Express

From Zack Beamer comes a baffling brain teaser of basketball, just in time for the NBA playoffs:

Once a week, folks from Blacksburg, Greensboro, and Silver Spring get together for a game of pickup basketball. Every week, anywhere from one to five individuals will show up from each town, with each outcome equally likely.

Using all the players that show up, they want to create exactly two teams of equal size. Being a prideful bunch, everyone wears a jersey that matches the color mentioned in the name of their city. However, since it might create confusion to have one jersey playing for both sides, they agree that the residents of two towns will combine forces to play against the third town's residents.

What is the probability that, on any given week, it's possible to form two equal teams with everyone playing, where two towns are pitted against the third?

Extra credit: Suppose that, instead of anywhere from one to five individuals per town, anywhere from one to N individuals show up per town. Now what's the probability that there will be two equal teams?

This is a nice little combinatorics problem, as such we can solve it by finding all combinations and then the combinations where the maximum value is equal to the sum of the remaining values:

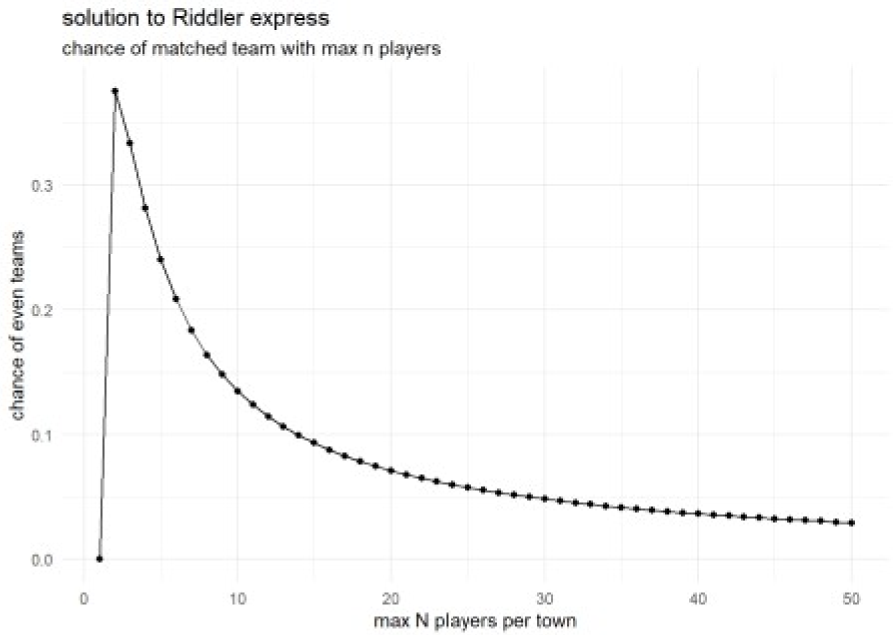
```
#create lists of possible values for all team a, b, or c
players <- list(a = 1:5, b = 1:5, c = 1:5)
#find all combinations
player_combinations <- do.call(expand.grid, players)

#get the value of the largest team in each combination
largest_team <- apply(player_combinations, 1, max)
#get the sum of the remaining teams in each combination
reamining_players <- apply(player_combinations, 1, function(x) sum(x) -
max(x))

#check when these match
matched_teams <- nrow(player_combinations[which(largest_team ==
reamining_players),])
#find the fraction which match
fraction_even_teams <- matched_teams / nrow(player_combinations)

fraction_even_teams
## [1] 0.24
```

So the answer to the main express question is 0.24, or about 1 in 4 chance.

It's easy to expand this to multiple players by allowing the first line to take any value:

```
#rewrite previous chunk as function that takes max_players as an
argument
find_matches_fraction <- function(max_players) {
```

```
  players <- list(a = seq(max_players), b = seq(max_players), c =
seq(max_players))
  player_combinations <- do.call(expand.grid, players)

  largest_team <- apply(player_combinations, 1, max)
  reamining_players <- apply(player_combinations, 1, function(x) sum(x)
- max(x))

  matched_teams <- nrow(player_combinations[which(largest_team ==
reamining_players),])
  fraction_even_teams <- matched_teams / nrow(player_combinations)
}

#run for n 1:50
fraction_even_teams <- lapply(seq(50), find_matches_fraction)

answers_df <- data.frame(
  townspeople = seq(50),
  chance = unlist(fraction_even_teams)
)

#for plotting
library(ggplot2)

#plot the answers for 1 to n players where max n is 50
p1 <- ggplot(answers_df, aes(x = townspeople, y = chance)) +
  geom_point() +
  geom_line() +
  labs(
    title = "solution to Riddler express",
    subtitle = "chance of matched team with max n players",
    x = "max N players per town",
    y = "chance of even teams"
  ) +
  theme_minimal()

p1
```



