Computational Complexity and Memory Allocation Slides

Adam Kaplan

March 30, 2022

Efficiency in R

- Why is some code "fast" and other code "slow" in R?

Efficiency in R

- Why is some code "fast" and other code "slow" in R?
- Two main culprits:

Efficiency in R

- Why is some code “fast” and other code “slow” in R?
- Two main culprits:
    - Bad algorithms,

Efficiency in R

- Why is some code "fast" and other code "slow" in R?
- Two main culprits:
  - Bad algorithms,
  - Inefficient memory usage.

Efficiency in `R`

- Why is some code “fast” and other code “slow” in `R`?
- Two main culprits:
  - Bad algorithms,
  - Inefficient memory usage.
- Does not suffice to use and understand `apply()` family of functions.

Efficiency in R

- Why is some code "fast" and other code "slow" in R?
- Two main culprits:
  - Bad algorithms,
  - Inefficient memory usage.
- Does not suffice to use and understand `apply()` family of functions.
- $\implies$ Need to understand basics of computational complexity and memory allocation.

Computational Complexity

Basics of Computational Complexity

- A theoretically rigorous way to measure "speed" of an algorithm as a function of input size.

Basics of Computational Complexity

- A theoretically rigorous way to measure "speed" of an algorithm as a function of input size.
- Put simply: How "fast" can $f(x_1, \ldots, x_n)$ be evaluated as a function of n.

Basics of Computational Complexity

- A theoretically rigorous way to measure "speed" of an algorithm as a function of input size.
- Put simply: How "fast" can $f(x_1, \ldots, x_n)$ be evaluated as a function of n.
- Before we go any deeper, what do we mean by speed?

Basics of Computational Complexity

- A theoretically rigorous way to measure "speed" of an algorithm as a function of input size.
- Put simply: How "fast" can $f(x_1, \ldots, x_n)$ be evaluated as a function of n.
- Before we go any deeper, what do we mean by speed?
  - How many inputs x_i does f need to go over.

Basics of Computational Complexity

- A theoretically rigorous way to measure “speed” of an algorithm as a function of input size.
- Put simply: How “fast” can $f(x_1, \ldots, x_n)$ be evaluated as a function of n.
- Before we go any deeper, what do we mean by speed?
  - How many inputs x_i does f need to go over.
- Why can’t we just measure time from start to end?

Basics of Computational Complexity

- A theoretically rigorous way to measure “speed” of an algorithm as a function of input size.
- Put simply: How “fast” can $f(x_1, \ldots, x_n)$ be evaluated as a function of n.
- Before we go any deeper, what do we mean by speed?
  - How many inputs x_i does f need to go over.
- Why can’t we just measure time from start to end?
  - Many other factors impact the speed with which your code runs; harder to pin down issues.

Basics of Computational Complexity

- A theoretically rigorous way to measure "speed" of an algorithm as a function of input size.
- Put simply: How "fast" can $f(x_1, \ldots, x_n)$ be evaluated as a function of n.
- Before we go any deeper, what do we mean by speed?
  - How many inputs x_i does f need to go over.
- Why can't we just measure time from start to end?
  - Many other factors impact the speed with which your code runs; harder to pin down issues.
- What can you do?

Basics of Computational Complexity

- A theoretically rigorous way to measure "speed" of an algorithm as a function of input size.
- Put simply: How "fast" can $f(x_1, \ldots, x_n)$ be evaluated as a function of n.
- Before we go any deeper, what do we mean by speed?
  - How many inputs x_i does f need to go over.
- Why can't we just measure time from start to end?
  - Many other factors impact the speed with which your code runs; harder to pin down issues.
- What can you do?
  1. Find more appropriate algorithms,

Basics of Computational Complexity

- A theoretically rigorous way to measure "speed" of an algorithm as a function of input size.
- Put simply: How "fast" can $f(x_1, \ldots, x_n)$ be evaluated as a function of n.
- Before we go any deeper, what do we mean by speed?
  - How many inputs x_i does f need to go over.
- Why can't we just measure time from start to end?
  - Many other factors impact the speed with which your code runs; harder to pin down issues.
- What can you do?
  1. Find more appropriate algorithms,
  2. Vectorize whenever possible (why? stay tuned),

Basics of Computational Complexity

- A theoretically rigorous way to measure "speed" of an algorithm as a function of input size.
- Put simply: How "fast" can $f(x_1, \ldots, x_n)$ be evaluated as a function of n.
- Before we go any deeper, what do we mean by speed?
  - How many inputs x_i does f need to go over.
- Why can't we just measure time from start to end?
  - Many other factors impact the speed with which your code runs; harder to pin down issues.
- What can you do?
  1. Find more appropriate algorithms,
  2. Vectorize whenever possible (why? stay tuned),
  3. If possible, parallelize.

