowym, dodawanie jest skończone
      if(all(reszta==0)) break
# jeśli nie, będziemy dodawać w następnej iteracji nadmiary przesunięte o jeden w lewo
      b = c(reszta[-1],0)
   }
# tutaj pozbywamy się zer nieznaczących z przodu, jeśli takie są
   indeks <- which.max(a!=0)

   if(indeks == 1 && a[1]==0)
      0
   else if(indeks == 1)
      a
   else
      a[indeks:length(a)]
}
```

2.14

```
sredniaRuchoma <-function(x,k)
{
# zadanie najlepiej najpierw dobrze sobie rozrysować i zorientować się, ile liczb z wektora
# wejściowego składa się na pojedynczy element wektora wyjściowego
   stopifnot(is.numeric(x))
   stopifnot(is.numeric(k), length(k)==1, k%%2 == 1)
   n <- length(x)
# w to będzie nasz wektor wyjściowy
   w <- numeric(n-k+1)
# potrzebna jest nam pewna suma. Suma pierwszych k elementów z wektora x. Jest to wartość
# elementu pierwszego wektora w. Ale także jest to prawie wszystko, co jest nam potrzebne, aby
# obliczyć wartość drugiego elementu wektora w. Brakuje nam tylko w tej sumie jednego elementu
# z prawej, za to mamy nadmiar w postaci elementu z tyłu.
   sumW <- sum(x[1:k])
   w[1] <- sumW
   indexW <- 2
   for(i in (k+1):n)
   {
# dlatego w pętli będziemy dodawać jeden element z przodu (x[i]) i odejmować jeden element z
# tyłu (x[i-k])
         sumW <- sumW + x[i] - x[i-k]
         w[indexW] <- sumW
         indexW <- indexW + 1
   }
# oczywiście wynikiem ma być średnia, a nie sumy. Dlatego dzielimy przez k
   w/k
}
```

```r
sredniaRuchoma2 <-function(x,k)
{
# to bardzo sprytna, zwektoryzowana wersja rozwiązania zadania. Polega na obserwacji, że jeśli
# mamy skumulowane sumy w wektora x, to aby policzyć sumę kolejnych elementów wektora x, od
# indeksu i do wektora j (i<j) włącznie, to wystarczy policzyć w[j]-w[i-1] (zakładamy, że
# w[0]==0)
   w <- cumsum(x)
   n <- length(x)
   w[(k):length(x)] <- w[(k):length(x)] - c(0,w[1:(n-k)])
   w[(k):length(x)]/k
}

x
sredniaRuchoma(x,3)
sredniaRuchoma2(x,3)
expect_equal(sredniaRuchoma(x,3), sredniaRuchoma2(x,3))
```

2.15

```r
# wersja 1, trywialna
potegowaniePetla <- function(x,n)
{
   wynik <- 1
   for(i in 1:n)
      wynik <- wynik * x
   wynik
}

# wersja 2, rekurencyjna, naiwna
potegowanieRekLiniowe <- function(x,n)
{
 # jest to typowa funkcja rekurencyjna: najpierw sprawdzamy przypadki brzegowe i zwracamy
 # oczywiste wartości
   if(n == 0) return(1)
   if(n == 1) return(x)
   if(n == 2) return(x*x)
 # x^n to tak innymi słowy x*x^{n-1}
   x*potegowanieRekLiniowe(x,n-1)
}

# wersja 3, rekurencyjna, wydajna
# Ta wersja bazuje na następującej obserwacji: gdy chcemy policzyć x^16, to tak naprawdę
# wystarczy nam wiedzieć, ile jest a = x^8. Wtedy wystarczy policzyć a*a, pomijając zbędne
# mnożenia.
# A co, gdy mamy nieparzysty wykładnik? Np. x^17? Wtedy wystarczy policzyć x^16 jak
# poprzednio, ale wynik domnożyć przez x, czyli x*a*a, gdzie a=x^8.
potegowanieRek <- function(x,n)
{
   if(n == 0) return(1)
   if(n == 1) return(x)
```