Let’s implement this in python. I won’t comment lines again, the flow of the function is fundamentally the same

```
import itertools

def find_matches_fraction(max_players):
  team_a = range(1,max_players)
  team_b = range(1,max_players)
  team_c = range(1,max_players)

  matched_team = []
  for players in list(itertools.product(team_a,team_b,team_c)):
      largest_team = max(players)

      l_combinations = list(players)
      l_combinations.pop(l_combinations.index(max(l_combinations)))
      remaining_players = sum(l_combinations)

      if remaining_players == largest_team:
          matched_team.append(1)
      else:
          matched_team.append(0)

  fraction_success = sum(matched_team) / len(matched_team)
  return(fraction_success)

answer_express = find_matches_fraction(6)
print(answer_express)
## 0.24
```

And in Julia

```
using IterTools

function find_matches_fraction_jl(max_players)
  team_a = 1:max_players
```

```
  team_b = 1:max_players
  team_c = 1:max_players

  matched_teams = []

  for players in product(team_a, team_b, team_c)
    largest_team = maximum(players)
    other_teams = collect(players)
    deleteat!(other_teams, argmax(players))
    remaining_players = sum(other_teams)

    if largest_team == remaining_players
      push!(matched_teams, 1)
    else
      push!(matched_teams, 0)
    end
  end

  fraction_success = sum(matched_teams) / length(matched_teams)
  return fraction_success
end
## find_matches_fraction_jl (generic function with 1 method)

answer_express = find_matches_fraction_jl(5);
answer_express
## 0.24
```

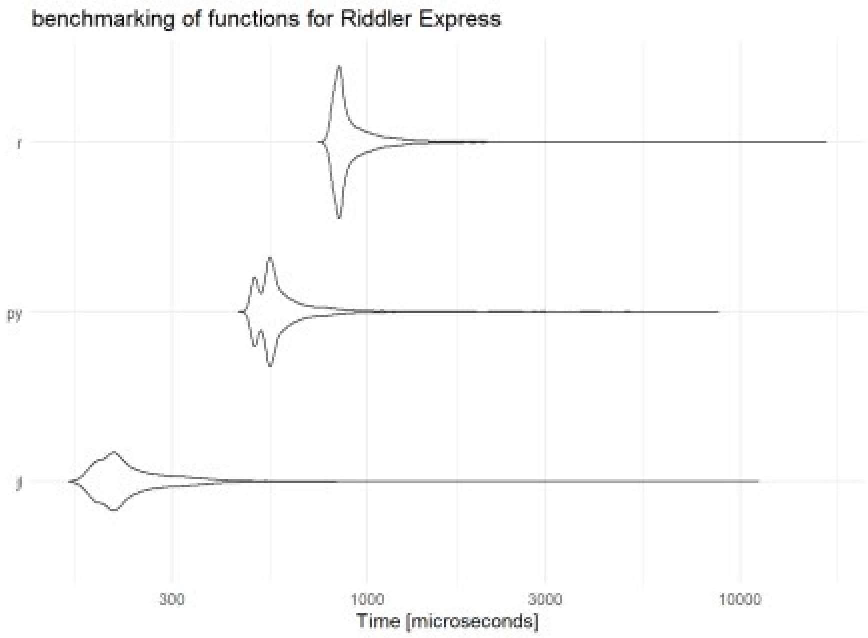
We can also run these chunks in R using reticulate and JuliaCall

```
#packages to call other languages into R
library(JuliaCall)
library(reticulate)

#run the functions to check answers
py$find_matches_fraction(as.integer(6))
## [1] 0.24
julia_eval("find_matches_fraction_jl(5)")
## [1] 0.24
```

We can then use microbenchmark to test the speeds of the functions written here. We run each n times and look at the distribution of times spent running each.

```
#microbenchmark to time functions
library(microbenchmark)

#run each function 10000 times
n <- 10000
bench_express <- microbenchmark(
  jl = julia_eval("find_matches_fraction_jl(5)"),
  py = py$find_matches_fraction(as.integer(6)),
  r = find_matches_fraction(5),
  times = n
)
```

```
#plot the speeds of each functions
p2 <- ggplot2::autoplot(bench_express) +
  labs(
    title = "benchmarking of functions for Riddler Express"
  ) +
  theme_minimal()

p2
```



I’m pretty happy with that. Even my rusty python ends up being faster than the R code (which I wrote for expressiveness and not speed per se), but my first ever solution in Julia outstrips both!

