Lecture 29: Nov 30, 2018

Algorithm Complexity

- *Empirical Run Time*
- *Theoretical (Big Oh) Run Time*
- *Big Oh for Code*

James Balamuta
STAT 385 @ UIUC

Announcements

- **Group Project Final Report, Demo Video,** and **Peer Evaluation** due **Tuesday, December 18th** at **10:00 PM**
  - Details: http://stat385.stat.illinois.edu/group-projects/
- **hw10** due **Friday, December 7th** at **6:00 PM**
- **Quiz 14** covers Week 13 contents @ CBTF.
  - Window: Dec 4th - Dec 6th
  - Sign up: https://cbtf.engr.illinois.edu/sched
- Want to review your homework or quiz grades? **Schedule an appointment.**

Last Time

- **Motivation**
  - R packages lower the bare to sharing code
  - Provide means to easily re-use functions in other analyses
- **Creating an R Package**
  - RStudio has built in package templates
  - Possible to use **devtools::create_package()** and **package.skeleton()**
- **Structure of R Packages**
  - Packages require **NAMESPACE** + **DESCRIPTION** files **R/** + **man/** folders
- **Unit Testing**
  - Standardize testing and incorporate it into an automatic framework

Lecture Objectives

- **Design** and **interpret** a benchmark
- **Describe** the difference between empirical and theoretical complexity analysis
- **Calculate** one-variable Big Ohs

Empirical Run Time

There are many ways to implement an algorithm…

What's the **best way** to write an algorithm?

Definition:
Benchmarks provide quantifiable results to facilitate making data driven decisions as to how well a method performs.

Source

Source

```
sum_vec = function(x) {
  total_sum = 0
  for(i in seq_len(length(x))) {
    total_sum = total_sum + i
  }
  return(total_sum)
}
```

Finding Time

... elementary benchmark ...

```
out = system.time({Sys.sleep(1)})
out
#    user  system elapsed
#  0.280   0.077   1.001

out[3]
# elapsed
#  1.001
```

user: cpu time spent to launch user code within the active R process

system: cpu time spent in executing operating system calls within the active R process.

elapsed: wall clock time or the time from start to finish.

---

* elapsed ≠ system + user

Benchmarking Code

... more strict use of code ...

```
# Load Library
library("rbenchmark")

# Run benchmark
benchmark(testfun1 = somefun(), testfun2 = otherfun(), ... ,
          replications = 100)
```

The **user**, **system**, and total **elapsed** times similar to **Sys.time()**
The **cumulative sum of user** and **system times** of any child processes spawned to assist CPU execution. (*.child, *.self)

Using the Right Data Structure

... a case where how the data structures is important ...

```
# Set seed for reproducibility
set.seed(1337)
# Construct large matrix object
matrix.op = matrix(rnorm(10000*100), 10000, 100)
# Convert matrix object to data.frame
dataframe.op = as.data.frame(matrix.op)

library("rbenchmark")
bench_ops = benchmark(mat.op = apply(matrix.op, 2, sd),
                      df.op  = apply(dataframe.op, 2, sd))
bench_ops
#     test replications elapsed relative user.self sys.self user.child sys.child
# 2    df.op          100   3.692    1.614     3.276    0.397      0.000     0.000
# 1 mat.op          100   2.288    1.000     2.054    0.216      0.342     0.065
```

{ or (

… the cost of an interpreted language…

```
# Different R implementations
f = function(n, x = 1) for (i in 1:n) x = 1/(1+x)
g = function(n, x = 1) for (i in 1:n) x = (1/(1+x))
h = function(n, x = 1) for (i in 1:n) x = (1+x)^(-1)
j = function(n, x = 1) for (i in 1:n) x = {1/{1+x} }
k = function(n, x = 1) for (i in 1:n) x = 1/{1+x}
Rcpp::cppFunction(code = 'int d(int n, double x = 1.0) {
                    for (int i = 0; i < n; ++i) x = 1/(1+x);
                    return x;
                }')
```

