MTH208: Worksheet 5

Benchmarking R code and coding efficiently in R

At this point, we have some experience with writing R codes for various tasks. As we learn to write more and more complicated code, it is important to know how to optimize the code.

Problems

1. Recall Problem 4 from Worksheet 3, where the task was to obtain the average number of attempts it takes to blow out the candles on my 25th birthday:

```
attempts <- function(age) {
  count <- 0
  remain <- age # age no. of candles remain in the beginning
  while(remain > 0) {
    count <- count + 1 # randomly choose any number between 1 and remain
    blown_out <- sample(1:remain, size = 1)
    remain <- remain - blown_out
  }
  return(count)
}
att_vec <- numeric(length = 1e3)
for(i in 1:1e3) {
  att_vec[i] <- attempts(25)
}
```

   Lets think about how we can make this code a bit faster. The `attempts()` function seems like we may not be able to make it faster. However, we can replace the replications using `for` loop with `replicate`. That is, use the following code to replicate the experiment 10^3 times.

   Replace the `for` loop in the example with the `replicate` function.

2. A natural question to ask is, which of the two codes are faster? For this we will use the library `rbenchmark` and the function `benchmark()` in there.

```
library(rbenchmark)
benchmark({
  att_vec <- numeric(length = 1e3)
  for(i in 1:1e3) {
```

```
    att_vec[i] <- attempts(25)
  }
},
replicate(1e3, attempts(25)), replications = 100)
```

Which of the two ways of running the loop is faster?

3. Repeat the above for 10^4 reps instead of 10^3 and set `replications = 20`. Do you notice any difference?

4. Many students do the following instead of the usual `for` loop:

```
att_vec <- NULL
for(i in 1:1e4) {
  att_vec <- c(att_vec, attempts(25))
}
```

Benchmark the above with the other two methods for `replications = 25`. What do you learn?

5. To improve performance for loop-heavy computations, we can use C++ within R via the `Rcpp` package, which is particularly useful for numerical simulations like the birthday candle problem. Follow these steps:

   a. Install `Rcpp` (`install.packages("Rcpp")`) and ensure a C++ compiler is installed (e.g., Rtools for Windows, Xcode for Mac, or `r-base-dev` for Linux).

   b. Save the following C++ code as `attempts.cpp`:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector attempts_cpp(int age, int n_reps) {
  NumericVector results(n_reps); // Preallocate vector for n_reps results
  for(int i = 0; i < n_reps; ++i) {
    int count = 0;
    int remain = age;
    while(remain > 0) {
      count++;
      // Generate random integer between 1 and remain (inclusive)
      int blown_out = floor(R::runif(1, remain + 1));
      remain -= blown_out;
    }
    results[i] = count;
  }
  return results;
}
```

   c. In R, source the C++ code and benchmark it against the `for` loop and `replicate` versions for

10^3 repetitions with `replications = 100`:

```
library(Rcpp)
library(rbenchmark)
sourceCpp("attempts.cpp")

attempts <- function(age) {
  count <- 0
  remain <- age
  while(remain > 0) {
    count <- count + 1
    blown_out <- sample(1:remain, size = 1)
    remain <- remain - blown_out
  }
  return(count)
}

benchmark(
  "For Loop" = {
    att_vec <- numeric(length = 1e3)
    for(i in 1:1e3) {
      att_vec[i] <- attempts(25)
    }
  },
  "Replicate" = {
    att_vec <- replicate(1e3, attempts(25))
  },
  "C++ via Rcpp" = {
    att_vec <- attempts_cpp(25, 1e3)
  },
  replications = 100,
  columns = c("test", "replications", "elapsed", "relative", "user.self", "sys.self")
)
```

d. Repeat the benchmark for 10^4 repetitions with `replications = 20`. What is the speedup achieved by the C++ version (check the `relative` column)? Why is the C++ version faster? Discuss how this relates to memory allocation (each number uses approximately 8 bytes) and R's loop overhead. Hint: C++ is compiled, not interpreted, and preallocates memory efficiently.

6. Write R code to draw 10^4 realizations from Uniform $(0, 1)$ one at a time (using a loop) and another code to draw the numbers all at once. Which one is faster?

7. Create a matrix of size $n \times m$ for your choice of n and m, that is made from random numbers between (0,1). (Recall function `runif()`). Using `colMeans()` function and the `apply()` function, find two ways to determine the mean of each column. Which method is faster? Is the answer dependent on n and

m?

8. Theoretically, assess roughy how much memory each of the objects below will take and then verify using `object.size()`.

```
num1 <- numeric(length = 1e3)
num2 <- numeric(length = 1e6)

mat1 <- matrix(runif(100*1000), nrow = 100, ncol = 1000)
mat2 <- matrix(0, nrow = 100, ncol = 1000)

arr <- array(0, dim = c(100,100,100))
```

General Guidelines on Efficient Coding in R

Avoid loops when possible in R. If loops cannot be avoided, then make sure to allocate memory for the loop. This is a feature of high-level languages in R. Sometimes we cannot avoid loops. In such cases, we could switch to implementing C++ within R.

You should know how much memory a certain object will take. A single number takes roughly 8 bytes of memory. 1024 bytes = 1KB, 1024 KB = 1MB, 1024 MB = 1GB. Thus a vector of length n takes around $8n$ bytes of memory.

Be careful about numerical instabilities. Sometimes numbers get too small or too large for R to handle. Make transformations to avoid this.