Algorithms
Vectorization
(JMR Ch. 9)

BTRY/STSCI 4520

On Computational Cost

- One of the key concerns of computer science – how long does it take to run this?
- For small inputs, usually not too long; important question is how does run time change with inputs?
- Relevant to data size: I can produce an answer for a data set of size 100, but how long will data of size 100,000 take?
- Usually calculated in terms of the number of operations required.

Example: The sort Problem

One of the classic problems in computer science.

- We have n numbers $x_1, \ldots, x_n$.
- We want to put them in order so that

$$x_1 \leq x_2 \leq \cdots \leq x_n$$

- How do we do this efficiently?
- How does computing time change as n increases?

Multiple ways to do this:

- Selection Sort
- Insertion Sort
- Bubble Sort
- Quick Sort

and others – how you do this makes a difference!

A First Problem – Finding the Minimum

Suppose that we just want $\min(x_1, \ldots, x_n)$.

Program 1: loop through and check if each x_i is the minimum.


```
FindMin1 = function(x){
  foundmin = FALSE  # Have we found the minimum?
  i = 0
  while(!foundmin){
    i = i+1
    ismin = TRUE                # Assume x[i] is the minimum
    for(j in 1:length(x)){      # Check against all others
      if(x[j] < x[i]){ ismin = FALSE }
    }
    # If nothing is less than x[i] it must be what we want.
    if(ismin){ foundmin=TRUE }
  }
  return(x[i])
}
```

An Analysis of Computing Time

We will count the number of comparisons made.

- At each i, we compare `x[i]` to n other entries.
- If `x[1]` is the minimum, stop; n comparisons.
- If `x[n]` is the minimum, we have made $n * n$ comparisons before we get there.
- What about somewhere in the middle?
  - If we get about half way, we consider $n/2$ entries, so make $(n^2/2)$ comparisons.
    - If we think about `x` being randomly arranged, we expect to have to look at a number of entries *proportional* to n.

A bit complicated; how do we simplify this?

Order Notation

- Suppose Algorithm 1 takes $3n^3 - 6n + 2$ operations and Algorithm 2 takes $4n^2 + 3$ operations.
- If n is large enough $3n^3 - 6n + 2 \gg 4n^2 + 3$, so Algorithm 1 will take more time for a big problem.
- In fact, if n is big, $3n^3$ and $4n^2$ will be much larger than the other terms in their expressions; we can forget the $-6n + 2$ and $+3$.
- In fact, the 3 and 4 don't matter either; $an^3 + bn^2 + cn + d$ will always be dominated by an^3 for any a, b, c and d.
- We say that $an^3 + bn^2 + cn + d$ is $O(n^3)$.
- For the minimum search above, our algorithm requires $O(n^2)$ comparisons.

Technicalities

We will not worry a great deal about the formalism about.

- "Big-O" notation:
  $f(x) = O(g(x))$ if $|f(x)| < M|g(x)|$ for all x large enough and some M.
- So that $|n(n-1)/2| < 3|n^2|$ for all n.
- We also have little-o notation (will come up later)
  $f(x) = o(g(x))$ if $|f(x)/g(x)| \to 0$ as $x \to \infty$
- For example $1/n^2 = o(1/n)$.
- Ie, Big-O means "bounded by", little-o means "much less than".

General Rules

Most expressions in terms of x^α, e^x, $\log(x)$

- If $x \to \infty$, $\alpha_1 > \alpha_2 > 0$ then for x large enough

$$e^x > x^{\alpha_1} > x^{\alpha_2} > \log(x)$$

  - If there is an exponential, take the largest one.
  - Otherwise take the largest power of x.
    - Adding expressions doesn't change order, multiplying them does.
- If $x \to 0$ (looking at small numbers) then

$$|x^{\alpha_1}| < |x^{\alpha_2}|$$

expression dominated by the smallest power of x.

Note $e^x \to 1$, $|\log(x)| \to \infty$; even larger than powers if these appear.