Based off of work by Dirk Eddelbuettel and Christian Robert

{ or (

... the cost of an interpreted language...

```
library("rbenchmark")
N = 1e6                                  # Number of Loop Iterations
bench_curly =                            # Test Approaches
  benchmark(f(N, 1), g(N, 1), h(N, 1), j(N, 1), k(N, 1), d(N, 1),
            order = "relative",          # Show fastest / lowest first
            replications = 20)           # Run each function x times
bench_curly
```

	test <fctr>	replications <int>	elapsed <dbl>	relative <dbl>	user.self <dbl>	sys.self <dbl>	user.child <dbl>	sys.child <dbl>
Rcpp	d(N, 1)	20	0.139	1.000	0.138	0.000	0	0
(	f(N, 1)	20	0.816	5.871	0.791	0.024	0	0
{	k(N, 1)	20	0.825	5.935	0.812	0.016	0	0
{{	j(N, 1)	20	0.829	5.964	0.821	0.008	0	0
((	g(N, 1)	20	0.839	6.036	0.814	0.025	0	0
Neg Power	h(N, 1)	20	1.327	9.547	1.298	0.027	0	0

Object Growth

... preallocation of space is important !!!

```
append_elements = function(n) {
  vec = numeric(0)                           # Vector of length 0
  for(i in seq_len(n)) vec = c(vec, i) # Append results
  vec
}
preallocate_elements = function(n) {
  vec = rep(NA, n)                           # Vector of length n
  for(i in seq_len(n)) vec[i] = i          # Access and update value i
  vec
}
vectorized_element = function(n) { seq_len(n) }
```

Object Growth

... preallocation of space is important !!!

```
n = 10000
bench_growth =
  benchmark(append       = append_elements(n),
            preallocate  = preallocate_elements(n),
            vectorized   = vectorized_element(n))
bench_growth
```

test <fctr>	replications <int>	elapsed <dbl>	relative <dbl>	user.self <dbl>	sys.self <dbl>	user.child <dbl>	sys.child <dbl>
append	100	22.721	**7573.667**	22.615	0.082	0	0
preallocate	100	0.097	**32.333**	0.097	0.000	0	0
vectorized	100	0.003	**1.000**	0.003	0.000	0	0

Microbenchmarking Code

... drilling down to each iteration ...

```
# Load Library
library("microbenchmark")

# Run benchmark
microbenchmark(testfun1 = somefun(),
               testfun2 = otherfun(), ... ,
               times = 100)
```

