# Coding More Efficiently in R

Yang Yang

University of Minnesota

February 21, 2014

# Static vs. Dynamic Memory

Static

- It is always better to declare the size of the data beforehand
- Allocate memory before the code is executed
- ```
  data <- rep(0, 100)
  for (i in 1:100) {
      data[i] = i
  }
  ```

Dynamic

- The following is bad
- ```
  data <- NULL
  for (i in 1:100) {
      cbind(data, i)
  }
  ```

# Euclidean Distances

- $n \times n$ points
- Calculate the mean Euclidean distance between all pairs of points
- Can be computationally intensive when $n$ is large

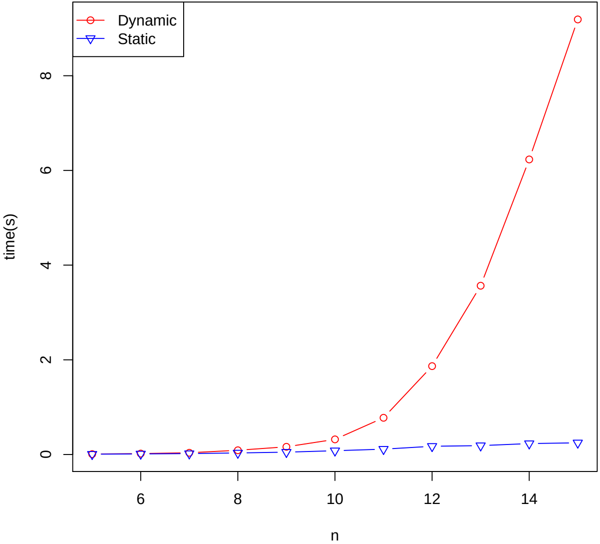
# Loop: Dynamic Memory

```
loop.dynamic <- function(n) {
    dis <- NULL
    for (x1 in 1:n) {
        for (y1 in 1:n) {
            for (x2 in 1:n) {
                for (y2 in 1:n) {
                  dis <- c(dis, sqrt((x1 - x2)^2 +
                    (y1 - y2)^2))
                }
            }
        }
    }
    mean(dis)
}
```

# Loop: Static Memory

```
loop.static <- function(n) {
    dis <- rep(0, n^4)
    pos = 0
    for (x1 in 1:n) {
        for (y1 in 1:n) {
            for (x2 in 1:n) {
                for (y2 in 1:n) {
                  dis[pos = pos + 1] <- sqrt((x1 -
                    x2)^2 + (y1 - y2)^2)
                }
            }
        }
    }
    mean(dis)
}
```

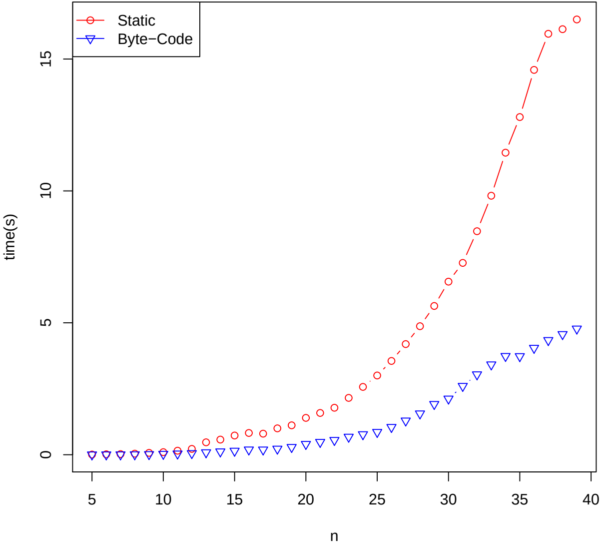
# Dynamic vs. Static



# Byte Code Compiler

- R is a high-level language
- R code→byte code→machine code
- Byte-code is lower-level language, faster
- `library(compiler)`
- Compile the function into byte code
- `loop.static.c <- cmpfun(loop.static)`