A More Efficient Search

$O(n^2)$ is pretty bad, can we make this better? Keep track of the minimum *up-till-now*.


```
FindMin2 = function(x){
  min = x[1] # Start at x[1]
  min.i = 1
  for(i in 2:length(x)){
    if(x[i] < min){
      min = x[i] # Update minimum if x[i] is
      min.i = i  # less that current value
    }
  }
  return(list(min=min,min.i=min.i))
}
```

Only does $n - 1 = O(n)$ comparisons.

Selection Sort

Now that we can find the minimum easily. Sort by continually finding the minimum:


```
SelectionSort = function(x){
  y = 0*x   # Store the sorted vector
  ind = 0*x # Also store the indeces of sorted elements
  n = length(x)
  for(i in 1:(n-1)){
    cur.min = FindMin2(x) # Find and record the current
    y[i] = cur.min$min    # minimum in x
    ind[i] = cur.min$min.i
    x = x[-cur.min$min.i] # Delete that entry from x.
  }
  y[n] = x                # Fix last element.
  return( list(y = y,ind = ind) )
}
```

Analyzing Selection Sort

- First iteration – it takes $n-1$ entries to find the minimum in `x`.
- Record these and remove them from `x`.
- Next iteration, `x` now length $n-1$, so we have $n-2$ comparisons to do.

- Going through all of these we make

$$\sum_{k=1}^{n}(n-k) = \frac{1}{2}n(n-1) = O(n^2)$$

comparisons.

- So if my data is 10 times as long, I have to put in 100 times the effort to sort it.

Insertion Sort

No R code this time:

- Start with x and assign y.
- Take each element of x in turn, insert it into y so y is sorted
- After k steps:

$$x = (x_1, \ldots, x_{n-k}) \quad \rightarrow \quad x = (x_2, \ldots, x_{n-k})$$
$$y = (y_1, y_2, \ldots, y_k) \qquad y = (y_1, y_2, \ldots, y_j, x_1, y_{j+1}, \ldots, y_k)$$

  so that $y_j \leq x_1 \leq y_{j+1}$.
- Might not need to compare x_1 to all the elements of y (can stop at first such that $y_j > x_1$.
- Configuration of x changes number of comparison (what's fastest?)
- Tends to be faster than Selection Sort; but still generally $O(n^2)$.

Bubble Sort

It might be useful not to need to store a new vector; instead just swap entries

Repeat

- Loop over `i` from `1` to `n`.
  - If `x[i]` > `x[i+1]`, swap them.

until `x` is sorted (and you make no more swaps).

Still $O(n^2)$ (exercise: what's the worst case?)

Tends to be slow; speed generally more an issue than memory.

Quick Sort

- Due to Hoare (1960):
  - Divide the data in two:
    1. Those less than `x[1]`; call this `a`
    2. Those greater than `x[1]`; call this `b`
  - Sort `a` and `b`, and then produce `c(a,x[1],b)`
  - But why can't we do the same thing to sort `a` and `b`?
    - Split `a` and produce `c(d,a[1],e)`
    - ...
  - Eventually we have only one element – that's easy to sort!
  - Nice Wikipedia animation.
  - But how are we going to set this scheme up?

Graphically

Divide and conquer:

Recursive Programming

It's ok to have a function call itself!

```
QuickSort = function(x){
  if(length(x) == 1 | is.null(x)){ return(x) }

  lower = c()  # Empty vectors for those less than
  upper = c()  # or greater than x[1]
  for(i in 2:length(x)){
    if(x[i] <= x[1]){ lower = c(lower,x[i]) }
    else{ upper = c(upper,x[i]) }
  }

  lower = QuickSort(lower)  # Now sort each of these
  upper = QuickSort(upper)  # and put them back together

  return( c(lower,x[1],upper) )
}
```

Graphically

Strategy goes left to right:

Analyzing QuickSort

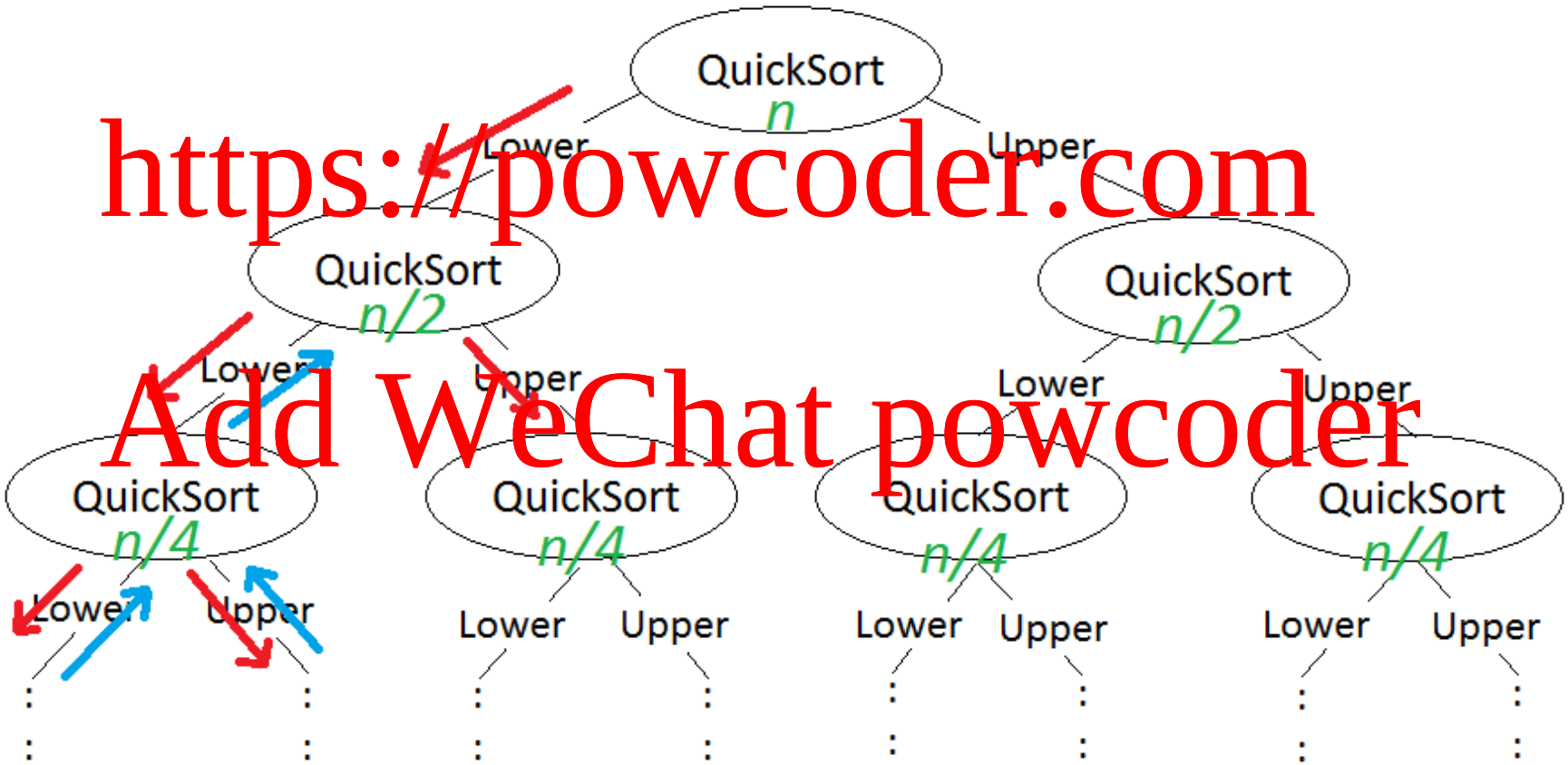
Suppose that we (luckly!) exactly partition the data set in 2 each time.

- At first level, I make $n-1$ comparisons.
- At second level, I make two lots of $(n-1)/2-1$ comparisons, for a total of $O(n)$
- At the third level, I make four lots of $((n-1)/2-1)/2-1$ comparisons, for a total $O(n)$.
- So every level has $O(n)$ comparisons, but if there are $n = 2^k$ objects, there are $k = \log(n)/\log(2)$ levels.
- That means the total cost is $O(n \log n)$ (and $n \log n = o(n^2)$; much better!).
- Worse case: `x` already sorted, then we still have $O(n^2)$.
- Start by randomly permuting `x`: expected cost is still $O(n \log n)$.