```
   if(n == 2) return(x*x)
   dodatek <- 1
   if(n %% 2 == 1)
      dodatek <- x
   ret <- potegowanieRek(x,n%/%2)
   ret*ret*dodatek
}

library(microbenchmark)
expect_equal(potegowanieRek(2,0), 2**0)
expect_equal(potegowanieRek(2,1), 2**1)
expect_equal(potegowanieRek(2,2), 2**2)
expect_equal(potegowanieRek(2,3), 2**3)
expect_equal(potegowanieRek(2,4), 2**4)
expect_equal(potegowanieRek(2,5), 2**5)
expect_equal(potegowanieRek(2,6), 2**6)
expect_equal(potegowanieRekLiniowe(2,6), 2**6)
expect_equal(potegowanieRekLiniowe(2,3), 2**3)
expect_equal(potegowanieRekLiniowe(2,17), 2**17)
expect_equal(potegowaniePetla(2,17), 2**17)
microbenchmark(potegowanieRek(2,17), potegowanieRekLiniowe(2,17), potegowaniePetla(2,17))
```

2.16

```
szachownica <- function(wiersze,kolumny)
{
# wytwarzamy wiersze, w zależności od tego, czy będziemy w parzystym wierszu, czy
# nieparzystym. Główną pracę wykonuje tu argument length.out
   wierszP <- rep(c("*","#"), length.out = kolumny)
   wierszN <- rep(c("#","*"), length.out = kolumny)
# w pętli sprawdzamy, w którym jesteśmy wierszu i wypisujemy odpowiedni, przygotowany
# wcześniej, wiersz
   for(i in seq_len(wiersze))
      if(i %% 2 == 0)
         print(paste0(wierszP, collapse= ""))
      else
         print(paste0(wierszN, collapse= ""))
}
szachownica(5,7)
```

2.17

```
powiekszonaSzachownica <- function(wiersze,kolumny, k)
{
# w tym wypadku musimy sobie pomóc parametrem each
   wierszP <- rep(c("*","#"), each=k, length.out = kolumny)
   wierszN <- rep(c("#","*"), each=k, length.out = kolumny)
# najciekawsze w tym zadaniu jest wykrywanie, w którym wierszu jesteśmy. Aby zbić je w grupy
# po k małych wierszy, należy dokonać dzielenia całkowitoliczbowego przez k
   for(i in seq_len(wiersze))
      if((i%/%k) %% 2 == 0)
         print(paste0(wierszP, collapse= ""))
```

```
   else
      print(paste0(wierszN, collapse= ""))
}

powiekszonaSzachownica(5,11,3)
```

2.11 ???

```
uzubraki <- function(x)
{
   n <- length(x)
   indices <- which(!is.na(x))
   indicesLength <- length(indices)
   x[1:indices[1]] <- x[indices[1]]
   x[indices[indicesLength]:n] <- x[indices[indicesLength]]

   for(i in seq_along(indices))
   {
      if(i < indicesLength && indices[i]+1 != indices[i+1])
      {
         left <- x[indices[i]]
         right <- x[indices[i+1]]
         len <- indices[i+1] - indices[i]
         wek <- left + ((right-left)/len)*1:len
         x[(indices[i]+1):(indices[i+1]-1)] <- wek[-length(wek)]
      }
   }

   x
}

x <- c(NA, NA, 1, 2, 3, NA, NA, 6, NA, NA)
uzubraki(x)
```

2.18

```
is.prime <- function(x)
{
   stopifnot(is.numeric(x))
   stopifnot(length(x)>0)
   stopifnot(unlist(lapply(x,function(el){floor(el)==el})))
   # zakladam, dla ulatwienia, ze naturalne liczby sa od 1 w gore
   stopifnot(unlist(lapply(x,function(el){el > 0})))

   unlist(lapply(x,
          function(el){
             if(el==1) {return(FALSE)}
             granica <- sqrt(el)
             for(i in 2:floor(granica)) #mam pewnosc, ze granica >= 1
             {
```

```
                if(el %% i == 0)
                {
                   return(FALSE)
                }

             }
             return(TRUE)
          }))
}

## ---- Przyklady ----
#dobre
is.prime(c(1,2,3,41))


is.prime(c(1,2.5,3))

Euler <- unlist(lapply(0:39,function(n){n*n+n+41}))
is.prime(Euler)
```