EE 325 Programming Assignment Report

Vedant Bhardwaj - 23B0068

September 12, 2024

Question 1

Catch m fish, mark them, and release them back into the lake. Allow the fish to mix well and then you catch m fish. Of these, p are those that were marked before. Assume that the actual fish population in the lake is n and has not changed between the catches.

1.1 Probability that p of the m caught fishes are marked

Number of unmarked fish in the lake is $n - m$.

Number of ways of choosing p fish from the m marked ones is $\binom{m}{p}$.

Number of ways to choose $m - p$ fish from the unmarked ones is $\binom{n-m}{m-p}$.

So the total number of ways of choosing combinations of m fishes with p being marked is $\binom{m}{p} \cdot \binom{n-m}{m-p}$.

The total number of ways m fish from the lake can be chosen is $\binom{n}{m}$.

So the probability of getting p of the m fishes marked with n fishes in the lake is :-

$$\boxed{Pr(p \text{ marked fish out of m chosen}) = \frac{\binom{m}{p} \cdot \binom{n-m}{m-p}}{\binom{n}{m}}}$$

1.2 Notion of best guess

Using the above equation we can easily create a function of n. Since m and p values are given to us, we can consider them as constants and vary the n.

Substituting for $m = m'$ and $p = p'$, we get the probability as a function of n:

$$\boxed{P(n) = \frac{\binom{m'}{p'} \cdot \binom{n-m'}{m'-p'}}{\binom{n}{m'}}}$$

Notion of best guess:

The best guess as per me would be the point of highest probability at given n. That n would be the best guess.

To do that I coded it over all the 4 cases and iterated over n=1 to n=1000.

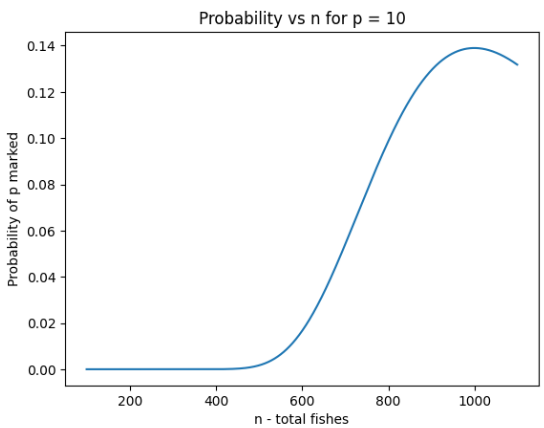
1.3 Source-code

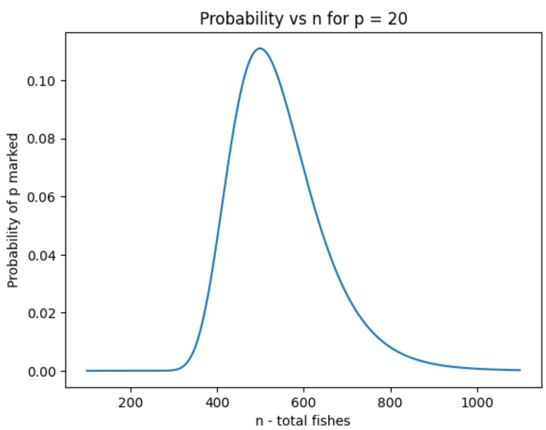
```
size = 1000 for j in
p:
  n = np. zeros ( size ) prob = np. zeros ( size )
  lognumerator = np. zeros ( size )
  logdenominator = np. zeros ( size ) logprob
  = np. zeros ( size )
  c = m
  maxprob = 0 maxn=0 for i in
  range(1000):
    n[ i ] = c
        prob [ i ] = (math.comb(m,              j ) ∗ math.comb( int (n[ i ])−m, m−j ))/

    (math.comb(  int  (n[  i  ])  ,  m))  if
    prob  [ i ] > max _prob : maxprob =
    prob [ i ] max_n = n[ i ]
    c += 1
  print( 'maximum probabilty is at n = {} ' . format(max _n)) The calculated values
     were:
```