To iterate is human, to recurse divine! - L. Peter Deutsch

Graphically

At each level, divide the data by 2, but twice as many nodes; but $\log_2(n)$ levels.



Why Should Statisticians Care?

n usually = size of data set

- Operations like sorting (and many others) are integral to parts of statistical computing.
- For an $O(n^2)$ operation, something feasible at $n = 100$ is not feasible at $n = 10,000$.
- *But shouldn't Moore's law take care of that for us?* (Moore's law: computing speed doubles every 18 months)
- Equivalent rate of growth of data (recent developments: web-commerce, social networking, brain imaging, satellite images and remote sensing, high-thruput genetic screens, astronomical surveys, citizen science,...)
- Each now produces either millions of records, or hundreds of thousands of variables, or both.
- Historically: data sets grow as fast as computing speed.
- Lesson: if it isn't $O(n)$, in the long-run it will be too slow. (but note the long run can be some time away)

P and NP

- Much of the topic of *algorithms* in CS devoted to computational complexity.
- Not always easy to calculate.
- Important distinction between polynomial-time algorithms ($O(n^k)$ for some k) and exponential run time algorithms ($O(e^{an})$ for some a)
- Exponential algorithms become infeasible much faster than polynomial time algorithms.
- Problems (eg how to sort a vector) divided into two classes:

  P The set of problems that can be solved in polynomial time.

  NP The set of problems for which a solution can be *verified* in polynomial time (eg: is this vector sorted?)

Example and a Question

Traveling salesman problem:

- Salesman must visit all of a set of cities.
- We are given distances between each city.
- Is there a route through all cities for which the total distance is less than some number D?

A solution is easy to check; but finding out if there is one is hard!

Question:
Clearly anything in **P** is in **NP**, but what about the other way around?

One of the great unsolved problems of mathematics.

NP Hard

Formally, NP Hard is defined in terms of reducing NP problems to NP Hard problems (eg, you can find the minimum with a sort algorithm, but that would be dumb).

Sometimes informally used to describe problems where you can't even check a solution in polynomial time.

Statistical Example: Variable Selection

- Linear regression: $y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_k x_{ik} + \epsilon_i$
- But only some of the covariates x_i are important – which subset gives the best MSE?
- 2^k possible subsets to check – increases exponentially in k.

$$2^{30} = 1,073,741,824$$

Require *approximate* solutions, often heuristic.

Some Caveats

- Big-O only relevant when n gets very large.
- $1,000,000n > 0.1n^2$ if $n < 1,000$: constants can matter.
- If you know that n is not going to grow; you care much less about the order of computational complexity.
- You also care about readable understandable code.
  Recursion, like the divine, can be pretty ineffable.
- Not only the number of operations matter, the types of operation and the context make a difference, too.

Other Speed Considerations

- Types of operation: multiplication/division take more time than addition/subtraction take more time than comparisons take more time than assignment.
- Assigning blocks of memory is much faster than concatenating vectors together.

- If R stores memory in RAM; if it runs out, it creates virtual RAM on your hard disk – this runs much slower.

- Small amount of memory within CPU is even faster (very old school).
- Programming language also matters.

Compiled versus Interpreted Code

Most important distinction to be aware of.

- Operating systems provide the basic controls for computer hardware:
  - Task scheduling.
  - Memory allocation.
  - Basic numerical calculations (at the level of shifting bits).

  these are coded in bits.
- Compiled code (C, C++, Fortran, COBOL, ALGOL,...) is translated into a string of bit instructions that work directly with the OS, before the code is executed.
- Interpreted code (R, Matlab, Java, Perl, Python,...) is translated into OS instructions as the program runs.
- Because of overhead in translating, interpreted code is *much* slower than compiled code.

Compiled versus Interpreted Code

So why interpreted code?

- Platform independent (if you have the right interpreter): R works on Windows, Mac, Linux,...
- Saves the annoyance of compiling; dealing with compile errors.
- More easily debuggable.
- Fewer hassles (no memory allocation, ease of changing array sizes and types,...)

Many interpreted languages (including R) also allow compiled code to be used to evaluate "chunks" of instructions much faster.