Computational Complexity Examples

```r
# Block 1: Simple for loop
for (i in 1:n) {
  foo()
}

# Block 2: Sequential for loops
for (i in 1:n) {
  foo()
}
for (i in 1:n) {
  bar()
}
```

```r
# Block 3: Nested for-loops
for (i in 1:n) {
  foo()
  for (i in 1:n) {
    bar()
  }
}

# Block 4: Simple bootstrap
for (i in 1:b) {
  # Hint: mle with BFGS takes O(n^2)
  estimate <- mle(X_1, ..., X_n, method = BFGS)
}
```

Memory Allocation in R

Memory Allocation in R

- Now we are algorithm experts and will measure the complexity of every piece of code.

Memory Allocation in R

- Now we are algorithm experts and will measure the complexity of every piece of code.
- Is this enough? Unfortunately no.

Memory Allocation in R

- Now we are algorithm experts and will measure the complexity of every piece of code.
- Is this enough? Unfortunately no.
- Even if the algorithms are efficient, we can lose real-life performance due to inefficient memory usage.

Memory Allocation in R

- Now we are algorithm experts and will measure the complexity of every piece of code.
- Is this enough? Unfortunately no.
- Even if the algorithms are efficient, we can lose real-life performance due to inefficient memory usage.
- Will go over basics of memory (in R), data types, and vectorization to motivate efficiency prescriptions.

What is memory?

- Computer consists of three main parts: CPU core(s) (different circuits), Memory (RAM, Hard drive), GPU (can ignore for this session)

What is memory?

- Computer consists of three main parts: CPU core(s) (different circuits), Memory (RAM, Hard drive), GPU (can ignore for this session)
- CPU is wicked fast.

What is memory?

- Computer consists of three main parts: CPU core(s) (different circuits), Memory (RAM, Hard drive), GPU (can ignore for this session)
- CPU is wicked fast.
- Many different types of memory: Volatile (RAM, Caches), Non-volatile (Hard drive, USB, CDs, . . .).

What is memory?

- Computer consists of three main parts: CPU core(s) (different circuits), Memory (RAM, Hard drive), GPU (can ignore for this session)
- CPU is wicked fast.
- Many different types of memory: Volatile (RAM, Caches), Non-volatile (Hard drive, USB, CDs, ...).
- Every R session lives on RAM memory.

What is memory?

- Computer consists of three main parts: CPU core(s) (different circuits), Memory (RAM, Hard drive), GPU (can ignore for this session)
- CPU is wicked fast.
- Many different types of memory: Volatile (RAM, Caches), Non-volatile (Hard drive, USB, CDs, . . .).
- Every R session lives on RAM memory.
- RAM is actually very very fast for memory standards, but painstakingly slow compared to CPUs, particularly saving new things to it (writes).

What is memory?

- Computer consists of three main parts: CPU core(s) (different circuits), Memory (RAM, Hard drive), GPU (can ignore for this session)
- CPU is wicked fast.
- Many different types of memory: Volatile (RAM, Caches), Non-volatile (Hard drive, USB, CDs, . . .).
- Every R session lives on RAM memory.
- RAM is actually very very fast for memory standards, but painstakingly slow compared to CPUs, particularly saving new things to it (writes).
- To write efficient code, we **need to avoid unnecessary memory writes!**

Memory Usage in R

- What happens when I delete an object with `rm()`?

Memory Usage in R

- What happens when I delete an object with `rm()`?
  - You tell R that this reference to an object is no longer needed.

Memory Usage in R

- What happens when I delete an object with `rm()`?
  - You tell R that this reference to an object is no longer needed.
  - If there are no more, it will be "garbage collected" *automatically*.

Memory Usage in R

- What happens when I delete an object with `rm()`?
  - You tell R that this reference to an object is no longer needed.
  - If there are no more, it will be “garbage collected” *automatically*.
- **Big culprit:** Copying of objects.

Memory Usage in R

- What happens when I delete an object with `rm()`?
  - You tell R that this reference to an object is no longer needed.
  - If there are no more, it will be "garbage collected" *automatically*.
- **Big culprit:** Copying of objects.
  - R uses a "copy if modified" framework so if `y <- x`, and `y[1] <- 0`, `y` will no longer point to `x`, but will allocate a new chunk of memory and point to that.

Memory Usage in R

- What happens when I delete an object with `rm()`?
  - You tell R that this reference to an object is no longer needed.
  - If there are no more, it will be "garbage collected" *automatically*.
- **Big culprit:** Copying of objects.
  - R uses a "copy if modified" framework so if `y <- x`, and `y[1] <- 0`, `y` will no longer point to `x`, but will allocate a new chunk of memory and point to that.
  - $\Longrightarrow$ Avoid implicit and explicit copying of objects whenever possible.