S.no	m	p	n
1	100	10	999
2	100	20	499
3	100	50	199
4	100	75	133

1.4 Graphs





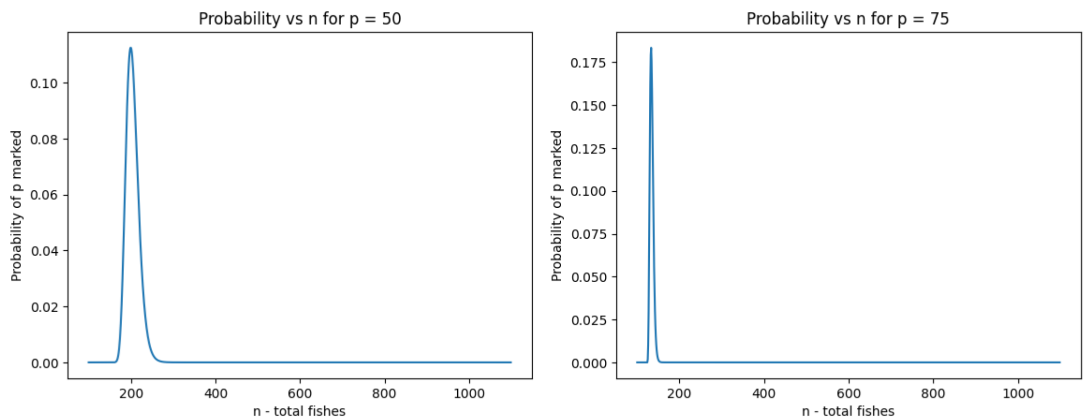


Figure 1: Graphs of Probability versus n for m = 100 and p = 10,20,50,75.

Question 2

In this question, there is infinite memory in the system and can accommodate any number of packets.

At first, there is a packet in the buffer memory. The packet leaves with a probability of 0.3 and another packet enters the buffer with a probability of 0.3 We take λ = 0.3 and μ = 0.4.

2.1 Solution

1. At every time step, we check if the number of packets in the queue is zeroor non-zero. A packet is removed with probability μ = 0.4 if it is greater than zero. And if it is zero, no removal is done.
2. At every time step, a new packet is added with probability λ = 0.3
3. **Now what are we supposed to do?**

We are supposed to find the number of packets in the buffer at which fraction time is the highest. We do this for a total of 1,00,000 time steps.

I created a list that max-timecount, this will measure the time given to each queue length possible.

3. We store the number of packets in the queue after step 2 in an array called queue history.

These steps are repeated for 1,000,000 times, storing the number of packets at each timestamp.

2.2 Formulation

1. The number of time stamps for which the queue has size n is calculated for each n = 0,1,2,...,50. This is divided by the total number of time stamps (1,000,000) to get the probability of the queue size being n.

$$P(n) = \frac{\text{No. of times queue had size } n}{\text{Total number of time stamps}}$$

2. The time average of the number of packets, i.e., the average value of thenumber of packets in the queue is calculated.

$$\text{Average queue size} = \frac{\text{Sum of the number of packets in the queue for all time stamps}}{\text{Total number of time stamps}}$$

The maximum fraction of time is at n = 1

Time average number of packets in memory (Single Queue): 2.09

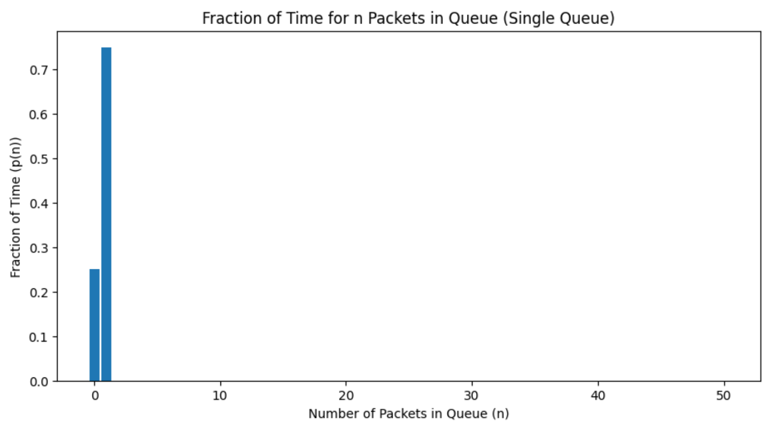
2.3 Source-code and error analysis

```
p = 0.4 q = 0.3 time steps =
1000000
max n = 50
queue length = 0 time count = np. zeros (max
_n + 1) for i in range ( timesteps ): t = np.
random . rand () i f t < p and queuelength >0:
queue _length −= 1

    i f    t < q :
       queue _length += 1
    i f        queue length <= max n :
       time count [ queue length ] += 1 p _n =

time count / time steps
```

The above code you see is faulty. It doesn't work why? There's one very good reason for it that's why I kept my faulty code too in the report



Above is the plot for the faulty code.

