```
#for working with polygons
library(sf)
library(sfheaders)

library(tidyverse)
library(gtools)

set.seed(22081992)
```

Riddler Express

This weeks express deals with an erratic driver:

In Riddler City, the city streets follow a grid layout, running north-south and east-west. You're driving north when you decide to play a little game. Every time you reach an intersection, you randomly turn left or right, each with a 50 percent chance.

After driving through 10 intersections, what is the probability that you are still driving north?

So all we have to do is create a binomial tree of depth 10 and then sum by final heading direction. As the driver *must* turn left or right at each junction, we actually only have to consider the final turn as this will change it from whichever North/South or East/West it is heading to the other with p = 0.5. But if we want to prove this, let's consider it as a more canonical ball-drawing probability task where one can draw balls from a bag:

- Red (right) ball with probability p or
- Lime (left) ball with probability q

drawing balls 10 times without replacement

We know that as there are only two balls, the total probability is

\[(p + q) = 1 \]
on the first pick we are just choosing p or q so can raise everything to the power 1 (pick) to get the same formula:

\[(p + q)^1 = 1^1 \]
and can generalise to n picks

\[(p + q)^n = 1^n \]
to expand this we're going to get combinations of p and q to the powers from 0:n, multiplied by the combinatorics from Pascal's triangle.

If we set this multiplication as m, we can express this as:

\[m = \frac{n!}{(n-k!)k!} \]
(where k is 0:n)

so for n = 10 (turns of the car, or picks of a ball), we get

```
#calculate pascals triangle via factorials
calc_pascal <- function(n,k) {
  factorial(n) / (factorial(n-k) * factorial(k))
}

#run for n turns
n_turns <- 10
m = map2_dbl(n_turns, 0:n_turns, calc_pascal)
m
```

```
##  [1]   1  10  45 120 210 252 210 120  45  10   1
```

so for

$$ (p + q)^{10}$$

we will expand this into

$$ 1p^{10} + 10p^9q + 45p^8q^2 + 120 p^7q^3 + 210p^6q^4 + 252p^5q^5 + 210p^4q^6 + 120p^3q^7 + 45p^2q^8 + 10pq^9 + 1q^{10}$$

But where we now diverge from the balls in a bag, each time we draw (/turn), the position of our car updates. We don’t care about the probability of each of these per se, but the probabilities grouped by the final direction of the car.

It should be clear that every p draw (a right turn), moves the car 1 cardinal direction to the right, whereas a left turn moves it -1 cardinal direction. In our expansion we have 210 examples of drawing 6 right turns and 4 left turns, which would end up having the car face due south (2 cardinal turns). For each term, we just have to minus the exponent of the left turns from the exponent of the right turns, then find the direction by taking the 4th modulus of this.

For a binomial expansion like this, it’s very easy:

```
#calculate the end heading for each term of the expansion
term_direction = (n_turns:0 - 0:n_turns) %% 4
term_direction
```

```
##  [1] 2 0 2 0 2 0 2 0 2 0 2
```

so we’re either going to end up facing north (0 overall turn) or south (2 overall turns). We can then multiply these by the m for each term

```
#list of cardinal direction
final_directions <- c("north", "east", "south", "west")

#loop through each expansion term to get the final direction
direction_p <- c()
for(d in 0:3) {
  direction_p[d+1] <- sum(m[term_direction == d])
}

#find the probability of facing any direction
names(direction_p) <- final_directions
direction_p / sum(direction_p)
```

```
## north  east south  west
##   0.5   0.0   0.5   0.0
```

so we have a 50% chance of ending up facing either north or south. So the answer to this weeks riddler express is

$$p(North) = 0.5 $$

Extra Credit

For extra credit, the driver decides instead to turn left, right, or continue straight with equal probability (1/3). In addition to p and q, we now also have an r probability where

