Part A Simulation and Statistical Programming HT15

Lecturer: Geoff Nicholls

University of Oxford

Lecture 10: Recursion, Efficiency and Runtime.

Overview for lecture 10

1. Recursive evaluation
2. Extended example: Cholesky Factorization
3. Runtime analysis
4. Extended example: sorting

Recursion

Recursive programmes call themselves.

Example: Plan and write a recursive function for $f(x) = x!$.

$$f(1) = 1, \qquad f(x) = xf(x-1) \qquad \text{for } x > 1.$$

Our factorial function returns $x! = 1$ on input $x = 1$ and otherwise calls itself to evaluate $(x-1)!$ and multiplies this by x.

```
factorial<-function(x) {
   if (x==1) return(1)
   if (x>1) return(x*factorial(x-1))
   stop('x must be a positive integer')
}
```

Each function in the nested sequence of calls to `factorial()` has its own variable environment with its own distinct version of the local variable `x`.

Recursive algorithms are often shorter and clearer than the corresponding implementation via `for` or `while`. However, they may be demanding of memory, if each level of recursion makes its own copy of local variables.

Example: Cholesky Factorization

Recall simulation for the multivariate normal, $X \sim N(\mu, A)$ with $X = (X_1, X_2, ..., X_n)$, and A a $n \times n$ symmetric positive definite variance matrix.

We find a matrix L so that

$$A = LL^T.$$

If $Z = (Z_1, Z_2, ..., Z_n)$ $Z_i \sim N(0, 1), i = 1, 2, ..., n$ and we set

$$X = \mu + LZ,$$

then $X \sim N(\mu, A)$.

There are many choices for L. The Cholesky factorization is particularly neat. Because A is positive definite, there is a lower triangular matrix L satisfying $A = LL^T$.

Here is a recursive algorithm for L. Chop A and L up into blocks

$$A = \left(\begin{array}{c|c} a_{11} & A_{21}^T \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} 1 \times 1 & 1 \times (n-1) \\ \hline (n-1) \times 1 & (n-1) \times (n-1) \end{array}\right)$$

Here $A_{21} = A_{2:n,1}$ is $(n-1) \times 1$ and $A_{22} = A_{2:n,2:n}$ is itself lower triangular and $(n-1) \times (n-1)$. Similarly

$$L = \left(\begin{array}{c|c} L_{11} & 0_{1\times(n-1)} \\ \hline L_{21} & L_{22} \end{array}\right)$$

Since L is lower triangular it is zero above the diagonal, and in particular all the entries in the top row except the first are zero.

Since $A = LL^T$,

$$\begin{pmatrix} a_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0_{1\times(n-1)} \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11} & {L_{21}}^T \\ 0_{(n-1)\times 1} & L_{22}^T \end{pmatrix}$$

$$= \left(\begin{array}{c|c} L_{11}^2 & L_{11}{L_{21}}^T \\ \hline L_{11}L_{21} & L_{22}{L_{22}}^T + L_{21}{L_{21}}^T \end{array}\right)$$

so $L_{11} = \sqrt{a_{11}}$, $L_{21} = A_{21}/\sqrt{a_{11}}$ and the A_{22} block gives

$$\begin{aligned} A_{22} - L_{21}{L_{21}}^T &= L_{22}{L_{22}}^T \\ \tilde{A} &= \tilde{L}\tilde{L}^T \qquad \text{now } (n-1)\times(n-1) \end{aligned}$$

To solve for L_{22}, we need the Cholesky factorization of the $(n-1) \times (n-1)$ matrix $\tilde{A} = A_{22} - L_{21}{L_{21}}^T$, so we have reduced the problem by one dimension. Finally, if $n = 1$ so A is a scalar, $L = \sqrt{A}$ terminates the recursion.

Runtime analysis

We measure the runtime in units of operations. This might be the number of additions, subtractions, divisions and multiplications. For a sorting algorithm we can count the number of comparisons.

We typically give the asymptotic run time - as a function of the input size, for large values of the input. We give the order of the function - quadratic, cubic etc. * More efficient algorithms have (asymptotically at least) smaller run times.

We can give the worst case (for any input) or the average case (usually more interesting but harder to calculate).

*If the runtime is $g(n)$ and $g(n)$ is $O(h(n))$ then $h(n)/g(n) \to c$ as $n \to \infty$.