Reason: This will not be independent. Why will this not be independent? Because here we are calculating random value before hand but probability dosen't work like that. It like tossing a coin. If my I want to remove a packet and add a packet with some probabilities given to both of them.

Now since both of them have probabilities, getting head on the coin for removing event doesn't decide for adding event. And if you do so that means your event are not independent. Since they are, we must use np.random.rand for each part. Now the better one is here:

```
p = 0.4 q = 0.3 timesteps =
1000000
max n = 50
queue length = 0 time count = np. zeros (max
_n + 1) max timecountt = 0 max timecount_c
= 0 for i in range ( time _steps ):
    i f np. random . rand () < p and queue _length >0: queue _length −=
        1

    i f      np. random . rand () < q :
        queue length += 1

    i f queue  length  <=  max  n  :  time  count  [
        queue length ] += 1
        i f time count [ queue length ] > max timecount _t: max timecount
t = time count [ queue length ] max timecount c = queue length p _n =
time count / time _steps
```

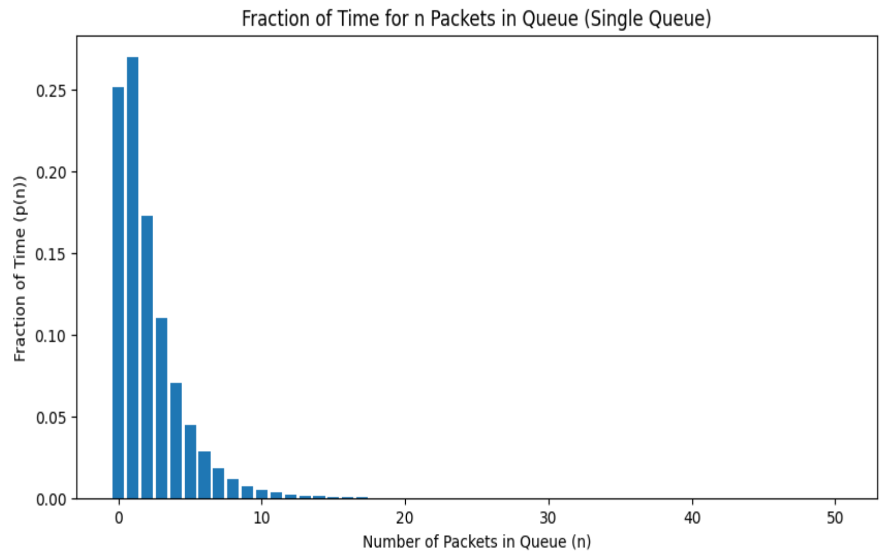


Figure 2: $P(n)$ vs n

Question 3

The program from the previous part is extended to simulate 10,000 queues simultaneously in parallel. When we stop the simulation after 100,000 time steps, we have 10,000 values for the number of packets in the system.

3.1 Solution

- The simulation is done similarly to the previous question by taking 10,000 queues and updating the number of packets in them in parallel by using np.random.rand() for each queue separately in each step.

We create two for loops and then subtract or add packets based on the probability. And then we do np.bincount and then divide by total time steps to average it out.