$$ r = p(No Turn) $$

We can then use expand.grid() to produce combinations of these three probabilities, and count the combinations by number of each of these:

```
#find combinations of p, q, and r
extra_credit <- expand.grid(rep(list(c("p", "q", "r")), n_turns)) %>%
  #label each combination
  mutate(id = 1:n()) %>%
  #count numbers of p, q, and r
  pivot_longer(cols = starts_with("Var")) %>%
  group_by(id, value) %>%
  summarise(n = n()) %>%
  #pivot back to wide
  pivot_wider(id_cols = id, names_from = value, values_from = n) %>%
  mutate_at(c("p", "q", "r"),  ~replace(., is.na(.), 0)) %>%
  #count numbers of each combination
  group_by(p, q, r) %>%
  summarise(n = n()) %>%
  arrange(n)

extra_credit
```

```
## # A tibble: 66 x 4
## # Groups:   p, q [66]
##         p     q     r     n
##
##  1      0     0    10     1
##  2      0    10     0     1
##  3     10     0     0     1
##  4      0     1     9    10
##  5      0     9     1    10
##  6      1     0     9    10
##  7      1     9     0    10
##  8      9     0     1    10
##  9      9     1     0    10
## 10      0     2     8    45
## # ... with 56 more rows
```

As we might expect, we get the same number of each combinations, but with 3x combinations for each x^n y^n (for each combination of p, q, and r). As we know that the final heading will be the difference in number of right and left turns, we can subtract these and count the number of combinations leading to each direction

```
extra_credit_answer <- extra_credit %>%
  mutate(net_turns = p - q) %>%
  mutate(final_direction = net_turns %% 4) %>%
  .$final_direction %>%
  table()

names(extra_credit_answer) <- final_directions
extra_credit_answer / sum(extra_credit_answer)
```

```
##     north      east     south      west
## 0.2727273 0.2272727 0.2727273 0.2272727
```

giving us an answer of

\[p(North) = 0.\dot{2}\dot{7} \]
# Riddler Classic

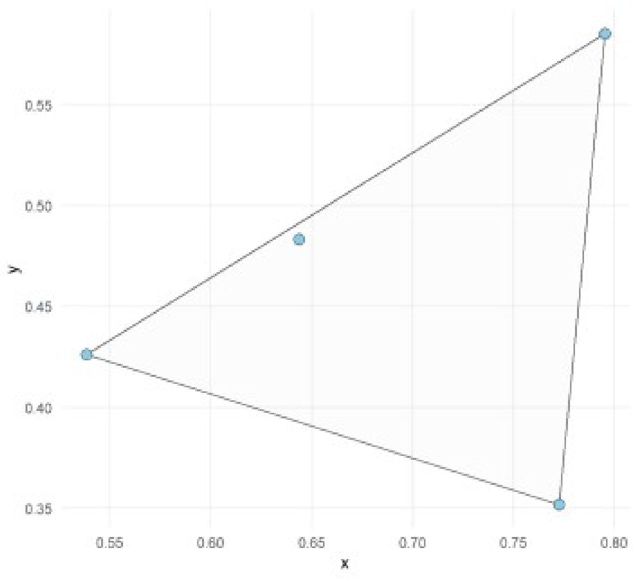
Polly Gawn loves to play “connect the dots.” Today, she’s playing a particularly challenging version of the game, which has six unlabeled dots on the page. She would like to connect them so that they form the vertices of a hexagon. To her surprise, she finds that there are many different hexagons she can draw, each with the same six vertices.

What is the greatest possible number of unique hexagons Polly can draw using six points?

This is a pretty tricky question! I can't see any way to analytically solve it and given that it involves polygons (and not just pure numbers) it seems like a tricky question to brute force. That doesn't mean we can't try though.

Let's start by using the data in the hint- that for n = 4 points, the maximum number is 3 polygons, given that the fourth point lies within an enclosing triangle of the other three. We can generate some points randomly for this pretty easily, and use the simple features package to test the properties of the resulting polygons:

```
#generate 3 random points
points <- data.frame(
  x = runif(3),
  y = runif(3)
)

#create a triangle from these points
triangle <- sf_polygon(points)

#randomly generate a fourth point within the bounding box of these points
new_point <- data.frame(
  x = runif(1, min = min(points$x), max = max(points$x)),
  y = runif(1, min = min(points$y), max = max(points$y))
)

#keep generate this point until it lies within the triangle of the previous 3
while(length(unlist(st_contains(triangle, sf_point(new_point)))) ==0) {
  new_point <- data.frame(
    x = runif(1, min = min(points$x), max = max(points$x)),
    y = runif(1, min = min(points$y), max = max(points$y))
  )
}

#bind the fourth point onto the previous 3
points <- rbind(points, new_point)

#plot the points
p2 <- ggplot() +
  #triangle
  geom_sf(data = triangle, alpha = 0.1) +
  geom_point(data = points, aes(x, y),
             shape = 21, fill = "skyblue", colour = "black", size = 3) +
  theme_minimal()

p2
```