Microbenchmarking Allocation

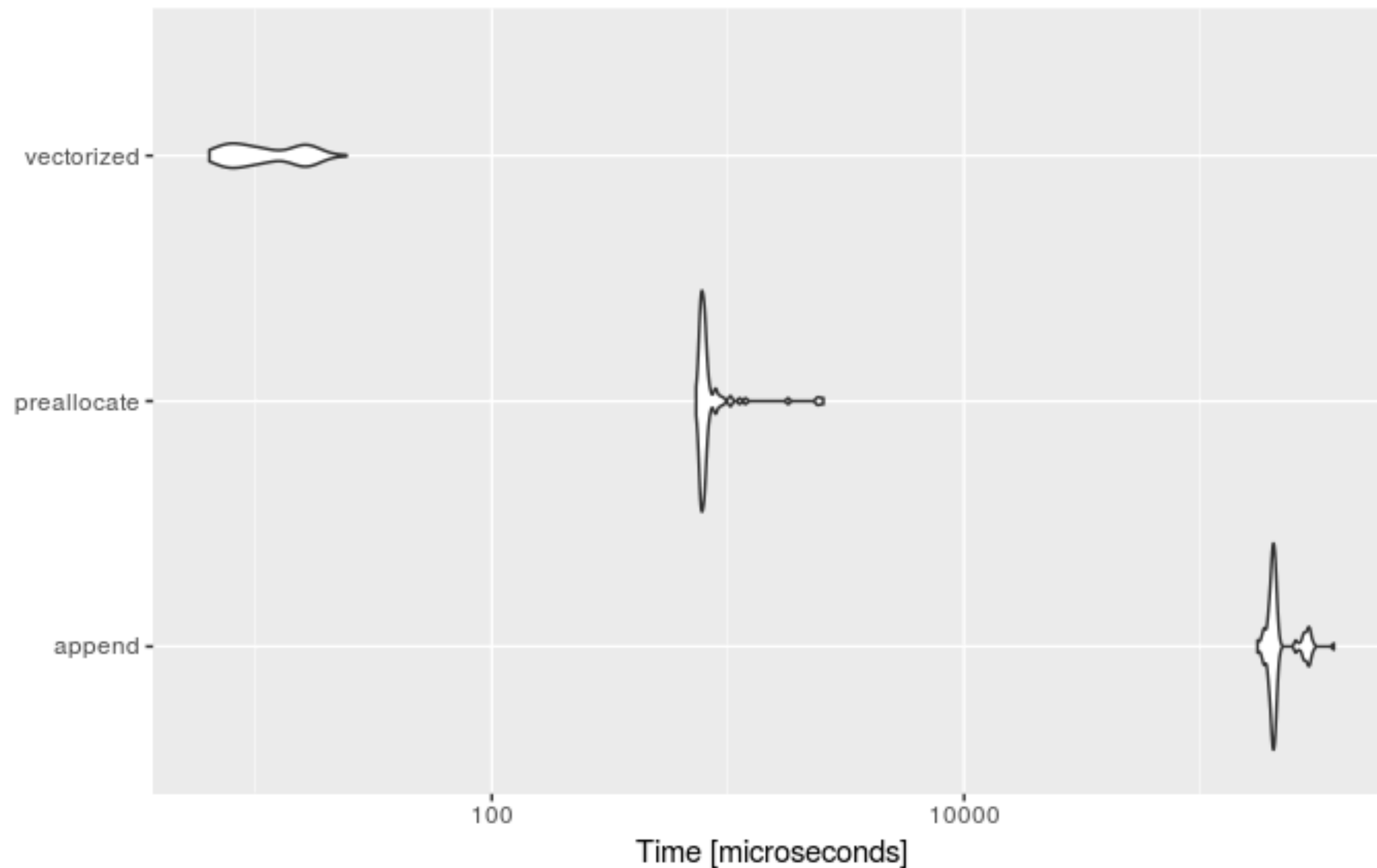
… revisiting allocation …

```
n = 10000
microbench_growth =
  microbenchmark(append      = append_elements(n),
                 preallocate  = preallocate_elements(n),
                 vectorized   = vectorized_element(n))
microbench_growth
# Unit: microseconds
#       expr        min          lq         mean      median          uq        max neval
#     append 174882.127 199656.7165 217195.33755 204863.8140 210854.7280 368289.831   100
# preallocate    733.297    768.0365    865.13745    787.7110    812.5395   2535.898   100
#  vectorized      6.371      7.9465     11.61857     10.3005     15.7525     24.258   100
```

217195.33755 *microseconds* is 0.21719533755 *seconds* per iteration.

Visualizing Microbenchmark

... violin graph showing distribution ...




```
autoplot(microbench_growth)
```

Theoretical Run Time

Definition:
Big O notation or *Landau's symbol* describes the amount of time an algorithm needs to run by analyzing asymptotic behavior of functions. This describes how fast an algorithm grows relative to the input (e.g. sample size *n*) as it goes to infinity (∞).

$$O(\cdot)$$

…Not….

Indepth

... answering Big Oh's definitions ...

Let $T(n)$ be called the *run time and*
$O(n)$ be the **asymptotic** *run time*.

Nonformal Definition

$T(n)$ is considered to be the **exact** complexity of an algorithm as a function of data size n.

$F(n)$ acts as an upper-bound in regards to that complexity.

Need to pick the smallest $F(n)$ to obtain the ***least*** upper bound.