Many R built-in functions are pre-compiled.

Measuring Speed in R

R has some timing functions that can be useful to evaluate efficiency.

- `system.time` reports the time it takes to evaluate some code.
- `proc.time` gives an absolute time that can be used for the same thing.


```
a = c()  # vector of no length
system.time( for(i in 1:10000){ a = c(a,log(i)) } )
```


You can put a number of lines of code inside the call to `system.time` if you put everything inside `{ }`.

proc.time

`system.time` does exactly the following

```
start = proc.time() # starting time

nsim = 25000; n = 30;  p = 0.07; mu = (1-p)/p
res = rep(0,nsim); t = rep(0,nsim)

for(i in 1:nsim){
    X = rnbinom(n,1,p)
    t[i] = sqrt(n)*abs( mean(X) - mu )/sd(X)
    res[i] = t[i] > qt(0.975,29)
}
proc.time()-start # time elapsed
```

Which I find easier to put down directly

proc.time()

```
> proc.time()
   user  system elapsed
148.16    8.75 1659.96
```


- `user` and `system` times sum to CPU effort put into evaluating the program; this is what you pay attention to.
- Difference between the two is subtle and unimportant: how much time spent on user instructions versus executing CPU functions.
- `elapsed` is clock time; can vary depending on other processes running.

R and Vectorization

R has compiled functions built in for vector/matrix operations

- sum, mean, sd, var,...
- Matrix/vector multiplication and addition.
- Element-wise multiplication and addition and other built-in functions for vectors/matrices/arrays.

These can each be written using for loops, but are much faster if used explicitly.

```
x = rnorm(100000)

start = proc.time()
m = x[1]
for(i in 2:length(x)){ m = ((i-1)/i)*m + x[i]/i }
proc.time()-start

system.time( {m2 = mean(x)} )
```

Making Use of Vectorization

Loops cannot always be avoided, but always ask "Could I do this with a vector?"

Eg: never loop through a vector if you are just doing arithmetical operations to its entries.

Compiled pointwize operations:

- `+`, `-`, `*`, `/`, `^`
- `sqrt`, `log`, `exp`, `dnorm`, ...

Vectorised operations:

- `mean`, `sum`, `var`, `sd`, `cumsum`, `diff`, ...

Matrix-vector operations:

- `t`, `%*%`, `%x%`, `diag`, `solve`

Vectorization and Linear Algebra

Linear algebra often helps: taking column means of a matrix

```
X = matrix(rnorm(1000*5000),1000,5000)
ms = rep(0,ncol(X))
for(i in 1:ncol(X)){ ms[i] = mean(X[,i]) }
```

Alternatively, the column means of X can be gotten with a matrix multiplication:

$$\begin{pmatrix} \frac{1}{n}\sum_{i=1}^{n} x_{i1} \\ \vdots \\ \frac{1}{n}\sum_{i=1}^{n} x_{ip} \end{pmatrix} = \begin{pmatrix} \frac{1}{n} \\ \vdots \\ \frac{1}{n} \end{pmatrix}^{T} X$$

In code:

```
ms2 = rep(1/nrow(X),nrow(X))%*%X
```

But remember: clarity vs efficiency trade-off!

apply Functions

apply allows you to apply a function to the rows or columns of a matrix

```
ms3 = apply(X,2,mean)
```

- X – the matrix to which we're applying something
- 2 – dimension to which it's applied; do this to every column
- mean – function to apply; must take a vector as input

Not actually any faster than a for loop; just saves typing.

tapply breaks up output by index factor giving a grouping of elements. (Output for elements with index==1, then those with index==2,...)

lapply/sapply applies to each element in a list (eg vectors of different lengths), differ in output format.

Summary

- The way a task is computed can have a big impact on run-time.
- The way run-time scales with tasks can be very important (but not always).
- Scaling measured in "big-O" notation; handy short-hand for thinking about complexity.
- Recursion: functions calling themselves – is frequently both elegant and efficient.
- But number of computations not the only determinant of run-time.
- In R, vectorization can have a dramatic impact on computational efficiency; most important thing to think about.
- Both complexity and vectorization can cost code readability – requires a balance, and good commenting.