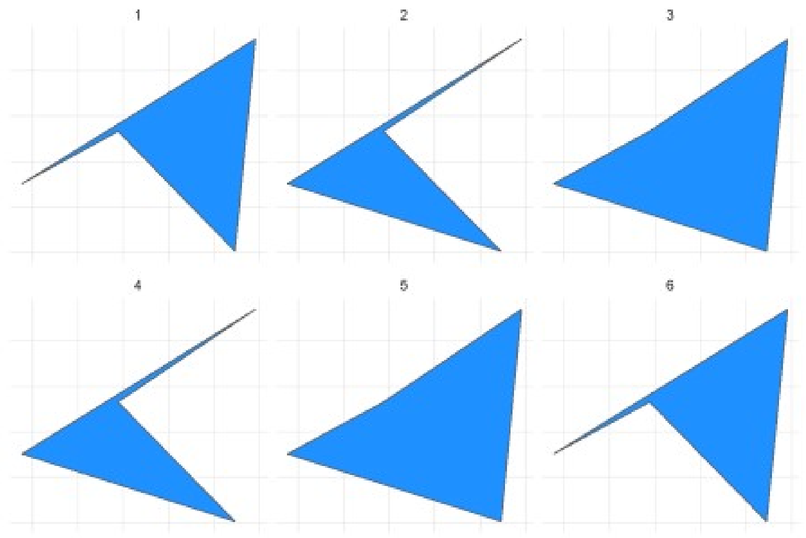


Now we need to brute force through every possible polygon. To do this we can use combinatorics again, this time with the permutations() function from the gtools package. We create every possible route of points, then take only the routes that start on the first point (to cut down our search space, as many routes will be the same just shifted to a different start node)

```
#create all possible routes of 4 points
routes <- permutations(4, 4, 1:4) %>%
  as.data.frame() %>%
  #filter those beginning with node 1
  filter(V1 == 1)
```

For each route we then create the resulting polygon by ordering the points and creating a simple features polygon. These are then bound together and each given an id.

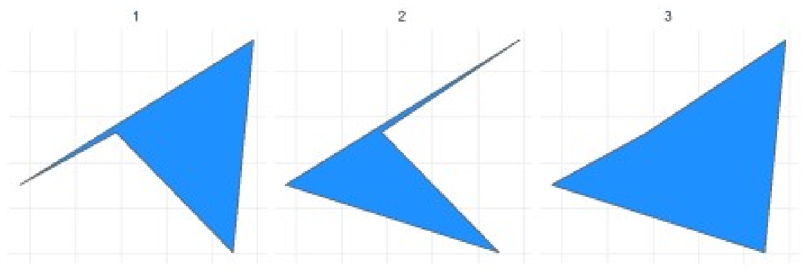
```
#cycle through routes to create polygons
for(r in seq(nrow(routes))) {
  nodes <- as.numeric(routes[r,])
  sf_points <- points[nodes,]
  sf <- sf_polygon(sf_points)

  if(r == 1) {
    polygons <- sf
  } else {
    polygons <- rbind(polygons, sf)
  }
}

polygons$id <- 1:nrow(polygons)

#plot the resulting polygons
p3 <- ggplot() +
  geom_sf(data = polygons, fill = "dodgerblue") +
  theme_minimal() +
  theme(axis.text = element_blank()) +
  facet_wrap(~id)

p3
```



However, we know that there are only 3 unique polygons for n = 4 points. Why have we found 6? From inspection it's pretty clear that even though they all have unique paths, 3 of these are duplicates of 3 others. This occurs as for each starting node, there are two paths to create each polygon, a 'clockwise' path and an 'anticlockwise' one.