3.2 Formulation

1. We calculate the number of queues with size n at the end of the simulation and divide it by the total number of queues to get the probability of a queue having size n for $n = 0,1,2,...,50$, at the end of the simulation.
2. We also calculate the average value of the number of packets in a queueafter the simulation.

$$\text{Average queue size} = \sum_{n=0}^{50} P(n) \cdot n$$

$$P(n) = \frac{\text{The number of queues with size } n}{\text{Total number of queues}}$$

Sample average number of packets in memory (Parallel Queues): 2.13

3.3 Source-code

```
queues = np. zeros (num _queues , dtype=int ) time steps _2 =
100000
max n = 50

forin range ( time steps 2 ): for i in range
        (num queues ):
            i f         queues [ i ] > 0 and np. random . rand () < p:
                queues [ i ] −= 1
            i f np. random . rand () < q : queues [ i ] +=
                1

packet counts _parallel = np. bincount (queues , minlength=max _n + 1) p n parallel = packet _
counts _parallel / num _queues
```

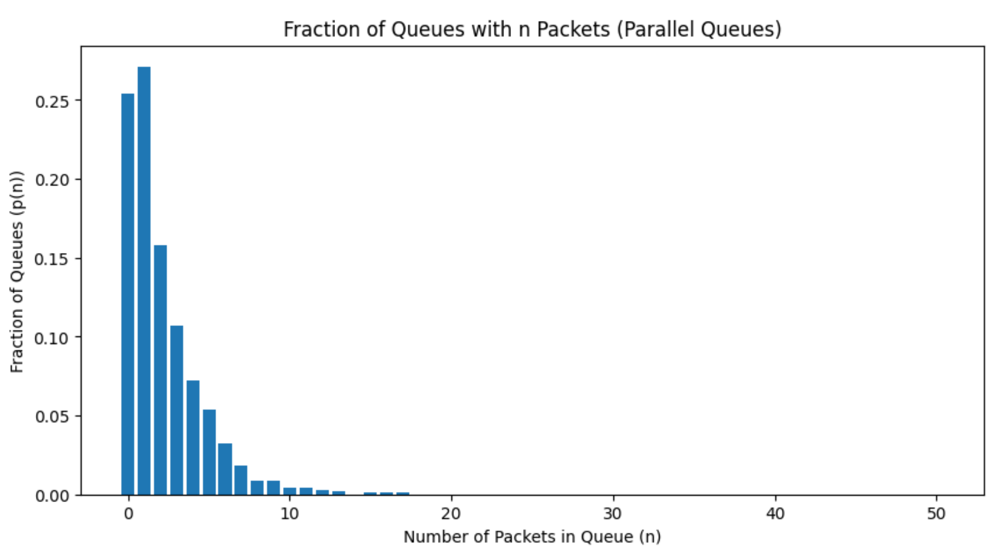


Figure 4: $P(n)$ vs n

Question 4

Let there be N members in the jury. Each judge makes a yes/no decision, and the jury's final decision is the decision with the majority of votes. The probability of an individual jury member making the right decision is $p = 0.5+c$, where c is discretized between 0.05 and 0.25 in steps of 0.01, i.e., c can take values 0.05, 0.06, 0.07, ..., 0.25.

Let X be a discrete random variable that denotes the number of judges who took the right decision. X can take values $0,1,2,...,N$. The probability that exactly m of N judges make the correct decision is

$$Pr(X = m) = \binom{N}{m} p^m (1-p)^{N-m}$$

The jury's final decision is the decision with the majority votes, i.e., the decision with at least k votes, where k = (N//2 +1).

Therefore, the probability that the jury makes the correct decision is the probability that the number of judges who make the correct decision is at least k, i.e., $Pr(X \geq k)$.

As $X = k, X = k + 1, X = k + 2, ...$ are mutually exclusive events,

$$Pr(X \geq k) = Pr(X = k) + Pr(X = k + 1) + ... + Pr(X = N)$$

$$Pr(X \geq k) = \binom{N}{k} p^k (1-p)^{N-k} + \binom{N}{k+1} p^{k+1} (1-p)^{N-k-1} + \ldots + \binom{N}{N} p^N (1-p)^{N-N}$$

where $p = 0.5 + c$.

4.1 Finding (N,c) combinations

We at first decide the range of N and c. I chose a range of N from (1, 101). 101 is pretty high considering it is just a jury. Anything after 20 is not practical. But I wanted to have a rough idea of the values that's why I kept it that high.

The range of c was already given. Now what I did is that for different values of (N,c) I checked if the probability is higher than the cap (in this case 0.75).