Formal Definition:

$T(n) = O(F(n))$ as $n \to \infty$ if and only if for every constant $M > 0$ there exists a real number n_0 such that

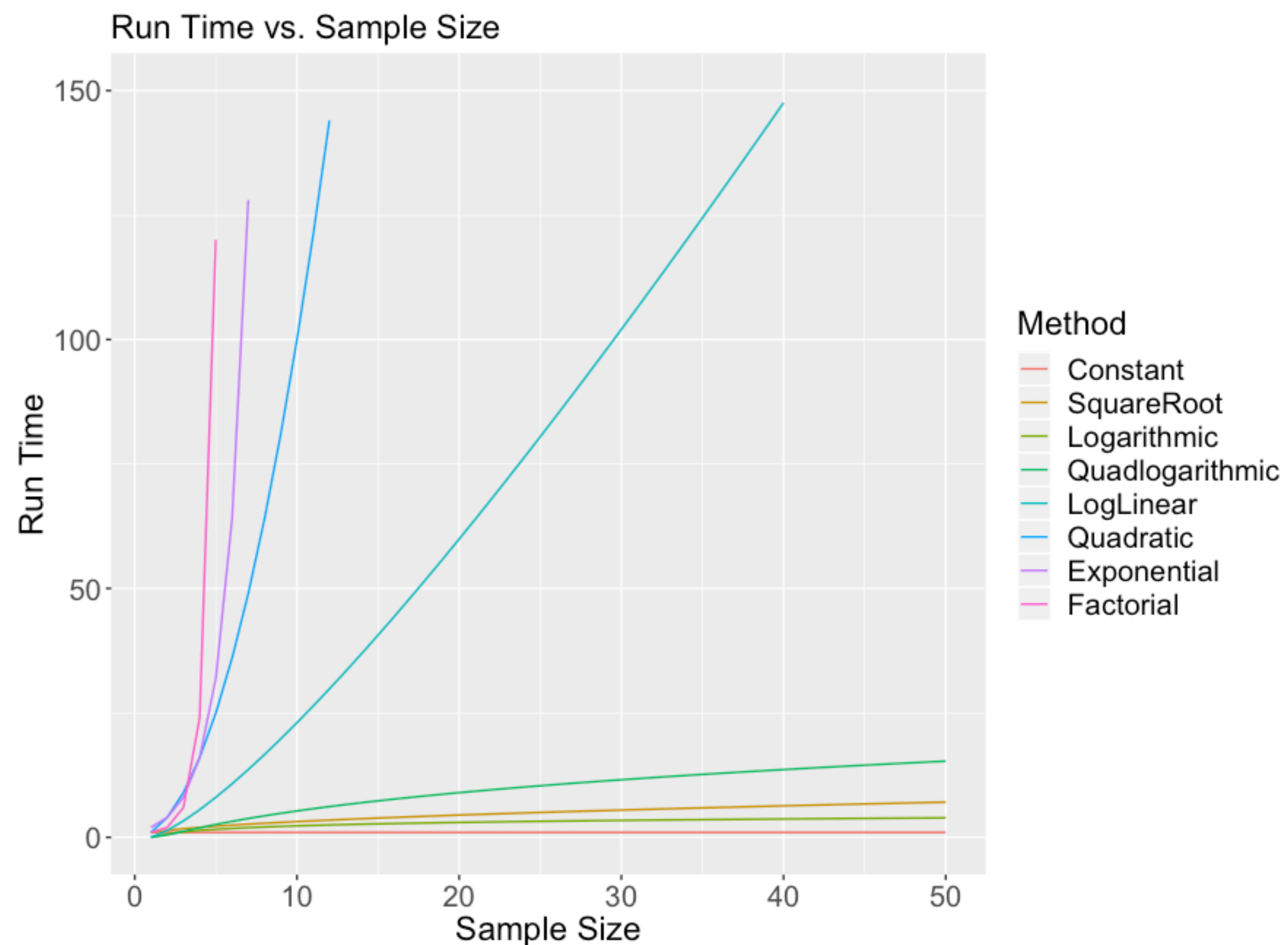
$$|T(n)| \leq M|F(n)|$$

for all $n \geq n_0$.

Notations and Run Time

... Common Big Oh's ...

Notation	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(\log(n)^k)$	Polylogarithmic
$O(n\log(n))$	Log Linear
$O(\sqrt{n})$	Square Root
$O(n)$	Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^p)$	Polynomial
$O(c^n)$	Exponential
$O(n!)$	Factorial



---

* Even with a small sample size, certain methods take considerably long.