We can easily test for this and remove half the polygons as such:

```
#test for duplicate polygons
duplicates <- as.data.frame(st_equals(polygons, polygons)) %>%
  #ignore self matches
  filter(row.id != col.id) %>%
  mutate(id = 1:n()) %>%
  #remove the last 3 polygons
  top_frac(0.5, id)

polygons <- polygons[-duplicates$row.id,]

#replot
p4 <- ggplot() +
  geom_sf(data = polygons, fill = "dodgerblue") +
  theme_minimal() +
  theme(axis.text = element_blank()) +
  facet_wrap(~id)

p4
```

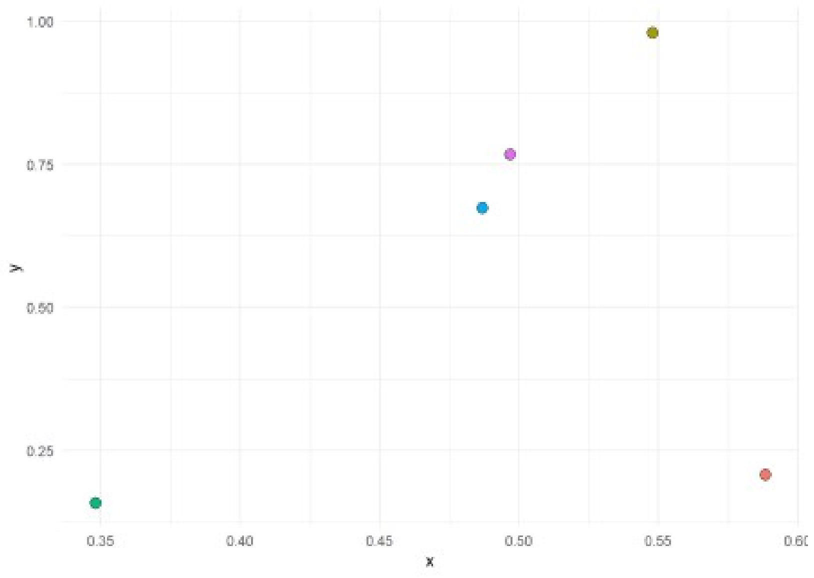


And we have our 3 unique polygons. For a higher number n, we want to spin out and generalise two functions:

- one to create points on a ‘page’
- one to build as many unique polygons as possible

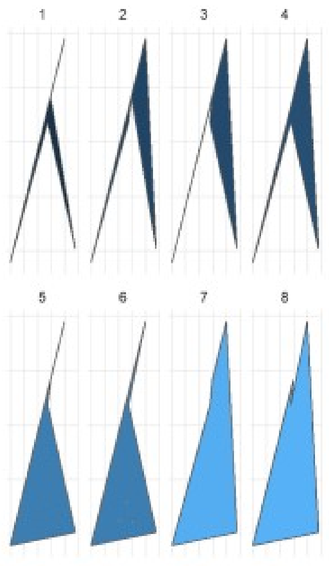
To create points, we can pretty much verbatim take the previous code. I’ve added a second argument of how many points should lie within a perimeter triangle of points, though this will always be n-3 (where n > 3), as far as I can see.

```
#take our previous code for any n
create_points <- function(sides, within) {
  points <- data.frame(
    x = runif(sides - within),
    y = runif(sides - within)
  )
  perimeter <- sf_polygon(points)

  new_points <- data.frame(
    x = runif(within, min = min(points$x), max = max(points$x)),
    y = runif(within, min = min(points$y), max = max(points$y))
  )

  while(length(unlist(st_contains(perimeter, sf_point(new_points)))) != within)
{
    new_points <- data.frame(
      x = runif(within, min = min(points$x), max = max(points$x)),
      y = runif(within, min = min(points$y), max = max(points$y))
    )
  }

  points <- rbind(points, new_points) %>%
    mutate(id = 1:n())
  return(points)
}

#run to create a pentagon
five_points <- create_points(5, 2)

#plot the five points
p5 <- ggplot() +
  geom_point(data = five_points, aes(x, y, fill = as.factor(id)),
             shape = 21, colour = "black", size = 3) +
  scale_fill_discrete(guide = FALSE) +
  theme_minimal()

p5
```