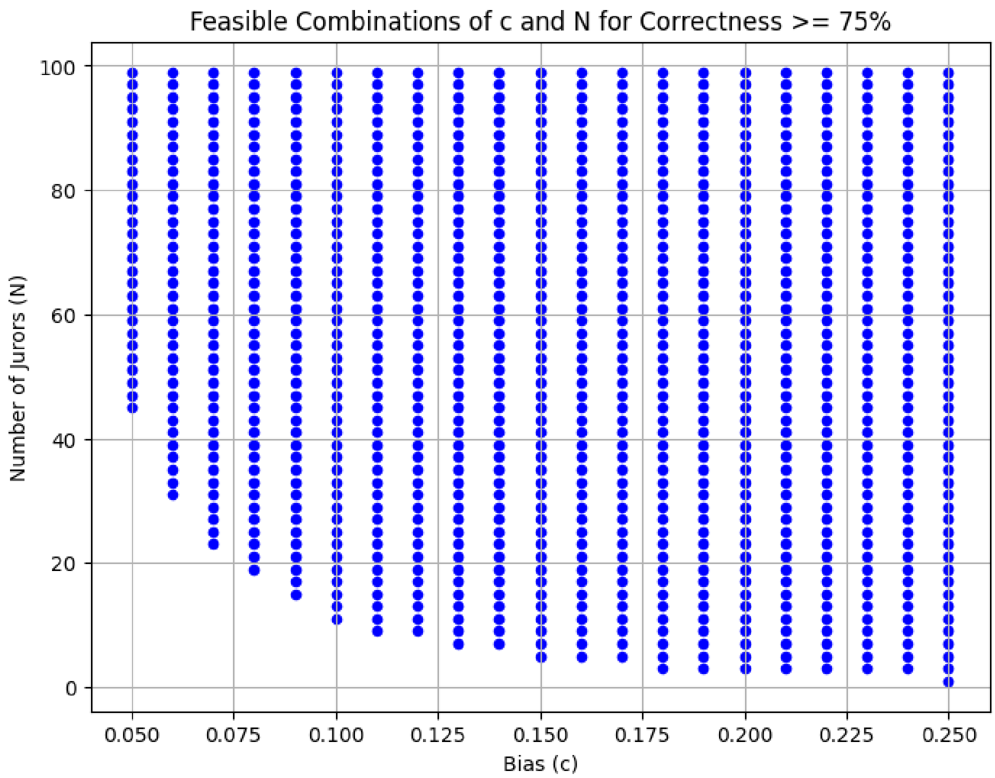


Figure 6: *c* values for *N* with a minimum probability of 0.75.