Here is an algorithm to find the smallest entry of $n > 1$ numbers.

```
my.min<-function(x) {
   a=x[1]
   for (k in 2:length(x)) {
      if (x[k]<a) a<-x[k]
   }
   a
}
```

Let $g(x)$ be the number of comparisons. Clearly $g(x) = n - 1$ independent of x, so the runtime is $O(n)$.

What is the runtime of `my.chol()`? Let $g(A)$ be the number of flops to factorize $n \times n$ matrix A_n.

It took $1 + (n-1) + (n-1)^2 + (n-1)^2$ additions, subtractions, multiplications and divisions (called 'flops') to solve for L_{11} and calculate the new A. The highest power is $2n^2$.

We have to repeat this for $n \to n-1 \to n-2... \to 1$. Since $\sum_{i=1}^{n} 2n^2 = 2n(n+2)(2n+1)/6$, so this implementation has approximately $g(A_n) \simeq 2n^3/3$ flops or $O(n^3)$.

If we had exploited symmetry we could get this down to about $n^3/3$ but we cant change the order (still $O(n^3)$).

```
#Cholesky
my.chol<-function(A) {
   n=dim(A)[1] #assume nxn
   if (n==1) return(sqrt(A))

   L=matrix(0,n,n)
   L[1,1]=sqrt(A[1,1])                      #count as 1 op
   L[2:n,1]=A[2:n,1]/L[1,1]                 #n-1 ops
   L[1,2:n]=rep(0,n-1)

   A22=A[2:n,2:n,drop=FALSE]
   newA=A22-L[2:n,1]%*%t(L[2:n,1]) #2(n-1)^2 here
                                   #but n(n-1) possible
   L[2:n,2:n]=my.chol(newA)

   return(L)
}
```

Example: runtime and sorting algorithm Here are a couple of R algorithms to sort a list $x = (x_1, x_2, ..., x_n)$ of n numbers.

Simple sort: find the smallest element $x_{(1)}$. Suppose it is the kth element. Remove the kth element from the list, so $y = (x_1, ..., x_{k-1}, x_{k+1}, ..., x_n)$. Return the vector $(x_{(1)}, f(y))$.

This takes $(n-1) + (n-2) + ... + 1 = O(n^2)$ comparisons independent of the order of the numbers in x.

Bubble sort: sweep through the vector, swapping x_i and x_{i+1} if $x_i > x_{i+1}$. Repeat this till the vector is in order. After i sweeps the last i elements $x_{(n-i)}, ..., x_{(n)}$ must be in their correct places so the algorithm terminates after n sweeps at most with each sweep using $n-1$ comparisons.

This takes $O(n^2)$ comparisons at worst, and $O(n)$ at best.

Merge sort let Merge sort be a function $f(x)$ that takes as input an array x of n numbers and returns a sorted array x'.

[0] If x has one entry it is sorted so return $x' = x$. Otherwise...

[1] Split x into two halves $y = (x_1, ..., x_{\lfloor n/2 \rfloor})$ and $z = (x_{\lfloor n/2 \rfloor + 1}, ..., n)$.

[2] Sort y and z using Merge sort so $y' = f(y)$ and $z' = f(z)$.

[3] Let $x' = g(y', z')$ where $g()$ is a function that takes as input two sorted vectors and merges their elements to return the sorted union of y' and z' (two sorted vectors of $n/2$ elements can be merged in $n - 1$ operations at worst).

[4] Return the sorted array x'.

The runtime of Merge sort is $O(n \log(n))$ for n-component x so it is preferred over Bubble sort. We wouldnt use an $O(n^2)$ algorithm when an $O(n \log(n))$ algorithm is available.

Proof (non-examinable): Suppose $n = 2^k$ for simplicity. Let g_k be the number of comparisons to sort this vector. Merge sort splits the vector into two vectors of length $n/2 = 2^{k-1}$. These two sub-vectors have to be sorted, which is $2g_{k-1}$ comparisons. The number of comparisons to merge the sorted sub-vectors is $2 \times (n/2) - 1 = 2^k - 1$ so

$$g_k = 2g_{k-1} + 2^k - 1$$

and $g_1 = 1$. The homogeneous solutions are $g_k = A2^k$ with particular solution $k2^k + 1$. Applying the initial condition gives $g_k = (k-1)2^k + 1$ or $g(x) = n\log_2(n) - n + 1$. We conclude that Merge sort needs $O(n\log_2(n))$ comparisons, irrespective of the input.