Family Name: *Given Name:* *Student Number:*

1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
T	

UNIVERSITY OF TORONTO
Faculty of Arts and Science

APRIL EXAMINATIONS 2017

CSC 121H1S

Duration - 3 hours

No Aids Allowed

Answer all questions in the space provided; if you run out of space, use the back of a page, and indicate where the answer continues.

NO AIDS ALLOWED. No books, notes, calculators, or other aids.

The 10 questions are each worth 10 marks.

You must obtain at least 40% on this final exam to pass the course.

1. In the ten blank spaces below, write what R will print if the commands shown are entered one after the other into the R console. (Note: The > characters are the R command prompts, not something that was typed in.)

```
> a <- 4
> b <- 2
> a*b+a/b
```

```
> a*(b+a)/b
```

```
> a <- b+1
> b <- a+1
> c(a+b,a-b,a*b)
```

```
> fn <- function (a,b,c) a+2*b+3*c
> fn(1,2,3)
```

```
> fn(3,a,b)
```

```
> fn(fn(1,0,0),fn(0,1,0),fn(0,0,1))
```

```
> a <- "cat"
> b <- "dog"
> x <- c(a,b,a,a,b)
> x[length(x)-1]
```

```
> x == "dog"
```

```
> x [x != "dog"]
```

```
> paste(a,b,a,sep="")
```

2. In the five blank spaces below, write what R will print if the commands shown are entered one after the other into the R console. (Note: The > and + characters at the beginnings of lines are the R command prompts, not something that was typed in.)

```
> v <- c(4,1,6,10,3,7,11,2)
> for (i in 1:8) if (v[i]>5) v[i] <- v[i+1]
> v
```

```
> for (i in 1:8) if (v[i]>5) v[i] <- v[i+1]
> v
```

```
> w <- c(1:100,3001:3100)
> for (i in 2:length(w)) w[i] <- w[i-1]
> sum(w)
```

```
> M <- matrix (1:12, nrow=3, ncol=4)
> M
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> s <- 0
> i <- 1
> while (s < 4) { s <- s + M[i,i]; i <- i + 1 }
> s
```

```
> i <- 1; j <- 1
> repeat {
+     M[i,j] <- M[i,j] + 1
+     if (M[i,j] > 10) break
+     if (i < 3) i <- i + 1
+     else { i <- 1; j <- j + 1 }
+ }
> M
```

3. Here is the definition of a function called `mystery1`:

```
mystery1 <- function (v) {
    s <- 0
    if (length(v) > 1) {
        for (i in 2:length(v)) {
            if (v[i-1] == v[i]) s <- s + 1
        }
    }
    s
}
```

For the three calls of this function below, write down the value that R will print:

```
> mystery1 (c(5,6,6,10,9,6,9,20,20,2,2,2))
```

```
> mystery1 (c("he","saw","that","that","is","what","that","is"))
```

```
> mystery1 (c(1:10,10:100,100:1000))
```

4. Write an R function called `check_bounds` that takes three arguments, called `v`, `low`, and `high`, which are all numeric vectors of the same length (which you may assume without checking). The `low` argument contains low bounds on the elements of `v`, and the `high` argument contains high bounds on the elements of `v`. The value returned by the function should be the same as `v` if all its elements are between the corresponding elements of `low` and `high`. If any elements of `v` are outside these bounds, the value returned should be like `v` except that the elements outside these bounds should be changed to `NA`. Here is an example call:

```
> check_bounds (v=c(2,9,7,1,8), low=c(1,9,8,0,4), high=c(2,10,12,3,7))
[1]  2  9 NA  1 NA
```

5. Here is the definition of a function called mystery2:

```
mystery2 <- function (A, B) {
    for (i in 1:nrow(A)) {
        for (j in 1:ncol(A)) {
            if (is.na(A[i,j])) {
                v <- B[i,j]
                if (i > 1) v <- c(v,B[i-1,j])
                if (j > 1) v <- c(v,B[i,j-1])
                A[i,j] <- sum(v) / length(v)
            }
        }
    }
    A
}
```

For the two calls of this function below, write down the value that R will print:

```
> X
     [,1] [,2] [,3]
[1,]    3   NA    9
[2,]   NA    5    3
[3,]    3    7   NA
> Y
     [,1] [,2] [,3]
[1,]    4    3    9
[2,]    5    5    4
[3,]    3   NA    8
>
> mystery2(X,Y)
```

```
> mystery2(Y,X)
```

6. Write an R function called sum_with_exception that takes as arguments a vector of numbers and a single number. You may assume that none of the numbers are NA. (You don't have to check that the arguments are really of this form.) The value it returns should be the sum of all the elements in the first argument except for any that are equal to the second element. Here is an example:

```
> sum_with_exception (c(3,1,5,2,5,1), 5)  # Should return 3 + 1 + 2 + 1
[1] 7
```

You may write this function in any way that you like, as long as it works correctly.