Classic

This month, the Tour de France is back, and so is the Tour de FiveThirtyEight!

For every mountain in the Tour de FiveThirtyEight, the first few riders to reach the summit are awarded points. The rider with the most such points at the end of the Tour is named “King of the Mountains” and gets to wear a special polka dot jersey.

At the moment, you are racing against three other riders up one of the mountains. The first rider over the top gets 5 points, the second rider gets 3, the third rider gets 2, and the fourth rider gets 1.

All four of you are of equal ability — that is, under normal circumstances, you all have an equal chance of reaching the summit first. But there’s a catch — two of your competitors are on the same team. Teammates are able to work together, drafting and setting a tempo up the mountain. Whichever teammate happens to be slower on the climb will get a boost from their faster teammate, and the two of them will both reach the summit at the faster teammate’s time.

As a lone rider, the odds may be stacked against you. In your quest for the polka dot jersey, how many points can you expect to win on this mountain, on average?

A quick guess can be gotten by assuming there were *no* teams and just taking the expected points after random assignment

```
riders <- 4
points <- c(5,3,2,1)

sum(points/riders)
## [1] 2.75
```

We can then work out the answer to the classic analytically by calculating the chance that the rider is bumped back a spot for any position they find themselves in. For instance, if they finish 2nd, there is a 1 in 2 chance the rider ahead of them is part of the team, which would bump our rider into 3rd to make run for the teammate.

```
expected_points <-
  #first
  (points[1] / riders) +
  #second
  (points[2] / riders)/(riders-1) + 2 * (points[(riders-1)] /
riders)/(riders-1) +
  #third
  (points[(riders-1)] / riders) / (riders-1) + 2 *(points[riders] /
riders)/(riders-1) +
  #last
  (points[riders] / riders)

expected_points
## [1] 2.416667
```

So we have our answer, but what about for any combination of team and points? We can write an R function to assign riders to teams and simulating many races to get an estimate of the total points. We could again solve these analytically, but that wouldn’t really benefit my programming.

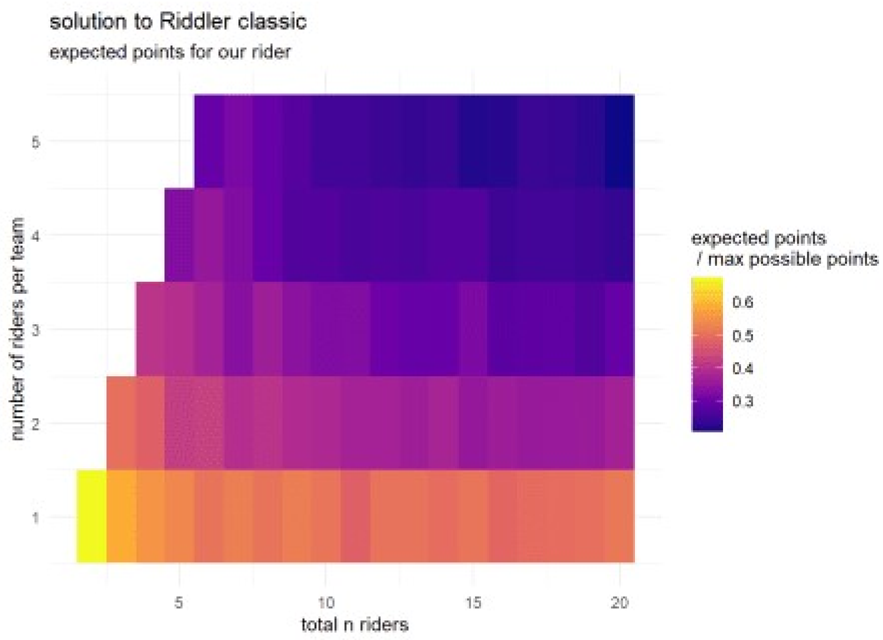
```
get_team_points <- function(teams, points) {
  team_pos <- sample(unique(teams), length(unique(teams)), prob =
table(teams))
  all_positions <- unlist(lapply(team_pos, function(p) rep(p,
length(which(p == teams)))))
  team_points <- lapply(unique(teams), function(i)
sum(points[which(all_positions == i)]))
  names(team_points) <- unique(teams)
  return(team_points)
}

sim_race <- function(n_riders, n_per_team = 2, points = c(5,3,2,1),
times = 1000) {
  leftover_riders <- (n_riders-1) %% n_per_team

  teams <- (n_riders - leftover_riders - 1) / n_per_team

  teamed_riders <- c(
    rep(seq(teams), each = n_per_team),
    rep(max(teams)+1, leftover_riders),
    999
  )
```

```
  all_points <- c(
    points,
    rep(0, n_riders - length(points))
  )

  simmed_points <- unlist(purrr::rerun(times,
get_team_points(teamed_riders, all_points)))
  expected_points <- tapply(simmed_points, names(simmed_points), sum) /
times
  expected_points[names(expected_points) == 999]
}

expected_points <- sim_race(4, 2, points = c(5,3,2,1), times = 10000)
expected_points
##    999
## 2.4093
```

