Practical 4 – Recursion and Runtime

Q1. Here is an R implementation of Bubble sort.

```
bubblesort<-function(x) {
      if ( (n<-length(x))<2) return(x)
      sorted<-FALSE
      while (!sorted) { #continue if last pass had a swap
            sorted<-TRUE
            for (i in 2:n) { #pass through all adjacent pairs
                  if (x[i]<x[i-1]) { #is the pair out of order
                        x[i:(i-1)]<-x[(i-1):i] #swap them
                        sorted<-FALSE #this pass had a swap
                  }
            }
      }
      return(x)
}
```

Modify the `bubblesort` function so that the number of pairs of elements that are compared and the number of pairs that are swapped are returned in a list with the sorted vector. Call this new function `bubblesort1`.

```
bubblesort1<-function(x) {
      if ( (n<-length(x))<2) return(x)
      sorted<-FALSE
      tests<-swaps<-0
      while (!sorted) { #continue if last pass had a swap
            sorted<-TRUE
            for (i in 2:n) { #pass through all adjacent pairs
                  tests<-tests+1
                  if (x[i]<x[i-1]) { #is the pair out of order
                        swaps<-swaps+1
                        x[i:(i-1)]<-x[(i-1):i] #swap them
                        sorted<-FALSE #this pass had a swap
                  }
            }
      }
      return(list(x,tests,swaps))
}
```

For each of the following 4 vectors use `bubblesort1 to find` the number of pairs of elements that are compared and the number of pairs that are swapped

(a) v1 = c(16, 12, 4, 6, 11, 19, 5, 2, 15, 1, 3, 18, 14, 8, 20, 10, 7, 13, 9, 17)
(b) v2 = 1:2000
(c) v3 = 2000:1
(d) v4 = sample(v2, 2000) [note : this samples 2000 integers without replacement from the vector v2 so you will get a different vector each time you run it.]

```
> v1 = c(16, 12,  4,  6, 11, 19,  5,  2, 15,  1,  3, 18, 14,  8,
20, 10, 7, 13,  9, 17)
> bubblesort1(v1)[2:3]
```

```
$tests
[1] 209

$swaps
[1] 87

> v2 = 1:2000
> bubblesort1(v2)[2:3]
$tests
[1] 1999

$swaps
[1] 0

> v3 = 2000:1
> bubblesort1(v3)[2:3]
$tests
[1] 3998000

$swaps
[1] 1999000

> v4 = sample(v2, 2000)
> bubblesort1(v4)[2:3]
$tests
[1] 3830084

$swaps
[1] 1002153
```

How many pairs of elements are compared in the worst case, as a function of the input length n? (Ans `n*(n-1)` or in other words `O(n^2)`)

Q2. Suppose A,B are n x n matrices and x is an n x 1 vector, and we need the matrix product ABx. How many multiplications are n A(Bx)? (Answer, 2n^2). How many in (AB)x? (Answer, n^2+n^3). Which of these does R use to evaluate `A%*%B%*%x`? (Answer, the slow one (AB)x).

Q3. Pascal's triangle is a geometric arrangement of the binomial coefficients in a triangle.

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
….

Entries in each row are determined by summing adjacent numbers in the previous row. The start and end of each row is always 1. Write an R function to calculate the nth row of Pascal's triangle using the idea of recursion.

```
pascal = function(n) {
  if (n==1) return(1)
  y = pascal(n-1)       #if this is the n-1st row
  return(c(0,y)+c(y,0)) #then this is the nth row
}
```

Q4. Here is an algorithm converting a non-negative integer x to binary.

> [0] If x is one or zero then return x. Otherwise, proceed as follows.
>
> [1] Take the remainder B0 when x0 = x is divided by 2. This is the first digit (coefficient of 2^0). (Hint – what do `%%` and `%/%` do?)
>
> [2] Now set x1 = (x0 − B0)/2. Repeat this collecting B1,B2 etc.
>
> [3] the algorithm stops when we have divided x down to xn = 1. Set Bn = 1 and return the binary number with digits BnBn−1...B1B0

Write a recursive R function implementing this algorithm. Your function should take as input a non-negative integer `x` and return the corresponding binary number. Represent the binary number as a vector, so for example 10 is `c(1,0,1,0)`.

```
dec2bin<-function(x) {
#convert decimal integer x>=0 to binary
if (round(x)!=x || x<0) stop('x should be an integer >=0')
if (x<2) return(x)
return(c(dec2bin(x%/%2),x%%2))
}
```

Q5 Implement the following sorting algorithm.

Insertion sort: sort `(x_1,...,x_{n-1})` then take `x_n` and insert it in correct position in the sorted vector. Sort `(x_1,...,x_{n-1})` using Insertion sort!

```
g<-function(x) {
      #insertion sort
      if (length(x)<2) return(x)
      p<-x[1]; x<-g(x[-1])
      if (p<=x[1]) return(c(p,x))
      if (p>=x[n<-length(x)]) return(c(x,p))
      i<-1; while (x[i]<p) i<-i+1
      return(c(x[1:(i-1)],p,x[i:n]))
}
```

Show that the worst case number of comparisons in insertion sort is `O(n^2)`.

Worst case for insertion-sort is a monotone decreasing list x=(n,n-1,...,1). Each time it takes the first entry off and compares it to all the others for (n-1)+(n-2)+...+1 comparisons (actually twice that as I have coded it). That is O(n^2).

Q6

(i) Write an R function which simulates birthdays for n people. Assume 365 days in a year, represent dates as integers from 1 to 365, and assume birth-dates are uniformly distributed over the year.

Your function should take as input the number n of people and return a vector of length n giving the n dates.

```
birthdays<-function(n=23) {
   #return n simulated birthdays as a vector
   ceiling(365*runif(n))
}
```

(ii) Write an R function which tests to see if any date is repeated r times or more in a vector of n birthdays. How does the runtime of your function depend on n?

```
is.repeated<-function(b,r=2) {
      #return true if an element of b is repeated
      #r times or more
      u<-rep(0,365)
      #add one to each day as a birthday falls on that day
      for (k in 1:length(b)) u[b[k]]<-u[b[k]]+1
      #do any day-tallies exceed r-1
      return((any(u>(r-1))))
}
```

```
This takes n additions and 365 tests – we go through the n dates
exactly once. The test for repeated dates goes through the 365
days of the year. So the above has runtime O(n).
```

(iii) Write an R function which estimates the probability that two or more people share a birthday in a group of `n` people, using `m` simulated sets of `n` birthdays. Your function should take as input the two integers `n` and `m` and return an estimate for the probability that two or more people share a birthday.

```
estimate<-function(n,m,r=2) {
   #probability a birthday is repeated r times or more
   #in a group of n individuals based on m simulated groups
   x<-rep(0,m)
   for (t in 1:m) {
      b<-birthdays(n)
      x[t]<-is.repeated(b,r)
   }
   mean(x)
}
```