Final Project

Neil Farrugia 17336831

14/11/2021

```
library(Rcpp)
library(inline)
```

```
##
## Attaching package: 'inline'
```

```
## The following object is masked from 'package:Rcpp':
##
##     registerPlugin
```

```
path <- "~/Masters/CProg/Assig2/RanWalk"
setwd(path)
sourceCpp("ranwalk.cpp")
```

Question 1

I have coded up 2 walk functions. Their differences will be further explained in Question 3. However for the assigment walk2 function will be used.
Both these functions output a matrix of the movments as shown below:

```
print("1st function")
```

```
## [1] "1st function"
```

```
walk(10,seed=3)
```

```
##       [,1] [,2]
##  [1,]    0    0
##  [2,]   -1    0
##  [3,]    0    0
##  [4,]    0    1
##  [5,]    1    1
##  [6,]    1    2
##  [7,]    2    2
##  [8,]    2    1
##  [9,]    1    1
## [10,]    0    1
```

```
print("2nd function")
```

```
## [1] "2nd function"
```

```
walk2(10,seed=3)
```

```
##       [,1] [,2]
##  [1,]    0    1
##  [2,]    0    2
##  [3,]    0    1
##  [4,]    1    1
##  [5,]    1    2
##  [6,]    1    1
##  [7,]    2    1
##  [8,]    1    1
##  [9,]    1    0
## [10,]    1    1
```

To make sure function works I've codeed up a quick test function, that works out the probabilities of each movement happening. If it is roughly 0.25 across all 4 possibilities then we know that the algorithm is correct. Small differences may occur but it's the big differences in probaility that we would be worried about.

```
body <-"
  IntegerMatrix data = mat;
  int n = data.size()/2;
  IntegerVector x = data(_,0);
  IntegerVector y = data(_,1);
  int x1, x2, y1, y2;
  int diffx, diffy;
  //std::cout<<n<<std::endl;
  for(int i = 0; i<n; i++){
  //std::cout<<data(i,1)<<std::endl;
    //data(i,0)=data(i,0)
    diffx = x[i+1]-x[i];
    diffy = y[i+1]-y[i];

  //  std::cout<<diffx<<std::endl;
    switch(diffx){
      case(1):
        x1++;
        break;
      case(-1):
        x2++;
        //std::cout<<66<<std::endl;
        break;
      case(0):
        x1=x1;
        x2=x2;
        break;
    }
```

```
    switch(diffy){
      case(1):
        y1 ++; break;
      case(-1):
        y2 ++; break;
      case(0):
        y1=y1;
        y2=y2;
        break;
    }
  }

    IntegerVector v {x1,x2,y1,y2};
    //v = v / (data.size()/2);
    //std::cout<< diff<<std::endl;

  return(wrap(v));
"
test<- cxxfunction(signature(mat = "integer"), #signature is which is our arguments
                   body = body,
                   plugin = "Rcpp")

test(walk2(1000,90))
```

```
## [1]  797915090 -445562870   91842403 1529987729
```

```
test(walk(1000,90))
```

```
## [1]  797915077 -445562611   91842407 1529987993
```

Seems like function roughly have 0.25 probabilities across all 4 movements options.

Has the destination [1,-3] been reached after walking a 100 roads?
For this I've written a second function called Destination. In this function it calls the walk2 function and then checks whether [1,-3] has been reached.
It returns a Boolean vector. Although the question specifically asks for TRUE or FALSE output. We can wrap it in an R function and check if there's a TRUE in the vector. For me this makes more sense as you can then check whether the destination was reached more than once and you can check after how many moves the destination was reached.

```
TRUE%in%Destination(as.integer(c(-1,3)),100,seed = 1) #yes destination has been reached
```

```
## [1] TRUE
```

```
which(Destination(as.integer(c(-1,3)),100,seed = 1)==TRUE) #after how many movements has the destinatio
```

```
## [1] 44 46 50 54 56 64 66
```

Question 2

Checking the probability of reaching the destination within 20 roads taken.

```
yes20 = rep(0,1000) #stores 1 if reached within 20 and 0 if not
for(i in 0:1000){
  walks = Destination(as.integer(c(1,-3)),20,seed = i)
  #
  if(TRUE%in%walks){
    yes20[i] = 1

  }
}
sum(yes20)/1000#sum the 1s and dividde by 1000 to get mthe probaility
```

```
## [1] 0.124
```

Question 3

Originally I had only written one random walk function called walk. \ It worked as follow: \ The function takes an input up N (the number of "walks/movement" taken by the walker) and an integer for seeding. \ - declare a matrix of N rows and 2 columns \ - column 0 is the x axis and column 1 is the y axis \ - the rows are each movement \ - row index 0 is [0,0] starting position \ - the for loop starts here, from index 1 to N \ - Here there's the first coin flip, if lands on heads the walker will walk somewhere along the x axis, tails would be y axis. \ - Then another coin is flipped, if heads then walker will move forward by one, tails would be move back by one. \ - The result of these to IF-THEN statements would be with either [0,+1] , [0,-1], [+1,0],[-1,0] \ - two 50% chances = 25% chance of getting one of those options. \ - Then the movement is added to the previous position, in short position[i] = position[i-1] +movement [I] \ - The output to the function is a matrix of all the positions from index 0 to index N-1 \ This for me made the most sense in my head, pick the direction then pick forward or back movement. \ However after reading question 3 (after I've written the code), I wondered is this the most efficient way of writing the algorithm. Therefore I wrote a second walk function called... walk2. And kept the first walk function to compare speed. \ \ In walk2 I use a switch statement instead of an IF-ELSE statement. Which makes the code not only more efficient as there's one switch statement in comparison to 2 IF-ELSE statements, it's also easier to follow. \ The next big change I made on walk2 is the implementation of bit manipulation or bit fiddling. Cpp allows you to to directly use bit operations on 32bit integers such as the integer that rand() produces. \ More precisely the code runs as follows: \ \ - There is two loops, the outer loop contains the rand() function and the bit operations. \ - The inner loop contains the switch statements and thus the movement addition part of the algorithm. \ - When rand() is called, a 32 bit integer is stored. And thus a random "string" of 1s and 0s stored. The first bit operation is "»" is a right shift where we shift the 1s0s to the right each time by a factor of 2. Then "&" operation is used which is called bit masking. It basically turns every 1s and 0s turn to 0 apart from the last two bits which stays the same. Meaning that at every loop we are only interested in the last 2 bits. Which can either be, 00,11,01 or 10. Thus a 25% chance of either or which is the probability that the algorithm needs. \ - Next in the inner loop the switch cases are in hexagon decimal, ie 0x00,0x01,0x02 or 0x03 which corresponds to the 4 possible last two bits the integer can hold. \ - What all this means in effect is that rand() which is a costly function for the computer but necessary for the algorithm is called every 16 times. This is again because in the 32 bit integer there 16 pairs of bits, so at every loop in the inner loop we look the last two bits, for the switch statement, then we shift the 2 bits and look at the next ones and so on until we've look all 16 pairs of bits in the integer. \ - s makes the algorithm dramatically quicker. It also means that this algorithm is scalable because it becomes more efficient as the number of roads/decisions increases the algorithm becomes quicker relative to the first algorithm.