7. Write a function called sums_without_row_index which takes as its only argument a numeric matrix without any NA values (you don't have to check this), and returns a vector with the same number of elements as the number of rows in this matrix, with element i being the sum of the elements in row i of the matrix except for any that are equal to i. Here is an example:

```
> M <- matrix (c(2,3,1,1,2,2,1,7,8), nrow=3, ncol=3)
> M
     [,1] [,2] [,3]
[1,]    2    1    1
[2,]    3    2    7
[3,]    1    2    8
> sums_without_row_index(M)
[1]  2 10 11
```

Your function must use (in a useful way) the sum_with_exception function from the question above. Except for that restriction, you may write it in any way that you like, as long as it works correctly.

8. Write an R function called good_rows that extracts a subset of rows of a data frame that have as few NA values as possible, while keeping a minimum number of rows. This function should take a data frame, df, as its first argument, and the minimum number of rows needed from this data frame, needed, as its second argument. If the number of rows in df is less than or equal to the needed, the good_rows function should return the data frame unchanged, regardless of how many NA values it contains. Otherwise, good_rows may return a data frame with fewer rows, eliminating those with some NA values. It should first try eliminating all rows with NA for any column in the data frame, and return this if the number of rows remaining is at least needed. Otherwise, it should try eliminating all rows with two or more NA values, returning that data frame if it has at least needed rows. The good_rows function should resort to allowing more and NA values in a row until the number of rows remaining is at least needed.

   Here are some example calls of this function:

```
> df
   a  b  c
1  4 NA NA
2  3  1 NA
3 NA NA  9
4  9  4  0
5 10  5  3
6 NA NA NA
> good_rows (df, 2)  # if we only need 2 rows we can use those with no NA
   a b c
4  9 4 0
5 10 5 3
> good_rows (df, 3)  # have to allow one NA value if we need 3 rows
   a b  c
2  3 1 NA
4  9 4  0
5 10 5  3
> good_rows (df, 4)  # have to allow two NA values if we need 4 rows
   a  b  c
1  4 NA NA
2  3  1 NA
3 NA NA  9
4  9  4  0
5 10  5  3
> good_rows (df, 6)  # have to keep rows with 3 NAs if we need 6 rows
   a  b  c
1  4 NA NA
2  3  1 NA
3 NA NA  9
4  9  4  0
5 10  5  3
6 NA NA NA
```

   Write your function on the next page.

9. In the five blank spaces below, write what R will print if the commands shown are entered one after the other into the R console. (Note: The > and + characters at the beginnings of lines are the R command prompts, not something that was typed in.)

```
> smaller <- function (P) UseMethod("smaller")
> smaller.minnie <- function (P) if (min(P$a) < min(P$b)) P$a else P$b
> smaller.maxime <- function (P) if (max(P$a) < max(P$b)) P$a else P$b
>
> shift <- function (P) UseMethod("shift")
> shift.minnie <- function (P) {
+     P$a <- P$a - min(P$a); P$b <- P$b - min(P$b)
+     P
+ }
> shift.maxime <- function (P) {
+     P$a <- P$a - max(P$a); P$b <- P$b - max(P$b)
+     P
+ }
>
> print.minnie <- function (P) cat("Minnie",P$a,":",P$b,"\n")
> print.maxime <- function (P) cat("Maxime",P$a,":",P$b,"\n")
>
> new_minnie <- function (a,b) {
+     P <- list(a=a,b=b)
+     class(P) <- "minnie"
+     P
+ }
> new_maxime <- function (a,b) {
+     P <- list(a=a,b=b)
+     class(P) <- "maxime"
+     P
+ }
>
> X <- new_minnie (c(3,4,1), c(2,2,3))
> smaller(X)


> shift(X)


> X <- new_maxime (c(3,4,1), c(2,2,3))
> smaller(X)


> shift(X)


> Y <- new_maxime (1:20, 1:200)
> length(smaller(shift(Y)))
```

10. Write an R function to simulate the exchange of email messages in one day between someone we'll call "Joe" and someone we'll call "Sue".

    Each morning, at the same time, Joe and Sue wake up. They stay awake for some period of time (the same for both), which we will call `wake_time`. Joe checks his email at various times while awake, with the intervals between email checks (and the time from waking to the first check) being distributed according to the exponential distribution with rate `joe_rate`, which you can generate a value from with `rexp(1,joe_rate)`. Similarly, Sue checks her email while awake, with the intervals between checks being distributed according to the exponential distribution with rate `sue_rate`.

    When Joe or Sue checks their email, they see if they got a message from the other person since the last time they checked, and if they did, they immediately reply to the other person. The time for Joe or Sue to check their email, to reply, and for the reply to be delivered are all very small compared to the average interval between when they check their email, so we will consider that these things happen instantly for this simulation. Joe and Sue only send email in reply to a new email from the other person. If they check their email and don't see a new message, they do nothing. When they wake up, one of them will have an unread email from the other, sent the previous day, which they will see the first time they check their email.

    Your simulation function should have `wake_time`, `joe_rate`, and `sue_rate` as arguments, along with a final argument called `last_sender`, either `"joe"` or `"sue"`, which identifies which of them sent the last email the previous day. The value returned by your simulation function should be the total number of emails read by either Joe or Sue during the day.

    Write your function below, or continuing on to the next page.