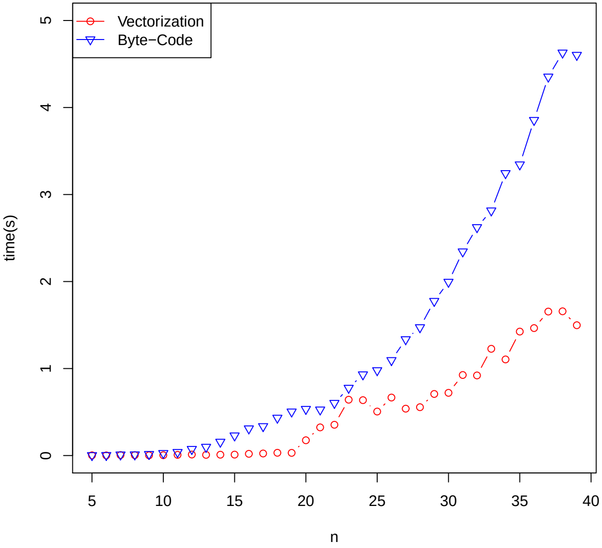
# Static vs. Byte Code



# Vectorization

```
vectorization <- function(n) {
    comb <- expand.grid(1:n, 1:n, 1:n, 1:n)
    mean(sqrt((comb[, 1] - comb[, 2])^2 + (comb[, 3] -
        comb[, 4])^2))
}
```

# Byte Code vs. Vectorization



# `lapply()`, `apply()`

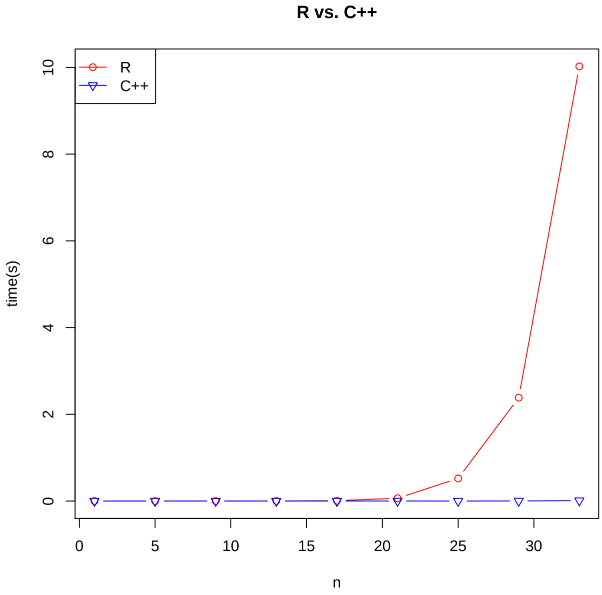
- **Myth**: `lapply()` and `apply()` are superior than `for()` loop
- Bad code is slow no matter what you use
- `lapply()` and `apply()` allocate memory beforehand; Easier to understand
- `lapply()` is a little faster since it uses more `C`
- ```
  colSum(matrix)
  ```

  is better than

  ```
  apply(matrix, 2, sum)
  ```
  ```

# Parallel Computing

- By default `R` only uses one core of the CPU
- Modern CPUs have up to 72 cores
- ```r
  library(multicore)

  mclapply(list, fun)
  ```
- for more details see Abhirup Mallik's Fall 2013 lit sem talk: `https://github.com/abhirupkgp/parallelseminar`
  ```

# What if Vectorization is Impossible?

- Recursive algorithms
- Use results from previous iterations
- Example:

$$f(x) = \begin{cases} 1 & \text{if } x \leq 2 \\ f(x-1) + f(x-2) & \text{if } x > 2 \end{cases}$$

# Good Old Fashioned R Function

```
fibo <- function(n) {
    if (n < 3)
        return(1)
    return(fibo(n - 1) + fibo(n - 2))
}
```

# Alternative: C++ (Rcpp)

- `library(Rcpp)`
- 
```
cppFunction('
int cfibo(int n) {
if (n < 3) return(1);
return(cfibo(n-1) + cfibo(n-2));
}
')
```

# R vs. C++

- Generate the 25th Fibonacci number. Run 100 replications
- ```
  library(rbenchmark)
  benchmark(fibo(25), cfibo(25))[, 1:3]
  ```

  ```
          test replications elapsed
  2 cfibo(25)          100   0.017
  1  fibo(25)          100  19.661
  ```
  ```


R vs. C++
R
C++
time(s)
0
2
4
6
8
10
0
5
10
15
20
25
30
n

# Take Home Message

- Static memory
- Byte code
- Vectorization
- Parallel processing
- `C++`

# Thank you

Questions?