For a range of n riders and team sizes, we can calculate our riders expected points per race (we'll use the same point structure of c(1:n-1, n+1)) for a little extra flourish

```
riders <- 1:20
n_per_team <- 1:5

library(dplyr)

arguments <- expand.grid(riders, n_per_team) %>%
  dplyr::rename(n_riders = Var1, n_per_team = Var2) %>%
  #must be more riders than riders per team
  dplyr::filter(n_riders > n_per_team)
arguments$points <- lapply(arguments$n_riders, function(r) c(r+1,
(r-1):1))

#use map2
library(purrr)

sims <- 1000
arguments$expected_points <- pmap_dbl(arguments, sim_race, times =
sims)

#plot the expected points
p3 <- ggplot(arguments, aes(x = n_riders, y = n_per_team)) +
  geom_tile(aes(fill = expected_points / (n_riders+1))) +
  scale_fill_viridis_c(option = "plasma", name = "expected points\n /
max possible points") +
  labs(
    title = "solution to Riddler classic",
    subtitle = "expected points for our rider",
    x = "total n riders",
    y = "number of riders per team"
  ) +
  theme_minimal()
```

p3



Lets now port our function for this over the python…

```
from numpy.random import choice
import numpy as np
import pandas as pd
import math
import itertools

def sim_race_py(n_riders, n_per_team, points):
  n_teams = math.ceil((n_riders - 1) / n_per_team) + 1
  filled_teams = math.floor((n_riders - 1) / n_per_team)
  leftover_riders = (n_riders - 1) % n_per_team

  if leftover_riders > 0:
    extra_riders = [leftover_riders, 1]
  else:
    extra_riders = 1

  if filled_teams == 1:
    win_prob = [n_per_team, extra_riders]
  else:
    win_prob = [n_per_team] * filled_teams
    win_prob.extend([extra_riders])
  flattened_probs = list(pd.core.common.flatten(win_prob))

  sum_probs = np.sum(flattened_probs)
  adjusted_probs = [p/sum_probs for p in flattened_probs]

  no_teams = list(range(len(flattened_probs)))

  finish_order = choice(no_teams, len(no_teams), p = adjusted_probs,
replace = False)
```

```
  expanded_finish_order = []
  for team in finish_order:
   if team < filled_teams:
     expanded_finish_order += [team] * n_per_team
   else:
      if team != max(no_teams):
        expanded_finish_order += [team] * leftover_riders
      else:
        expanded_finish_order += [team]

  won_points = points[np.argmax(expanded_finish_order)]
  return won_points

def sim_races_py(n_riders, n_per_team, points, n_times):
  won_points = []
  for _ in range(n_times):
    sim_points = sim_race_py(n_riders, n_per_team, points)
    won_points.append(sim_points)
    expected_points = np.sum(won_points) / len(won_points)

  return(expected_points)

answer_classic = sim_races_py(4,2,[5,3,2,1], 10000)
print(answer_classic)
## 2.4286
```