** You may have multiple variables in big oh notation. We'll only use one variable.

Strategy

... calculating Big Oh ...

1. Identify Algorithm Complexity, e.g. $T(n) = 4n^2 + 2n + 1$

2. Find highest term e.g. $O(4n^2)$

3. Remove constants e.g. $O(n^2)$

Equivalence of Big Ohs

… alignment of multiple Big Oh …

$O(8n^2)$
$O(n(n - 1)/2))$
$O(5n^2 + n - 1)$
$O(n^2 + n \log n + 9)$
$O((n - 1)(n + 1))$

$O(n^2)$

Your Turn

What is the Big Oh of…

1. $O(n)$
2. $O(n(n - 1)(n + 1)/2))$
3. $O(5n^5 + 2n^3 - n^2 +1)$
4. $O(n \log n + n + 2)$
5. $O((1 + 1/n)^n)$

Theoretical Big Oh for Code

Big Oh Rules

... some useful properties ...

If $T_1(n) = O(F(n))$ and $T_2(n) = O(G(n))$, then:

1. $c*T_1(n) = O(F(n))$
2. $c(n) * T_1(n) = O(c(n) * F(n))$
3. $T_1(n) + T_2(n) = \max(O(F(n)), O(G(n)))$
4. $T_1(n) * T_2(n) = O(F(n)) * O(G(n))$

$$O(1)$$

… constant run time for simple statements …

```
# Pieces of code
statement 1
statement 2
...
statement j
```

$$T(n) = T(statement_1) + T(statement_2) + \ldots + T(statement_j)$$

---

* This means it is possible to have a Big Oh of 1/n, which is really 0; however, the algorithm would be empty.

$$O(n)$$

… linear run time for **iteration** …

```r
# Iterate n times
for(i in seq_len(n)) {
  # Statement
}
```

Statement is repeated *n* times.
Thus, we have $O(n)$.

$O(n)$

… example for summed values …

```
sum_vec = function(x) {      # Cost: 1
  total_sum = 0                # Cost: 1

  # Cost: 1 (Variable), N+1 (In Check), 2N (i = i + 1)
  for(i in seq_len(length(x))) {
    # Cost: 2N (1 addition, 1 assignment)
    total_sum = total_sum + i
  }

  return(total_sum)          # Cost: 1
}
```

The total time in this case is:

$T(n) = 1 + 1 + 1 + (n + 1) + 2n + 2n + 1 = 5n + 4$

which has a Big O of $O(n)$.

Polynomial Run Time

… multiple loops combined together …

```
for(i in seq_len(n)) {
  for(i in seq_len(n)) {
    for(i in seq_len(n)) {
      # statement
    }
  }
}
```

If one **for** loop is *n*, then what would three **for** loops give?

What's the Big-Oh if we
hide the loop in a function?

Composition Dependent

… what are the functions' big Ohs and how are they called ???

```
# Iterate n Times
for(i in seq_len(n)) {
  result = f(x)
}
```

If **f(x)** is:

- **O(1)**, then the composition is **O(n)**
- **O(n)**, then the composition is **O(n^2)**
- ….

$$\max\left(O\left(F\left(n\right)\right), O\left(G\left(n\right)\right)\right)$$

... run time for **if-else** ...

```
if (expression) { # True Case
  # statement 1
} else {          # False Case
  # statement 2
}
```

Either the **first statement** or the **second statement** will run. Hence, the worst-case or the longest run time is given by the **max between two cases**.

Your Turn

What is the Big Oh of the following algorithms:

mean:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

variance:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2$$

matrix multiplication (naively):

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj}$$

Recap

- **Empirical Run Time**
  - Timing the duration that code runs for using a benchmark
- **Theoretical Run Time**
  - Big-Oh provides a framework for classifying algorithmic complexity.
  - Emphasis is on the **worst** case run-time.
- **Theoretical Big Oh for Code**
  - Big Oh varies depending on the underlying code structure.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License