CSC 121: Computer Science for Statistics

Radford M. Neal, University of Toronto, 2017

http://www.cs.utoronto.ca/∼radford/csc121/

Week 12

Computers are Fast

Modern computers are so fast that most simple operations on not-too-large amounts of data appear to happen instantly.

If you're working with a data frame with 1000 rows and 10 columns, you can expect all of the following to happen so fast that a human can't percieve the delay:

- Adding or deleting one row or one column to make a new data frame.
- Replacing all the NA values in a column with zeros.
- Fitting a linear model for one column in terms of other columns.

If you see any apparent delay, it's probably not for the operation itself, but for things like fetching the data over the internet, or waiting for your computer to stop doing something else.

. . . But Not Always Fast Enough

Nevertheless, computing speed is still an issue today.

- Because today's computers are fast, people try to use them on bigger problems than before. If you work on a data frame with 1000000 rows and 1000 columns, many operations will be noticeably slow, maybe very slow.
- Sometimes you want to do simple things many, many times. For example, how well a statistical method works is often assessed by trying it on many randomly-generated data sets.
- Some tasks are inherently extremely slow for computers to do. If the famous $P \neq NP$ conjecture is true, this includes many useful tasks like finding the shortest route visiting all locations in some set (the "Travelling Salesman" problem).
- You can get computers to be very slow if you write your program in an inefficient way, when there would have been a better way.

Computing Time and Problem Size

The time it takes for a program to run depends on what computer you run it on. A low-end laptop computer might take five times longer to run a program than a high-end desktop computer.

When analysing program speed, we therefore often look not at the actual time, but at how the time grows with problem size.

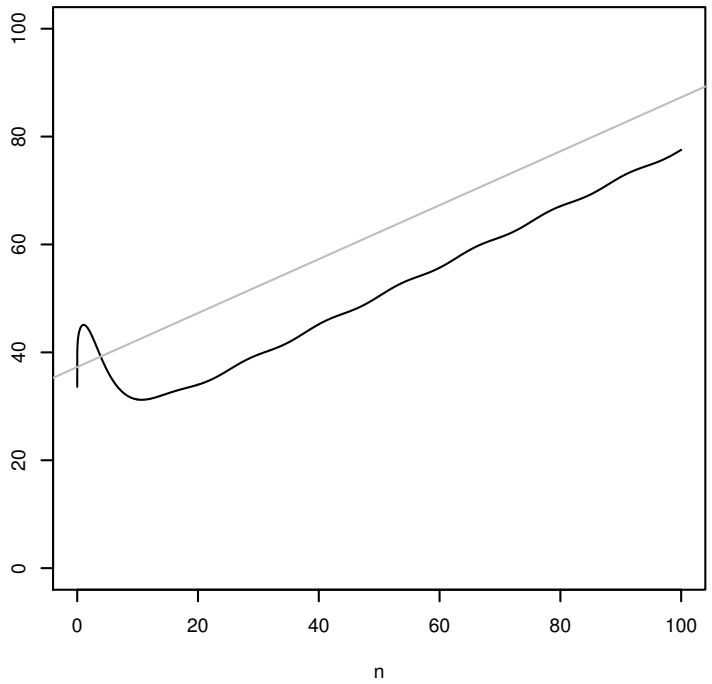
Example: Suppose we want to sort a vector of numbers in increasing order, creating another vector with the sorted list. How might the the time for this grow with the length of the vector, which we'll call n?

- For many simple methods, the time grows in proportion to n^2.
- Cleverer methods reduce this to a time growing as $n \log n$.
- If we assume that the numbers are integers that aren't huge, it can be done in time proportional to n.

What Does "Time Growing in Proportion to n" Mean

To say that the time grows in proportion to n (or to n^2, or $n \log n$), means that *asymptotically*, as n becomes larger and larger, the time will grow that way, with some unknown constant of proportionality.

Here's an example of a function that asymptotically grows in proportion to n:



The constant of proportionality seems to be 0.5 (grey line has that slope).

But if this is the time for a program to run, that constant will vary from computer to computer.

Example of Time Growing in Proportion to n

Here's an example of a simple R function that (pointlessly) counts up to n, and how its time grows with n:

```
> count <- function (n) { r <- 0; for (i in 1:n) r <- r + 1; r }
> system.time(count(100000))
   user  system elapsed
  0.017   0.000   0.017
> system.time(count(1000000))
   user  system elapsed
  0.160   0.001   0.163
> system.time(count(10000000))
   user  system elapsed
  1.583   0.013   1.596
```

The constant of proportionality seems to be about 0.00000016. But it would be smaller on a faster computer, bigger on a slower one.

Example of Time Growing in Proportion to n^2

```
> sums <- function (v) {
+   r <- numeric(length(v))
+   for (i in 1:length(v)) r[i] <- sum(v[1:i])
+   r
+ }
> system.time(sums(1:1000))
   user  system elapsed
  0.005   0.000   0.005
> system.time(sums(1:2000))
   user  system elapsed
  0.019   0.003   0.024
> system.time(sums(1:4000))
   user  system elapsed
  0.069   0.013   0.082
> system.time(sums(1:8000))
   user  system elapsed
  0.261   0.064   0.325
```

How could you write a `sums` function that is faster?

(R's built-in `cumsum` function does it the faster way.)