The second function (to test how many polygons can be drawn) needs two minor tweaks. Polygons are created as before, but to test for duplicates, we now take the smaller id each time, and also use st_is_valid() to check that the polygon does not contain any self-intersections (where lines cross each other).

```
#create polygons for n points
get_unique_polygons <- function(points) {
  #create polygons as before
  sides <- nrow(points)
  routes <- permutations(sides, sides, 1:sides) %>%
    as.data.frame() %>%
    filter(V1 == 1)

  for(r in seq(nrow(routes))) {
    nodes <- as.numeric(routes[r,])
    sf_points <- points[nodes,]
    sf <- sf_polygon(sf_points)

    if(r == 1) {
      polygons <- sf
    } else {
      polygons <- rbind(polygons, sf)
    }
  }

  polygons$id <- 1:nrow(polygons)

  #find duplicate polygons
  duplicates <- as.data.frame(st_equals(polygons, polygons)) %>%
    filter(row.id != col.id) %>%
    mutate(smaller = case_when(
      row.id < col.id ~ row.id,
      col.id < row.id ~ col.id
    ))
  #always take the smaller id
  polygons <- polygons[polygons$id %in% duplicates$smaller,]
  #test for valid polygons
  #i.e. no self-intersections
  polygons <- polygons[st_is_valid(polygons),]

  return(polygons)
}

#create pentagons
pentagons <- get_unique_polygons(five_points)
#calculate and arrange by the area of each for aesthetics
pentagons$area <- st_area(pentagons)
pentagons <- pentagons %>%
    arrange(area) %>%
    mutate(id = 1:n())

#plot the unique pentagons
p6 <- ggplot() +
  geom_sf(data = pentagons, aes(fill = area)) +
  scale_fill_continuous(guide = FALSE) +
  theme_minimal() +
  theme(axis.text = element_blank()) +
  facet_wrap(~id, nrow = 2)

p6
```



So for 5 points, the answer seems to be 8 unique polygons that can be drawn.

For higher n, I then ran these function repeatedly and found the largest number of polygons for any random allocation of points, I've used 6 here, but the number can be any. At n = 6 points it's already struggling (my code here wasn't written for efficiency) and at 7 is reaaallly slow, so the loops can be arbitrarily large and run while you make dinner/watch TV etc.

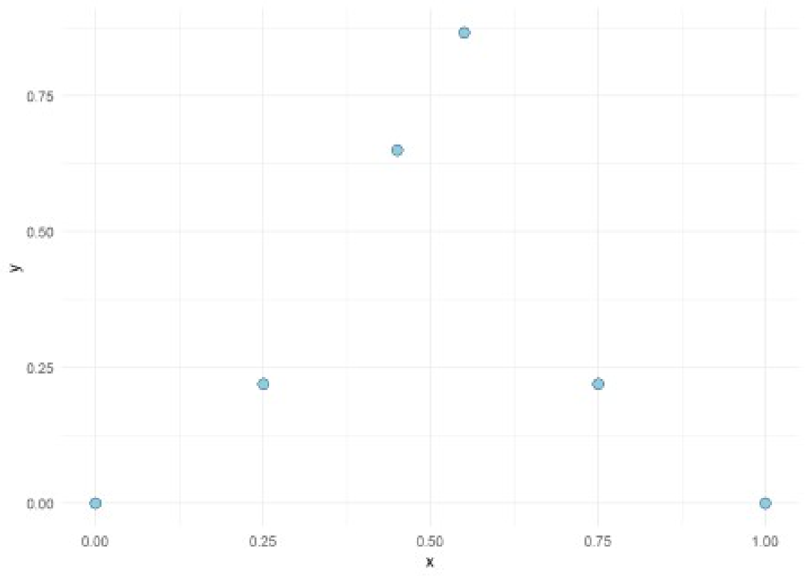
```
#very dirty inefficient brute force code
all_n <- c()
n_points <- 6
for(i in 1:1){
  #randomly create points
  points <- create_points(n_points, n_points-3)
  #build polygons from these
  polygon <- get_unique_polygons(points)
  n <- nrow(polygon)
  all_n[i] <- n
  #report back from the loop
  print(paste(i, "loops run"))
  print(paste("biggest n so far is", max(all_n)))
  print(all_n)
}
```

```
## [1] "1 loops run"
## [1] "biggest n so far is 24"
## [1] 24
```