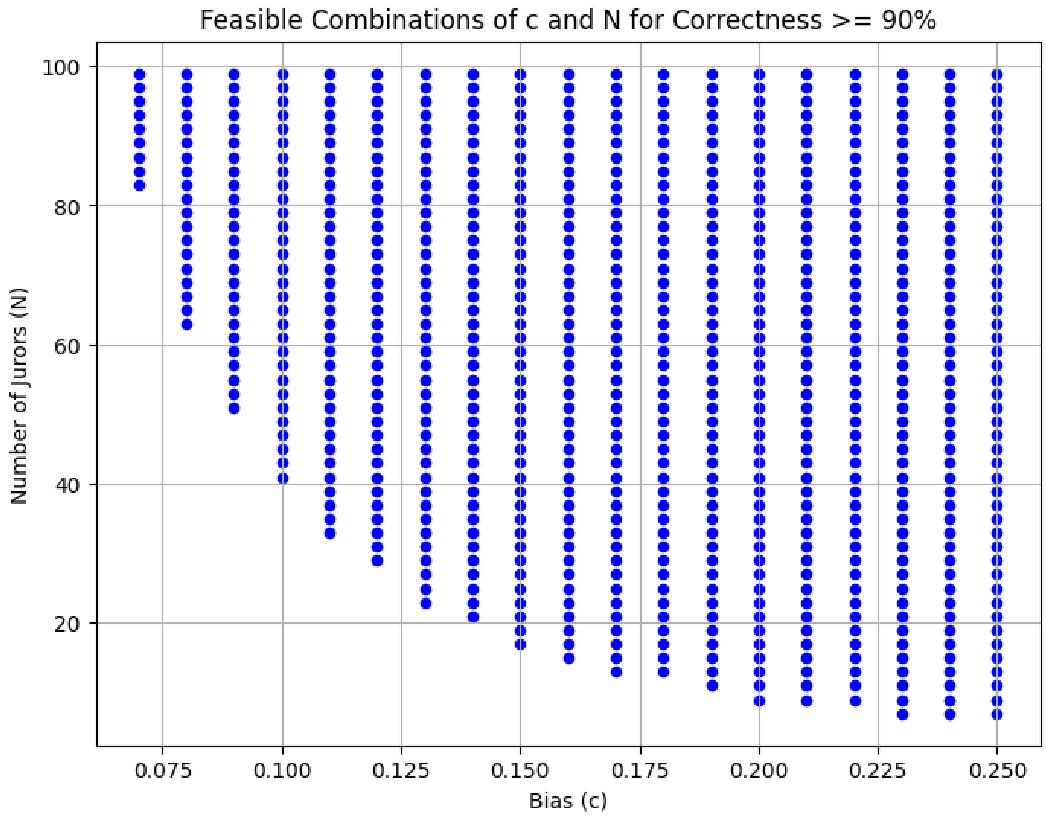


Figure 7: c values for N with a minimum probability of 0.90.

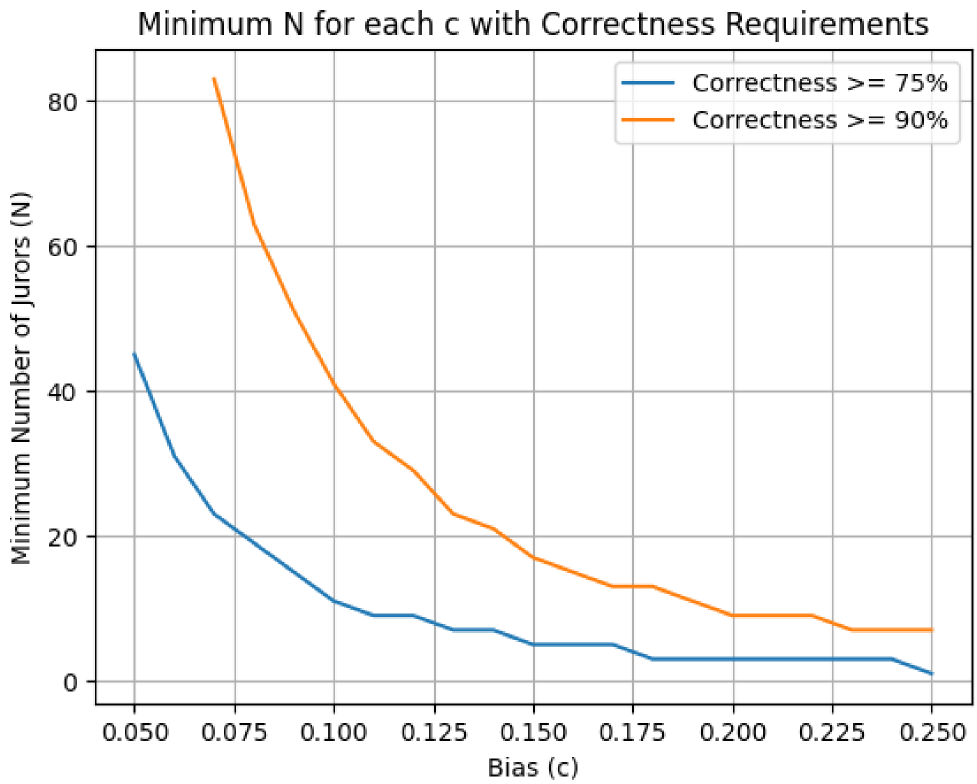


Figure 8: Both the graphs in one plot for comparison.

These are the min values and those (N,c) combinations are entire space (obviously odd values of N) above the curve of each probability.