Memory Usage in R

- What happens when I delete an object with `rm()`?
  - You tell R that this reference to an object is no longer needed.
  - If there are no more, it will be "garbage collected" *automatically*.
- **Big culprit:** Copying of objects.
  - R uses a "copy if modified" framework so if `y <- x`, and `y[1] <- 0`, `y` will no longer point to `x`, but will allocate a new chunk of memory and point to that.
  - $\Longrightarrow$ Avoid implicit and explicit copying of objects whenever possible.
  - $\Longrightarrow$ "Vectorizing" is mostly faster because it avoids this implicit copying.

Implicit copying of objects

```
# Example from: http://adv-r.had.co.nz/memory.html
library(pryr)
x <- data.frame(matrix(runif(100 * 1e4), ncol = 100))
medians <- vapply(x, median, numeric(1))

for (i in seq_along(medians)) {
  x[, i] <- x[, i] - medians[i]
  print(c(address(x), refs(x)))
}
```

Explicit copying objects

- Define the size and type of object to avoid copying object.

Explicit copying objects

- Define the size and type of object to avoid copying object.
- Example: creating a vector of ones.

```
f1 <- function() { # adding ones one by one
  x <- 1.0
  for (i in 2:10000) {
    x <- c(x, 1.0)
  }
  return(x)
}
f2 <- function() { # define length + type of vector
  x <- numeric(10000) # can be character(), interger() etc
  for (i in 1:10000) {
    x[i] <- 1.0
  }
  return(x)
}
f3 <- function() { # vectorization
  x <- rep(1.0, 10000)
  return(x)
}
```

```r
bench::mark( # sum two columns for each row
  f1(), f2(), f3()
) %>% select(expression, median)
```

```
## # A tibble: 3 x 2
##   expression   median
##   <bch:expr> <bch:tm>
## 1 f1()         197.2ms
## 2 f2()           536us
## 3 f3()          17.1us
```

Functions in R

- From last time, arguments are not copied until modified, yay!

Functions in R

- From last time, arguments are not copied until modified, yay!
- What if we return a function in a function?

Functions in R

- From last time, arguments are not copied until modified, yay!
- What if we return a function in a function?
- **Turns out:** Functions save their surrounding environment!

Functions in R

- From last time, arguments are not copied until modified, yay!
- What if we return a function in a function?
- **Turns out:** Functions save their surrounding environment!
- So what? Imagine you create a large variable (`x <- 1:1e+50`) before you create your function, then your function will also copy the large variable into memory.

Functions in R

- From last time, arguments are not copied until modified, yay!
- What if we return a function in a function?
- **Turns out:** Functions save their surrounding environment!
- So what? Imagine you create a large variable (`x <- 1:1e+50`) before you create your function, then your function will also copy the large variable into memory.
- Now on to more applied examples of why data types matter and how to vectorize in R.

Memory Prescriptions

Memory Prescription 1: Data Types Matter

- List vs. Matrix vs. Dataframe (or Tibble)

Memory Prescription 1: Data Types Matter

- List vs. Matrix vs. Dataframe (or Tibble)
    - Matrix algebra is heavily optimized!

Memory Prescription 1: Data Types Matter

- List vs. Matrix vs. Dataframe (or Tibble)
  - Matrix algebra is heavily optimized!
  - Neither lists nor matrices have memory overhead.

Memory Prescription 1: Data Types Matter

- List vs. Matrix vs. Dataframe (or Tibble)
  - Matrix algebra is heavily optimized!
  - Neither lists nor matrices have memory overhead.
  - Dataframes have many specialized functions such as grouping that are much faster than DIY approaches on lists or matrices.

```r
x <- runif(10000)
y <- rnorm(10000)
DF <- data.frame(x = x, y = y) # dataframe
TIB <- tibble::as_tibble(DF) # tidyverse tibble
MAT <- cbind(x, y) # matrix
```

```r
bench::mark( # sum two columns for each row
  apply(DF, 2, sum),
  apply(TIB, 2, sum),
  apply(MAT, 2, sum),
  colSums(MAT) # Implemented in C
) %>% select(expression, median)
```

```
## # A tibble: 4 x 2
##   expression            median
##   <bch:expr>          <bch:tm>
## 1 apply(DF, 2, sum)    328.3us
## 2 apply(TIB, 2, sum)   316.3us
## 3 apply(MAT, 2, sum)   193.5us
## 4 colSums(MAT)          30.5us
```

Memory Prescription 2: Vectorize

- "Vectorize" your codes as much as you can.

Memory Prescription 2: Vectorize

- “Vectorize” your codes as much as you can.
- Most `R` functions allow you to vectorize by default.