While I was running this to check if I'd missed anything, I tried to solve the problem logically (but not analytically). It seemed clear that you want as many points within larger perimeter triangle of 3 points. It also seemed like you wanted to make sure that none of these points were on a straight line of 3 points (which would limit the number of possible connections of those 3 points). For n = 6 I settled on a slightly offset (by rounding errors) triangle-within-a-triangle

```
#logic-created six points
six_points <- data.frame(
  x = c(0, 1, 0.55, 0.25, 0.75, 0.45),
  y = c(0, 0, sqrt(0.75), 0.22, 0.22, 0.65)
)
```

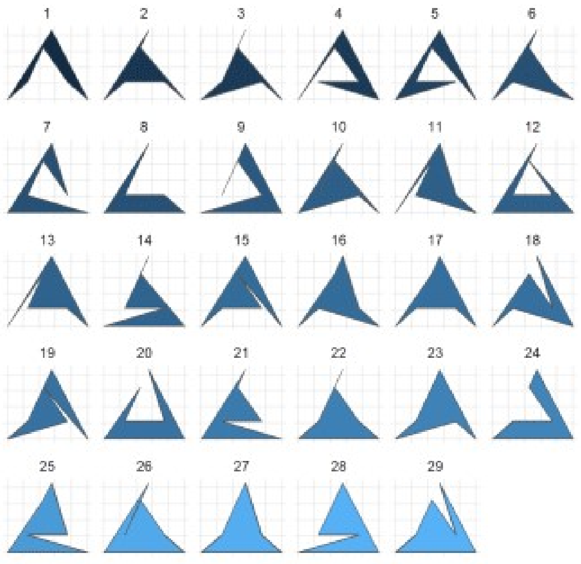
```
#plot the six points
p7 <- ggplot() +
  geom_point(data = six_points, aes(x, y),
             shape = 21, fill = "skyblue", colour = "black", size = 3) +
  theme_minimal()

p7
```



if we pass these points through our function we find that it can create 29 unique polygons (the same number I found from ~100 loops of my brute force attack). Plotting them as before, these are:

```
#test the six points and munge
heaxgons <- get_unique_polygons(six_points)
heaxgons$area <- st_area(heaxgons)
heaxgons <- heaxgons %>%
    arrange(area) %>%
    mutate(id = 1:n())
```

```
#plot
p8 <- ggplot() +
  geom_sf(data = heaxgons, aes(fill = area)) +
  scale_fill_continuous(guide = FALSE) +
  theme_minimal() +
  theme(axis.text = element_blank()) +
  facet_wrap(~id)

p8
```



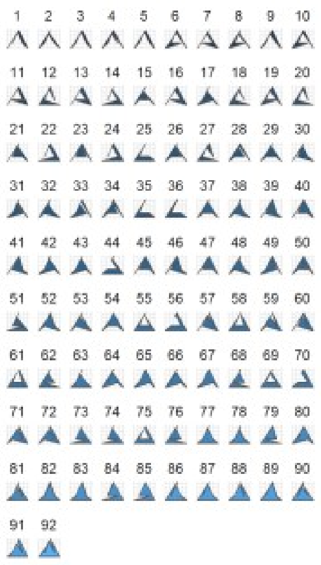
This isn't a proof, but I feel reasonably confident in this as the answer for the classic

Extra Credit

As mentioned, now we want to find this for 7 points creating heptagons. Given we can now fit 4 spare points inside our original triangle, I decided to see what would happen if you stretched the triangle-within-a-triangle and point the final point inside this.

```
#logic-created seven points
#stretched y axis on point six
#point seven lies within new triangle
seven_points <- data.frame(
  x = c(0, 1, 0.55, 0.25, 0.75, 0.5, 0.45),
  y = c(0, 0, sqrt(0.75), 0.22, 0.22, 0.75, 0.65)
)

#munge our heptagons
heptagons <- get_unique_polygons(seven_points)
heptagons$area <- st_area(heptagons)
heptagons <- heptagons %>%
    arrange(area) %>%
    mutate(id = 1:n())

#aaaaand plot
p9 <- ggplot() +
  geom_sf(data = heptagons, aes(fill = area)) +
  scale_fill_continuous(guide = FALSE) +
  theme_minimal() +
  theme(axis.text = element_blank()) +
  facet_wrap(~id)
```

p9


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92