4.2 Source-code

```
cvalues = np. arange (0.05 ,        0.26 ,    0.01) N _
values = np. arange (1 , 101 , 2) correctness
requirements = [0.75 ,     0.9]

def jury correct _decision _probability (N, p ): k = (N // 2) + 1
        probability = 0.0 for i in range (k ,
        N + 1):
                probability += comb(N, i ) ∗ (p ∗∗ i ) ∗ ((1 − p) ∗∗ (N − i )) return probability feasible

combinations = {requirement : [ ] for requirement in correctness requirements}
```

```
for c in c values : p = 0.5 + c for
    N in N values : _
        prob correct = jury _correct decision _probability (N, p) for requirement in
        correctness requirements :                              _
            i f prob correct >= requirement : feasible combinations [ requirement ] .
                append ((c , N))
```

4.2 Cost function

The cost function increases with an increase in the number of judges and also increases if c increases which means the probability of individuals making fair decisions are tough to find.

So the cost function will look like:

$$\text{cost}(N,c) = aN + bp = aN + b(0.5 + c)$$

where $a = 1$ and $B = 80$ are proportionality constants.

Now the question comes - where do these values come from?

The answer to this question is subjective. For me, it was some trade-off between c and N. My values of N for both cases are higher than usual. For 0.75 it's 5 and for 0.9 it's 9. Like in these cases values of c are higher. That's why the tradeoff.

Code

```
                                                   _
cvalues = np. arange (0.05 ,      0.26 ,    0.01)   N  _
values = np. arange (1 , 101 , 2)

a = 1 b =
80   def
jury    _
correct

_
decisio
n       _
probabi
lity  (N,
p ): k =
(N // 2)
+ 1
    probability = 0.0 for i in range (k ,
    N + 1):
```

probability += comb(N, i) ∗ (p ∗∗ i) ∗ ((1 − p) ∗∗ (N − i)) return probability

def compute _cost (a , b, N, c): p = 0.5 + c
return a ∗ N + b ∗ p

def _ find min N for each c (correctness _requirement):
min N for c = [ ] for c in c
values : p = 0.5 + c min N
= None
for N in N values :
prob correct = jury _correct decision _probability (N, p)
i f prob correct >= correctness requirement : min N = N
break i f min N is not None:
min N for c . append ((c , min _N))
return min N for c

correctness requirement = 0.75 min N for c = find min N for each c (correctness _requirement) df = pd.

DataFrame(min N for c , columns=['c ' , 'N']) df [' Cost '] = df . apply (lambda row : compute _cost (a ,

b, row ['N'] , row [' c ']) , axis=1)

The calculated costs were:
The minimum costs in each case are:

	c	N	Cost
0	0.05	45	89.0
1	0.06	31	75.8
2	0.07	23	68.6
3	0.08	19	65.4
4	0.09	15	62.2
5	0.10	11	59.0
6	0.11	9	57.8
7	0.12	9	58.6
8	0.13	7	57.4
9	0.14	7	58.2
10	0.15	5	57.0
11	0.16	5	57.8
12	0.17	5	58.6
13	0.18	3	57.4
14	0.19	3	58.2
15	0.20	3	59.0
16	0.21	3	59.8
17	0.22	3	60.6
18	0.23	3	61.4
19	0.24	3	62.2
20	0.25	1	61.0

Minimum Cost Details:
Minimum Cost: 57.0000
Associated Bias (c): 0.15
Associated Jurors (N): 5.0

	c	N	Cost
0	0.07	83	128.6
1	0.08	63	109.4
2	0.09	51	98.2
3	0.10	41	89.0
4	0.11	33	81.8
5	0.12	29	78.6
6	0.13	23	73.4
7	0.14	21	72.2
8	0.15	17	69.0
9	0.16	15	67.8
10	0.17	13	66.6
11	0.18	13	67.4
12	0.19	11	66.2
13	0.20	9	65.0
14	0.21	9	65.8
15	0.22	9	66.6
16	0.23	7	65.4
17	0.24	7	66.2
18	0.25	7	67.0

Minimum Cost Details:
Minimum Cost: 65.0000
Associated Bias (c): 0.20
Associated Jurors (N): 9.0

For $P \geq 0.75$ **For $P \geq 0.90$**

Thus, in the $P \geq 0.75$ case, for minimum cost, the number of judges will be N = 5, and the probability of taking the right decision for each of them will be p = 0.65. For the $P \geq 0.90$ case, the values will be N = 9 and p = 0.70 which is considered high in such cases but it is better than hiring more jurors.