…and in Julia

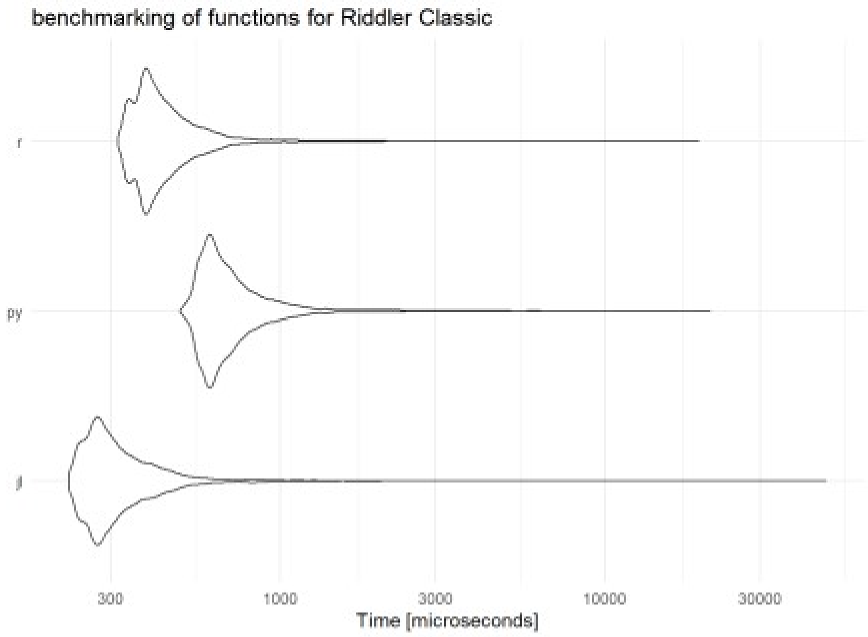
```
using StatsBase

function sim_race_jl(n_riders, n_per_team, points);
  n_teams = Int(ceil((n_riders - 1) / n_per_team));
  filled_teams = Int(floor((n_riders - 1) / n_per_team));
  leftover_riders = mod(n_riders - 1, n_per_team);

  if leftover_riders > 0
    extra_riders = [leftover_riders, 1];
  else
    extra_riders = 1;
  end

  if filled_teams == 1
    win_prob = vcat(n_per_team, extra_riders);
  else
    win_prob = vcat(repeat([n_per_team], filled_teams), extra_riders);
  end

  finish_order = sample(1:length(win_prob),
                        ProbabilityWeights(win_prob),
                        length(win_prob),
                        replace = false
  );
```

```
  expanded_finish_order = Vector{Int}();
  for team in finish_order
    if team <= filled_teams
       append!(expanded_finish_order, repeat([team], n_per_team));
    else
      if team != length(finish_order)
        append!(expanded_finish_order, repeat([team],
leftover_riders));
      else
        append!(expanded_finish_order, team);
      end
    end
  end

  rider_position = findall(expanded_finish_order .==
maximum(expanded_finish_order));
  points_won = points[rider_position];
return points_won
end
## sim_race_jl (generic function with 1 method)

function sim_races_jl(n_riders, n_per_team, points, n_times);
  won_points = Vector{Int}();

  for _ in 1:n_times
    sim_points = sim_race_jl(n_riders, n_per_team, points);
    append!(won_points, sim_points);
  end

  expected_points = sum(won_points) / length(won_points);

  return expected_points;
end
## sim_races_jl (generic function with 1 method)

answer_classic = sim_races_jl(4,2,[5,3,2,1], 10000);
answer_classic
## 2.4098
```

And then lets benchmark each of these functions again

```
#run each function 10000 times
n <- 10000

bench_classic <- microbenchmark(
  jl = julia_eval("sim_race_jl(4,2,[5,3,2,1])"),
  py = py$sim_race_py(as.integer(4),as.integer(2),c(5,3,2,1)),
  r = sim_race(4,2,c(5,3,2,1), times = 1),
  times = n
)
```

```
#plot the speeds of each functions
p4 <- ggplot2::autoplot(bench_classic) +
  labs(
    title = "benchmarking of functions for Riddler Classic"
  ) +
  theme_minimal()

p4
```



A bit closer this time. I think I haven't quite got efficiency for more involved functions down for python and Julia. Julia still wins this round but I feel could be speed up by at least a factor 2 or 3x.